Adaptive Real-Time Decoding of Brain Signals for Long-Term Control of a Neuro-Prosthetic Device

Tushar Ashok Dharampal
Marquette University

ADAPTIVE REAL-TIME DECODING OF BRAIN SIGNALS FOR LONG-TERM CONTROL OF A NEURO-PROSTHETIC DEVICE

by

Tushar Dharampal, B.E.

A Thesis submitted to the Faculty of the Graduate School,
Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science

Milwaukee, Wisconsin

August 2011

ABSTRACT
ADAPTIVE REAL-TIME DECODING OF BRAIN SIGNALS FOR LONG-TERM CONTROL OF A NEURO-PROSTHETIC DEVICE

Tushar Dharampal, B.E.

Marquette University, 2011

Changes in the statistical properties of neural signals recorded at the brain-machine interface (BMI) pose significant challenges for accurate long-term control of prostheses interfaced directly with the brain by continuously altering the relationship between neural responses and desired action. In this thesis, we develop and test an adaptive decoding algorithm that can recover from changes in the statistical properties of neural signals within minutes. The adaptive decoding algorithm uses a Kalman filter as part of a dual-filter design to continuously optimize the relationship between the observed neural responses and the desired action of the prosthesis. Performance of the algorithm was evaluated by simulating the encoding of arm movement by neurons in the primary motor cortex under stationary conditions as well as nonstationary conditions depicting loss and/or replacement of neurons in the population. The time taken for the system to fully recover (3-12 minutes) was faster than other adaptive systems (Rotermund et al 2006) and resulted in errors that were well matched to the initial system performance. The algorithm adapts to the instantaneous properties of the stimulus and is able to decode movements with high accuracy outside the trained movement space. This implementation lends itself favorably toward a portable long-term decoding approach at the brain-machine interface capable of providing accurate real-time decoding of neural signals over periods of weeks to months without outside intervention.

ACKNOWLEDGEMENTS

Tushar Dharampal, B.E.

I would like to thank my parents, my wife Priyanka, my brother and my friends. I would like to thank my committee and my thesis director, Dr. Scott Beardsley. I would like to thank the Graduate School and all of the Marquette University administration.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION AND SPECIFIC AIMS

Changes in the statistical properties of neural signals recorded at the brain-machine interface (BMI) pose significant challenges for accurate long-term control of prostheses interfaced directly with the brain by continuously altering the relationship between neural responses and desired action (Schwartz et al, 2006, Rotermund et al, 2006). Prosthesis control algorithms rely on the accuracy of the information carried by these neural signals and optimally use this information to generate motion of the prosthesis as desired by the subject. Due to a variety of phenomenon including neuron loss and/or recruitment, neuroplasticity, and modulation due to attention/adaptation, such changes may manifest as 'nonstationary' signals whose statistical properties (including mean, variance etc.) are not constant. Such changes impact the accuracy of the prosthesis control algorithms (also referred to as 'decoding algorithms') thus requiring that the decoding of neural activity be continuously re-optimized.

Current optimization procedures are typically performed intermittently and are computationally intensive, resulting in degraded performance between sessions. For algorithms that adapt continuously, recovery can take several hours (Rotermund et al, 2006) or may not be easily realized in a portable implementation with current technologies (Rotermund et al, 2006, Srinivasan et al, 2007).

While different approaches to neuronal signal loss and/or changes in recorded neurons over time scales of minutes to days have been investigated both with simulated neural signals (Rotermund et al, 2006) and physiological recordings (Wu et al, 2008), the algorithms have not been tested against the effects of neuro-physiological phenomenon

that occur over short timescales (seconds to minutes) such as attention modulation of neuron responses, neuroplasticity, and neuronal adaptation that could also bring about statistical changes in the neural signals. It is proposed that an adaptive decoding algorithm that is resistant to changes in the statistical properties of the neural signals across temporal scales will provide more accurate decoding of intended movement to actively control prosthetic systems. Therefore the specific aims are:

Aim 1: Identify and characterize the effects of different sources of nonstationarity on non-adaptive decoding of neuronal signals in a simulated population of neurons.

Aim 2: Design and implement an adaptive decoding algorithm that is resistant to nonstationary changes in neural signals and validate its performance using simulated datasets.

Aim 3: Compare the performance of the proposed algorithm against current approaches and evaluate its potential implementation in a portable system.

2 BACKGROUND AND SIGNIFICANCE

2.1 Neuromotor Prostheses

Neuromotor prostheses are a subset of cortical neuroprostheses that replicate lost limb function for patients with intact cortical areas but disabled motor pathways or end effectors (Schwartz A. B. 2004). These may include amputees, patients with muscular dystrophy and paralysis patients. Either invasive (cortical implants) or noninvasive methods (Electroencephalography recordings) of recording neural data may be used in such prostheses to establish desired limb movement.

Invasive neuromotor prostheses are made up of three essential components – the artificial limb (end effector), cortical implant and neuronal decoding system. In an ideal system, brain signals (single or multi-unit neuron recordings) from the relevant cortical area (e.g., pre-motor or motor cortex) are collected using the cortical implant and passed to the decoding system that estimates the intended movement parameters (e.g., velocity, position) to control the artificial limb based on the recorded neuronal responses (Lebedev M. A., et al. 2006; Schwartz A. B. 2004; Schwartz A. B., et al. 2006) .

The most commonly used electrode implant is the Utah Array (Maynard E. M., et al. 1997). The implant contains 100 electrodes placed in a 10x10 grid (on a 4mm x 4mm surface), with each electrode capable of recording action potentials from 1 – 3 neurons. Through an invasive procedure, the electrode is placed directly on the surface of the cerebral cortex in the cortical area considered most relevant to the task. Typically, wires carrying data to the control system pass from the electrode and transcutaneously through

the skull to the control system. The control system is in turn connected to the actuators that drive the prosthesis. The control system typically consists of a digital microprocessor based system that runs a mathematical decoding algorithm to map movement-related activity in the brain to the specific control signals used to drive the prosthesis. Parameters (coefficients) of the decoding algorithm are trained/optimized over a training session(s) involving repeated movements within a predetermined training space so that both the patient and the algorithm learn the space and the use of the limb. Typically, such coefficients are determined using an error minimization technique (e.g. Least Squares Minimization) to associate the activity of a recorded neuron with a particular type of limb movement. Each neuron typically has a higher response to a preferred movement direction (or set of directions) and increases its activity when the desired limb movement is in that direction. The learning or optimization technique associates a higher coefficient with the neuron when the movement is in the neuron's preferred direction. These coefficients (weights) are then used by the decoding algorithm to decode intended movement from the neuronal responses (Hochberg L. R., et al. 2006; Kalaska J. F. 2008; Schwartz A. B., et al. 2006).

2.2 Nonstationary Neural Responses

For invasive neuromotor prosthesis, the electrode is designed to be implanted for a prolonged period of time. During long term implantation of the electrode, a number of different processes can occur at the neuron – electrode interface that can influence the quality of the signals being recorded. These include biochemical processes (electrode immunological response), mechanical (movement/migration of electrode) and cognitive

processes (attention and adaptation). Each of these processes introduces unwanted changes in the neuron recordings that impact the statistics of the recorded data. Such changes are referred to as nonstationary changes or nonstationarities.

In non-adaptive decoding algorithms, the weighting coefficients are optimized to the statistics of the recorded neuronal ensemble so as to minimize the overall error in the decoded movement. These weighting coefficients are used to obtain the movement variables from the encoded neural responses. Any changes in these statistical properties for a neuron population such as mean and variance result in non-optimal decoding and create undesired errors in the decoded movement.

2.3 Improving Neuronal Recordings

Coating the electrode with materials that encourage neuron growth or reduce inflammation at the site of implantation have been developed to improve recording performance. MEMS (micro-electro-mechanical system) electrodes with changeable depth and algorithms that position the electrode automatically have been shown to facilitate neuron recordings. However, the desired recording performance (recording from an adequate sample of neurons) is not typically sustained for the intended period (at least two – three years) (Kalaska J. F. 2008; Lebedev M. A., et al. 2006). Due to biological processes such as death of the neuron cells or dead tissue surrounding the electrode, the number of suitable recorded units changes over time with neurons dropping out and being replaced by other neurons. Physical and chemical solutions may alleviate the errors in performance due to biochemical and mechanical nonstationarities (such as scar tissue formation or electrode movement) but they do not correct for cognitive

processes that are intrinsic to the neuron or neuronal system. Additionally, they may increase the complexity of the implant procedure and the size of the implant. Finally, a specific method to deal with each nonstationarity-inducing process may be needed when such solutions are employed. An algorithmic solution may be easier to implement using current technology without increasing cost/size of the implant or taxing the implant procedure. Once programmed, the algorithmic solution would run continuously in the background, reducing movement error regardless of its source.

2.4 Neural Decoding

The decoding algorithm is a mathematical relationship that relates neuronal response to the desired movement parameters (e.g. movement velocity or position). It is based upon neuron responses that are parameterized in the movement space. For the purposes of decoding, these neuron responses are computed over a small time interval and are related to the decoding weights established during an initial optimization process. The decoding weights or coefficients are used by the algorithm to obtain movement information from the neuron responses (e.g. firing rates).

To obtain movement information from neural recordings, a wide variety of decoding algorithms such as linear filters, Kalman filters (Wu et al, 2002, 2008; Gage et, 2004, 2005) and Bayesian decoders (Rotermund et al, 2006) among others have been employed. For e.g, in a Kalman decoding approach, the movement parameters can be modeled as the Kalman state variables that are estimated by the Kalman filter. The neural responses can be modeled as the output of the system. An initial optimization process establishes the decoding coefficients of the filter (Kalman weights). During decoding, the

Kalman weights are used by the filter to obtain estimates of the state variables (movement variables). The Kalman filter does this in a two step process – by making a prediction of the movement and then correcting its estimate to minimize error (Wu et al, 2002).

2.5 Adaptive Decoding of Movement

When the source and/or quality of the neurons is affected by any undesirable biological, physical or biochemical processes (growth of scar tissue, movement of electrode, etc.) over the long term, the decoding parameters learnt by the algorithm may no longer be valid. Thus, nonstationary changes in neuronal signals may manifest themselves in the erroneous prediction of intended movement by the decoding algorithm. For the algorithm to cope with changes in the statistics of neuronal signals, corresponding changes to the decoding weights need to be made. This calls for an adaptive algorithm that updates the weights when it detects the presence of a nonstationary change in the neuronal signals.

Typically, the decoding system is re-optimized before a decoding session within a laboratory or clinical environment. To achieve this, the subject with the implant may be asked to perform a set of pre-determined movements in the training space while the sampled neuron activity is recorded. Using a mathematical optimization procedure (as described above), the decoding coefficients are determined to minimize error in the decoded movement. These optimization procedures are intermittent and computationally intensive, resulting in degraded performance between sessions and limited portability for the patients (Rotermund D., et al. 2006) as they have to periodically revisit the

laboratories to maintain decoding accuracy. Since the long term goal is for the patient to be unconstrained by the assistive device, it is important that the adaptive decoding system be portable (Kalaska J. F. 2008).

2.6 Adaptive Decoding Algorithms in Literature

A number of decoding algorithms have been employed for estimating movement – these include linear filters (Paninski et al., 2001), neural networks (Wessberg et al, 2000), classifier algorithms (Isaacs et al, 2000), Kalman filter algorithms (Wu et al, 2002, 2008; Gage et, 2004, 2005) and Bayesian decoders (Rotermund D., et al. 2006). Ideally, a neural decoding algorithm would operate in real-time and be implemented in a portable system (i.e. with low power, computing and memory requirements). Thus, speed and ease of computation along with accurate prediction of movement are desired (Kalaska J. F. 2008; Lebedev M. A., et al. 2006; Schwartz A. B., et al. 2006).

Gage et al (Gage et al 2004; 2005) developed a 'co-adaptive' decoding filter based on a Kalman filter design that adjusts to changes in the measured neuronal activity as rats learn to control an auditory device during an auditory frequency-matching task. Kalman filter weights were used to decode an auditory signal from the neuronal ensemble that was matched to a test tone. The subject and the filter were naïve to the task and learnt how to perform the task over time. A sliding window consisting of ten trials (900 ms) was used to update the filter weights. Subsequent re-optimization of the weights was achieved during adaptation, when the Kalman filter weights were intermittently re-optimized using the past 45 seconds of auditory signal (frequency). Such re-optimization is contingent on the space in which the errors driving the adaptation are defined and is not

easily extrapolated beyond this space. The estimate of auditory frequency made by the Kalman filter was fed back to the rats and with rewards offered for correct trials, the rats adopted a strategy to minimize the auditory errors.

While error signals may be derived from brain areas or using external sensors and localizers, it may not always be possible to obtain errors represented in terms of the decoded movement parameters. The temporal history used in re-optimizing the system may place a lower bound on the speed at which the system can recover by requiring that nonstationary changes in the signal move beyond the re-optimization window (e.g. 45 seconds). Also, the nature of the neuron ensemble encoding the task-relevant information drives the selection of the time window over which adaptation occurs. For example, responses from a small population of neurons responding to the task would result in sparse data and consequently require a longer time window.

Eden et al (Eden et al 2004a; 2004b) have used a point process approach to construct an adaptive decoding filter wherein the intended movement (two-dimensional cursor movement on a video monitor) and the tuning of the individual neurons to movement were simultaneously estimated by the filter. This allowed the filter to learn and detect changes in the movements preferred by individual neurons thus making the system more resistant to changes in the response properties of the recorded neurons. They simulated a nonstationary population of 20 neurons from a set of physiological recordings in which the neurons ”died” and were subsequently replaced by new neurons at the rate of one per minute. The algorithm was trained for 20 minutes and allowed to estimate the tuning parameters of the neurons given the movement signal and the spiking activity of the neurons. After training the algorithm to obtain the tuning parameters, the algorithm

was used to reconstruct movement trajectory for 24 hours with a trial length of 10 seconds during which neurons dropped out of the population and were replaced at the rate of one every minute. The algorithm was successfully able to estimate the tuning parameters for movement direction for the novel neurons in the 20 neuron ensemble after 2 hours of decoding. Of the two point process filters, the receptive field parameter responsible for speed modulation decreased thus degrading the estimate for the speed over time.

This algorithm employed unsupervised feature extraction learning using two point process filters in lock step and computed the neuronal parameters as estimates in a novel approach. With a simulated nonstationary neuronal ensemble constructed from a population of 20 neurons, the algorithm was able to accurately estimate the movement direction but not the speed of movement – which may be a limiting factor of the algorithm.

Rotermund et al. (Rotermund D., et al. 2006) have described a supervised adaptive system using a Bayesian approach to combat abrupt changes in the sources of neural signals used to decode movement. An error signal encoding differences between actual and simulated movement signals (a horizontal figure of eight stimulus) was used as an external teacher to drive the adaptation. A simulated population of 64 cosine tuned neurons in motor cortex underwent abrupt complete replacement which drove the accuracy in the reconstruction low. After a period of several hours (~17), the reconstruction was able to adapt to the performance level observed before the change occurred. While Bayesian estimators have been shown to approximate an optimal solution (Wu W., et al. 2006), the Bayesian approach is computationally intensive since it

involves a high number of computations making it difficult to implement in a portable system.

Srinivasan and colleagues (Srinivasan L., et al. 2007) have developed a general purpose point–process lock-step adaptive filter based on Eden et al (2004) that refines the filter parameter estimates over each timestep to compensate for changes in the neuronal-electrode interface. A population of 25 neurons was simulated for the reconstruction of movement in an arm reaching task. As with the Eden algorithm, neuron parameters were estimated along with the movement using an unsupervised feature extraction learning algorithm. When the population lost one neuron per minute for 10 minutes for a total of 10 neurons, the adaptive system was able to reliably decode the velocity of movement with 10% error in the estimate.

With a loss of neurons, one might suspect that a loss in accuracy would result (as was the case with Eden et al) since less information is available to the decoder (with decoding error increasing in a $1/N^2$ fashion, N = number of available neurons). Their framework calls for a parallel processing architecture to be realized as a real-time decoding solution. With portable approaches, this may not always be possible.

Wu et al (Wu W., et al. 2008) describe an adaptive decoding filter approach that reoptimizes the decoding weights in a fashion similar to Gage et al. A recursive adaptive approach to the reoptimization was used to improve efficiency and performance was evaluated using data recorded from monkeys. Adaptive linear filter and adaptive Kalman filter implementations were compared with their respective non-adaptive counterparts in terms of efficiency and accuracy. The adaptive Kalman filter was found to be most efficient and accurate in decoding the neuronal ensemble.

For their experiments, Wu et al decoded two samples of electrical recordings offline - 33 and 45 minutes in length respectively. The sparse nature of the data influenced the length of the training stages. The speed of recovery was dependent on when the nonstationary changes (variation in firing rate of about 50% of the population) in the signal moved beyond the re-optimization window (between 350 and 500 seconds). Similar to the Gage algorithm, the adaptive Kalman filter requires explicit error information in the same dimensions as the movement parameters being estimated. Within the comparatively small time scale described, average neuron firing rates of about 50 % of the population varied over time with consistent hand positions thus exhibiting a nonstationary effect within the population. Loss of neurons may increase the size of the re-optimizing window because of the increase in sparsity of the data. While rate of adaptive decoding was better due to the iterative nature of the reoptimization (as opposed to the approach by Gage), the error levels described for the adaptive and non-adaptive Kalman filters were 30% MSE and 35% MSE respectively which is higher than comparable approaches (Srinivasan L., et al. 2007, 5 % MSE error in velocity (m/s)).

2.7 Summary

Accurate control of a neuromotor prosthetic system requires the development of adaptive decoding algorithms that quickly adapt to changes in properties of the recorded neural signals and are able to be implemented in a portable system. While other studies have looked at simultaneous loss and replacement of neurons, the proposed adaptive system performance was also tested in the presence of neuron loss, adaptation and attention modulation of neuron responses in addition to neuron replacement.

The primary aim of the system proposed here is to develop an adaptive decoding algorithm capable of compensating for the full range of nonstationary changes in the neural signals recorded at the brain-machine interface. The adaptive decoding algorithm proposed here is constrained computationally with the goal of ultimately implementing the algorithm in real-time in a low power, minimally computationally constrained microprocessor based environment. Finally, the algorithm is designed to facilitate system recovery from catastrophic changes in the properties of the neural signals of a time frame of seconds (as opposed to current algorithms that take minutes to hours (Rotermund et al 2006)).

3 NEURON MODEL AND ADAPTIVE FILTER DESIGN

An adaptive decoding algorithm was developed utilizing a Kalman filter framework to continuously optimize the internal state of the decoding algorithm in response to changes in the statistical properties of neural signals. In order to evaluate the resistance of the algorithm to the changes in neural signals typically encountered in a neuroprosthetic system, a population of spiking neurons was simulated with four different neuro-physiological effects (loss and replacement of neurons, attention modulation and adaptation) that together contribute to the recording of nonstationary neural signals at the brain-machine interface. At each time step, simulated neural signals were input to the Kalman filter to provide an estimate of the desired movement. The error between the decoded and desired movement were in turn used to update the Kalman filter weights to minimize movement error.

The population of spiking neurons in primary motor cortex was simulated to evaluate the impact of external and physiologic nonstationary changes in the recorded neural signals on the performance of three decoding algorithms – the proposed adaptive Kalman filter, a reoptimizing linear filter and a reoptimizing Kalman filter (based on Wu et al 2008). The performances of these adaptive decoding algorithms were compared to a non-adaptive static Kalman filter to illustrate the impact of the nonstationary phenomenon on decoding. Decoding performance was also compared to an optimal decoding error corresponding to the best case non-adaptive Kalman filter decoding for a given condition. The movement was setup as a bandlimited white noise signal in two-dimensions.

3.1 Neuronal Model

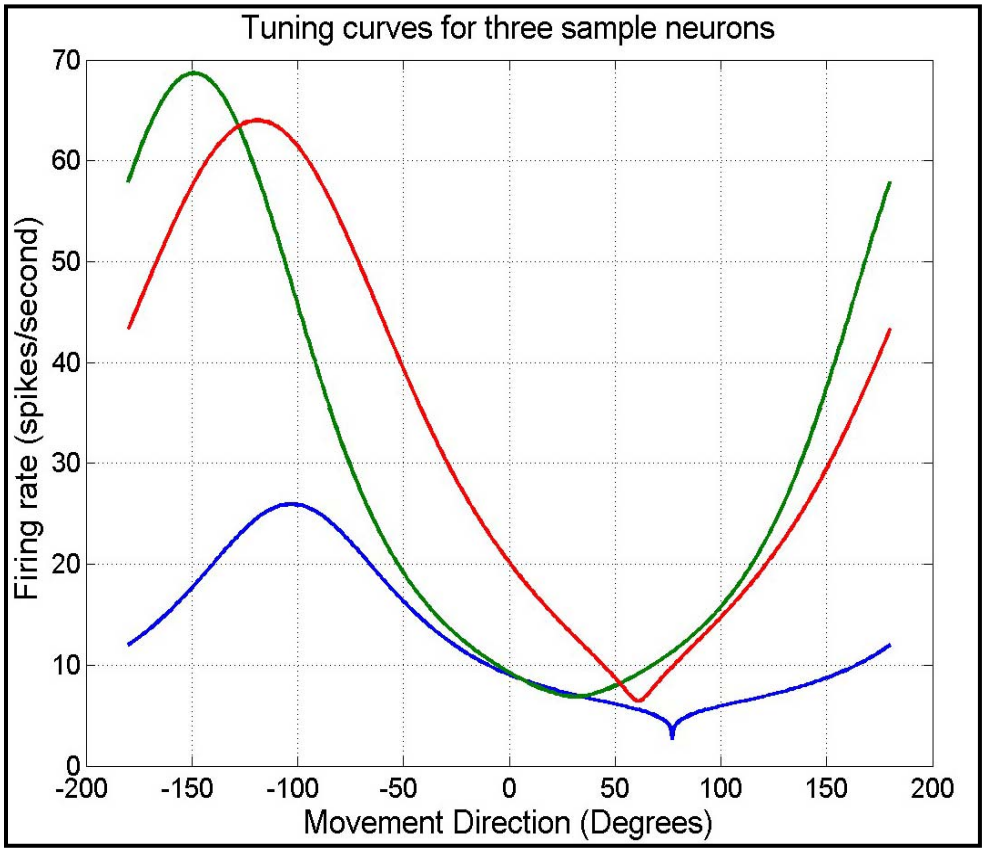


Figure 3.1: Direction tuning curves for three example neurons in primary motor cortex with varying tuning width and response rate. Neurons were *von Mises* tuned for direction in the 2D task space with responses determined by the difference between the intended movement direction and each neuron's preferred direction (corresponding to the peak of each neuron's tuning response).

To evaluate the algorithm, we constructed a population of 100 leaky-integrate-and-fire (LIF) neurons in MATLAB© (R2008a) whose spiking responses to movement where modeled on neurons in motor cortex (Figure 3.1) (Amirikian et al. 2000; Swindale 1998; Moran et al. 1999). In the simulations, neuron responses increased linearly with the amplitude of movement and were tuned to movement direction using a von Mises function (Amirikian et al. 2000; Swindale 1998) of the form,

$$\boldsymbol{f(\theta) = b + ke^{\kappa \cos(\theta - \mu)}}, \quad (3.1)$$

where μ is the neuron's preferred direction of movement, θ is the intended movement direction, and κ is related to the tuning half-width at half maximum ($\theta_{1/2}$) by the expression,

$$\boldsymbol{\theta_{1/2} = \cos^{-1}\left[\frac{\left(\ln\left(e^{2\kappa}+1\right) - \ln 2 - \kappa\right)}{\kappa}\right]}, \quad (3.2)$$

Preferred directions (μ) were uniformly distributed across the population from 0° to 360° and $\theta_{1/2}$ was selected from a range of 30° to 89° (Amirikian et al. 2000) for each neuron. Neuron responses (spikes/sec) were computed over 50 ms intervals (bins), commonly used in neural electrode recordings (Moran et al. 1999). The maximum response (k) of each neuron was drawn from a uniform distribution ranging from 10 to 40 spikes/sec (Moran et al. 1999) at a speed of 1, and the background firing rate (b) and encoding error were set to 10% of the neuron's maximum response. The neuron responses were linearly tuned to speed such that maximum responses between 20 to 80 spikes/sec were observed for a speed of 2. For the simulated neural populations, approximately 40% of neurons responded above background over each 50 ms interval.

The leaky integrate and fire (LIF) neuron model (as shown in Figure 3.2) approximates the nonlinear spiking behavior of a physiological neuron using a Resistive – Capacitive (RC) circuit that integrates the somatic current to a preset voltage threshold voltage Vth (sub-threshold phase) and generates an action potential (spike) after the somatic voltage crosses the threshold (super-threshold phase). After the formation of the spike, the model resets for a time period τref (absolute refractory period) before integrating the somatic current again.

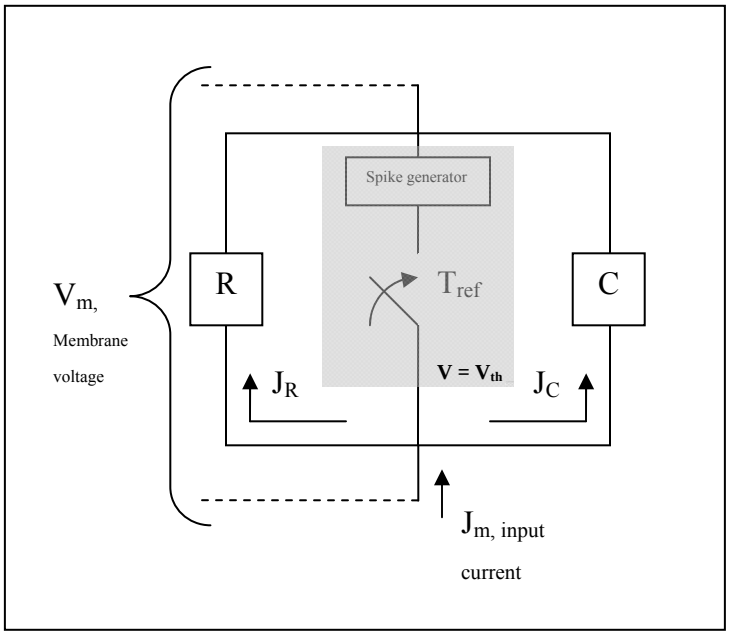


Figure 3.2: Simulated Leaky Integrate and Fire Neuron. A resistive-capacitive circuit simulating a LIF neuron model (based on Eliasmith et al. 2002). Gray area indicates the super threshold behavior used to generate the action potentials (spikes).

The membrane current $\boldsymbol{J_M(\dot{x}) = J_d(\dot{x}) + J_{bias}}$ is used to drive the somatic voltage above threshold to generate an action potential, after which the somatic voltage resets to zero. The membrane current incorporates an input driving current ($\boldsymbol{J_d}$) that simulates the dendritic input to the soma which is a function of the input stimulus, ***x,*** such that,

$$\boldsymbol{J_d(\dot{x}_{intended}) = \alpha g(\dot{x}_{intended})}, \tag{3.3}$$

where $\boldsymbol{\dot{x}_{intended}}$ defines the intended movement, $\boldsymbol{\alpha}$ is a parameter that defines the gain of the driving input, $\boldsymbol{g(\dot{x}_{intended})}$ is the encoding function (e.g. von Mises tuning function) and $\boldsymbol{\dot{x}_{intended}}$ corresponds to the magnitude of the movement variable being

encoded. The input bias current ($\boldsymbol{J_{bias}}$) defines the 'background' current due to neuron processes or constant current input from the nervous system.

The differential membrane voltage is given by the equation,

$$\frac{\boldsymbol{dV}}{\boldsymbol{dt}} = -\frac{\mathbf{1}}{\boldsymbol{\tau^{RC}}}(\boldsymbol{V} - \boldsymbol{J_M R}),$$

$$\boldsymbol{\tau^{RC}} = \boldsymbol{RC}$$

$\boldsymbol{\tau_{RC}} = \boldsymbol{R} * \boldsymbol{C}$ is the time constant of the resistive capacitive circuit responsible for the sub-threshold properties of the neuron, $\boldsymbol{R}$ represents the leakage resistance across the cell membrane due to the presence of ion channels $\boldsymbol{C}$ represents the dielectric nature of the membrane that separates the ionic charges across it.

Once the membrane voltage exceeds the threshold voltage ($V \geq V_{th}$), an action potential (spike) is generated. Thus, the membrane voltage V for a steady-state input is given as,

$$\boldsymbol{V(t) = J_M R\left(1 - e^{-t/T^{RC}}\right)}$$

Under steady-state conditions, the firing rate is then given by,

$$\boldsymbol{a(t_{th}) = \frac{1}{t_{th} + \tau^{ref}}}$$

τ_{ref} is the absolute refractory time of the neuron which defines the period after the occurrence of a spike when the somatic voltage is shunted to its resting potential (zero).

In the simulations, the number of spikes within a 50ms time bin is counted to compute the firing rate for each neuron. The firing rate as a function of the encoded variable for a constant input can be approximated by the expression,

$$\boldsymbol{a(x)} = \frac{\mathbf{1}}{\boldsymbol{\tau_{ref} - \tau_{RC}} \ln\left(\mathbf{1} - \frac{J_{th}}{J_M(x)}\right)}, \tag{3.4}$$

where J_{th} is the threshold current given by $J_{th} = \frac{V_{th}}{R}$ that specifies the threshold boundary.

The simulations were constructed with a leakage resistance (R) of 1, voltage threshold (V_{th}) of 1, neuron refractory periods (τ_{ref}) between 2-5 ms and sub-threshold RC time constants (τ_{RC}) between 10-30 ms. Ranges for τ_{ref} and τ_{RC} are based on neurophysiological data from Moran et al. 1999). A value of 1 was chosen for the leakage resistance and the threshold voltage for convenience.

A *von Mises* tuning function allows for variable (especially narrower) tuning widths, which closely approximate to the observed profiles of motor cortical cells. Amirikian and Georgopoulos (2000) show that the commonly employed cosine tuning function (Georgopoulos et al. 1982), which has a fixed tuning width = 90°, is not the most appropriate model for a majority of motor cortex cells. The *von Mises* tuning function is a circular function that approximates a normal distribution over angle and permits different tuning widths among a population of neurons.

Nonstationarities (undesirable processes that impact the statistical properties of the neural data) were induced into the neuronal population to simulate chronic implant effects. The processes were designed to modify the mean and variance of the tuning

properties of the simulated neuron population thus influencing decoding performance. The impact of removing neural signals, recruiting new neurons, neural adaptation, and attention were simulated to evaluate the performance of the adaptive algorithm. Table 3.1 shows the nonstationary conditions that were simulated along with their effects:

Simulated changes in the neural representation over time	**Physical effect**
Loss of neurons	Encapsulation of the electrode as an immunological response
Simultaneous loss and recruitment of neurons	Movement of electrode
Increase / decrease in maximum neuronal responses	Modulation by attention
Changes in the tuning properties of neurons	Modulation by adaptation

Table 3.1 Modeling nonstationary sources in the simulation. Four nonstationary processes were simulated to model undesirable changes at the neuron-electrode interface. Twenty simulations for each 'nonstationary' condition were created and the undesirable effects as well as system recovery were characterized.

3.2 Compensating for the Effects of Nonstationary Signals at the Neuronal-Electrode Interface

In the system developed here, we applied a supervised learning approach within the context of Kalman filter architecture to continuously adapt to nonstationary changes in the neural signals recorded at the brain-machine interface. The algorithm was designed to facilitate system recovery from catastrophic changes in the neural interface within minutes while minimizing the computational requirements of the system.

3.2.1 Design Specifications

The adaptive decoding system was developed to meet several design criteria –

- Accuracy
- Time to recovery
- Computational cost
- Real-time performance

3.2.1.a. Accuracy

The algorithm is required to produce accurate estimates of the stimulus properties encoded by the neurons (i.e., velocity) as quantified using a Normalized Root Mean Square Error (NRMSE) measure. The adaptive algorithm should achieve accuracy levels that are comparable with current decoding algorithms (0.1 – 0.2 NRMSE) for a stationary population of 100 neurons.

3.2.1.b. Time to Recovery

The performance of the proposed adaptive decoding system (which is based on a Kalman filter formulation) was compared to that of an optimal Kalman decoding system, given by movement decoded using optimal coefficients for the altered neuron population. Time to recovery for the adaptive filter was defined as the time taken for its decoding performance to achieve an accuracy level that approaches within 20% of the optimal decoding, after the appearance of a nonstationary condition. In our simulations,

catastrophic nonstationarities were used to replicate worst-case scenarios. The algorithm was designed to recover to the desired accuracy within minutes after a nonstationary occurs.

3.2.1.c. Computational Cost

The number of computations for an algorithm is the count of mathematical operations that the algorithm performs in a single iteration. It negates the effects of hardware and allows for direct comparisons of performance between algorithms. It also provides a means of estimating the hardware requirements for implementation of an algorithm in the face of additional constraints. The number of computations is desired to be less or equivalent to currently available adaptive schemes described in the literature. The reoptimizing Kalman algorithm described by Wu et al 2008, for example, requires the use of a number of discrete random variables with numerous possible values at each timestep resulting in a computational cost given by O (N^3), where N is the size of the matrices (number of simulated neurons) and O denotes order of the operation. Typically, the big-O notation describes the order of the largest term in the number of steps required for computation. Within this document, the big-O notation illustrates the order of matrix multiplications that dominate the computations. A lower computational cost would make the algorithm amenable to a portable implementation in a microprocessor based prosthetic control system.

3.2.1.d. Real-time Performance

The algorithm was designed to decode movement variables (such as velocity) from neural signals (simulated or obtained from the motor cortex). In the rate based decoding scheme, neuron responses were obtained as firing rates (spikes/sec) over 20-50 ms temporal intervals. The algorithm should be able to decode neural signals in real time – i.e. within the bin width (50 ms) used for rate-based decoding. A real-time decoding algorithm would allow the prosthetic control system to provide control signals to the prosthesis within the movement duration and enable smooth movement.

3.2.2 Implementation

The adaptive decoding algorithm developed to achieve these design criteria is composed of two parts – a Kalman decoding filter and a corrective filter (Figure 3.3).

3.2.2.a. Kalman Decoding Filter

A Kalman filter was used to decode intended movement based on the firing rates (spikes/s) obtained from the sampled neural responses. As a linear control system, the Kalman filter has been well studied in the literature, and is widely used in cases where accurate estimation of the internal system properties is required from noisy measurements (Maybeck 1979 – Chapter 1, Welch and Bishop – SIGGRAPH 2001). Moreover, the Kalman filtering approach is computationally less intensive than other control strategies and has a standard implementation, making it ideal for decoding neural signals at the brain-machine interface.

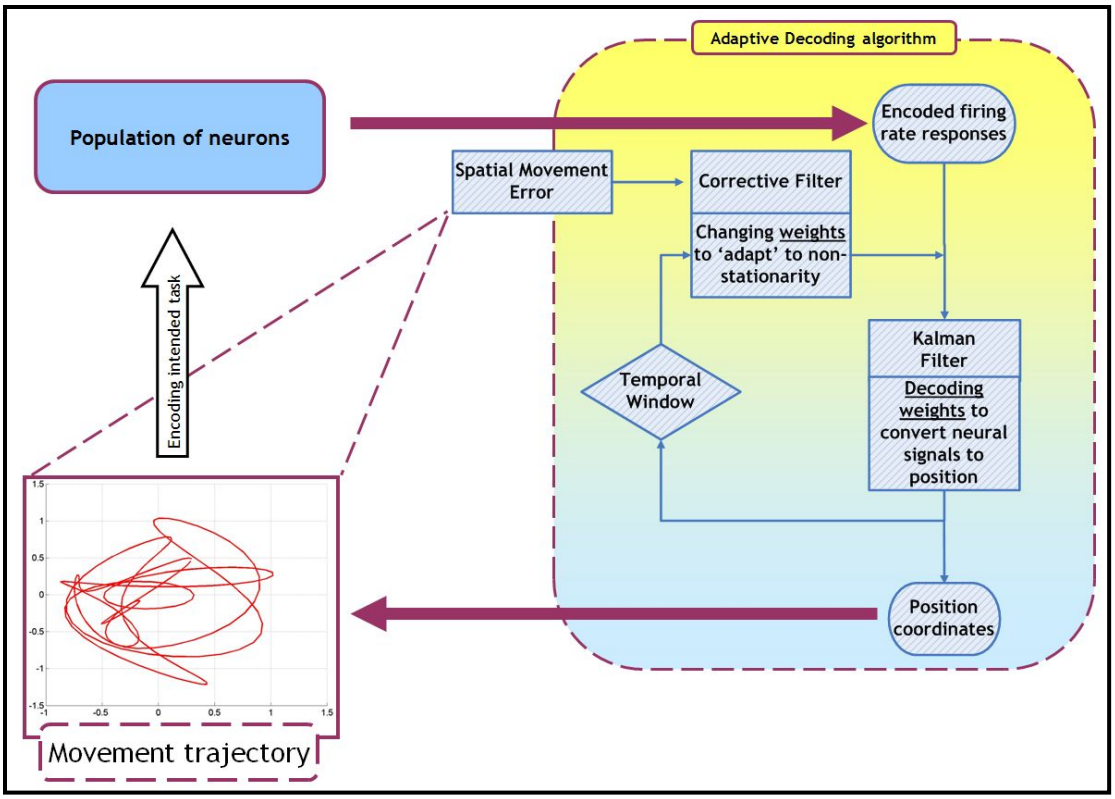


Figure 3.3: Block Diagram of the Adaptive Filter system. The adaptive filter system consists of the adaptive algorithm and the feedback of movement error. This error was used by the adaptive decoding algorithm to dynamically adjust the Kalman decoding weights for each neuron. A Kalman decoding filter was then used to estimate the movement parameters from the motor neuron response input.

In the Kalman filter implementation described here, the neural responses formed the measurement/observation matrix $\boldsymbol{z}$ while the state variable $\dot{\boldsymbol{x}}_{\boldsymbol{intended}}$ represented the velocity of the intended movement [$\boldsymbol{v}_x$; $\boldsymbol{v}_y$],

$$\boldsymbol{z}_i = \boldsymbol{H}^* \dot{\boldsymbol{x}}_{\boldsymbol{intended_i}} + \boldsymbol{b}_i, \qquad \boldsymbol{b} \subseteq \boldsymbol{N(0,R)} \tag{3.5}$$

$$\dot{\boldsymbol{x}}_{\boldsymbol{intended_i}} = \boldsymbol{A}^* \dot{\boldsymbol{x}}_{\boldsymbol{intended_{i-1}}} + \boldsymbol{w}_i, \qquad \boldsymbol{w} \subseteq \boldsymbol{N(0,Q)} \tag{3.6}$$

where, for a population of N neurons encoding a p-dimensional stimulus space, **A** is the state transition matrix (p x p), that relates the current iteration of the intended

movement velocity vector **x**, to the preceding velocity estimate, and **w** is Gaussian noise sampled from a normal distribution N (0, **Q**), where **Q** is the process noise covariance (p x p) estimated during the least squares optimization (Wu et al 2002). **H** is the measurement matrix (N x p), that defines the relationship between the neural responses **z**, and the estimated movement velocities $\dot{\mathbf{x}}$, for the current time-step, and **b** is Gaussian noise sampled from a normal distribution N (0, **R**), where **R** is the measurement noise covariance (N X N) estimated during the least squares optimization.

The Kalman filter employs a two-step prediction-correction computation for estimating its state variables. In eq. (3.6), a prediction for the state variable, i.e. movement velocity, at the i^{th} timestep is generated based on the velocity from the previous (i-1) timestep. This estimate is then corrected for by using the following relationship,

$$\dot{\boldsymbol{x}}_{\boldsymbol{intended_i}} = \dot{\boldsymbol{x}}_{\boldsymbol{intended_{i-1}}} + \boldsymbol{K}_{\dot{\boldsymbol{x}}}*(z_i - \boldsymbol{H}*\dot{\boldsymbol{x}}_{\boldsymbol{intended_{i-1}}}), \tag{3.7}$$

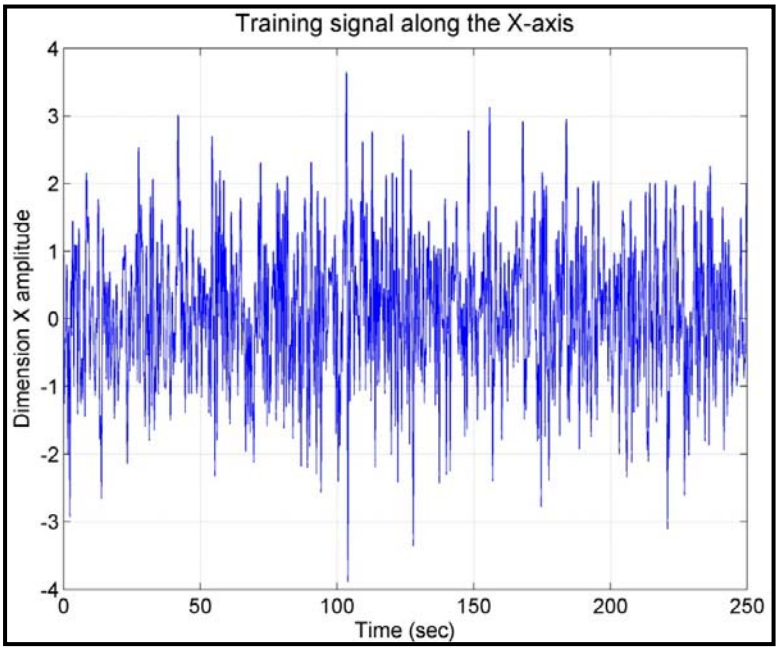
where $\boldsymbol{K}_{\dot{\boldsymbol{x}}}$ is the Kalman filter gain that serves to bring the error in the neuron responses to the domain of the state variables (movement velocities). To facilitate decoding, the Kalman filter weights (***A***, ***H***, ***Q and R***) were optimized (as described below in Equation 3.8) using the firing rates from the population of neurons and the movement amplitudes.

In the general Kalman formulation, (***A***, ***H***, ***Q and R)*** may be time varying. Here, for simplicity and so that they can be estimated during the optimization phase, the state, measurement, and noise matrices were considered to be constant for the static Kalman

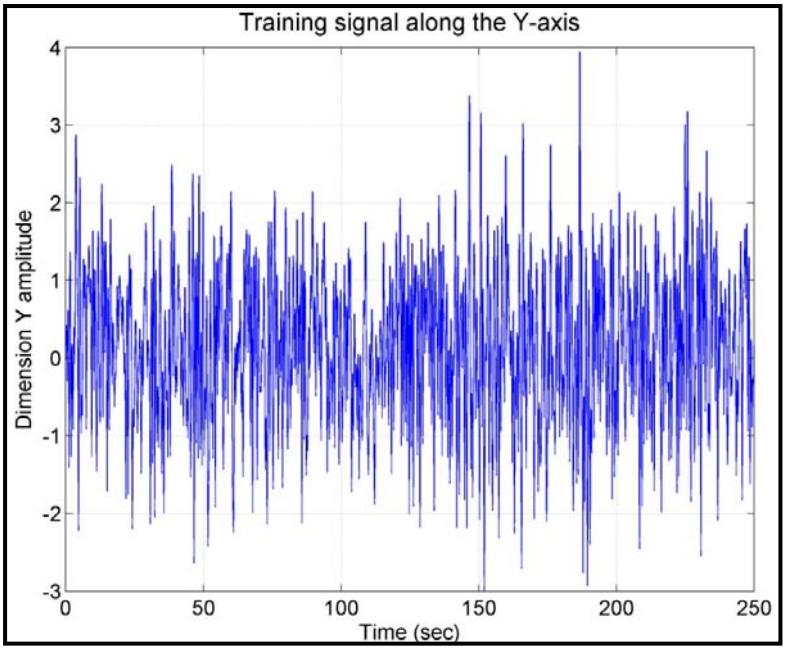
filter. During an initial optimization phase, the Kalman coefficients (***A***, ***H***, ***Q*** *and* ***R***) were optimized for each neuron using a least squares error minimization algorithm as described in Wu et al (2002). During this optimization process, the relationship that minimizes the decoding error between the firing rates of the neurons and the movement variable over the entire length of the training signal was established as described below in equation 3.8.

A 250 second long band-limited white noise training signal sampled at 1 ms (Figure 3.4), was used to optimize the Kalman filter weights. The white noise movement with frequencies within the (0 – 1.5 Hz) range was chosen to span the movement space to approximate the motion of a prosthetic system. Since the decoding weights were determined so as to obtain the minimum decoding error over the entire length of the training signal, choosing a signal that sufficiently samples the space was necessary to ensure accurate decoded movements throughout the space.

(A)



(B)



(C)

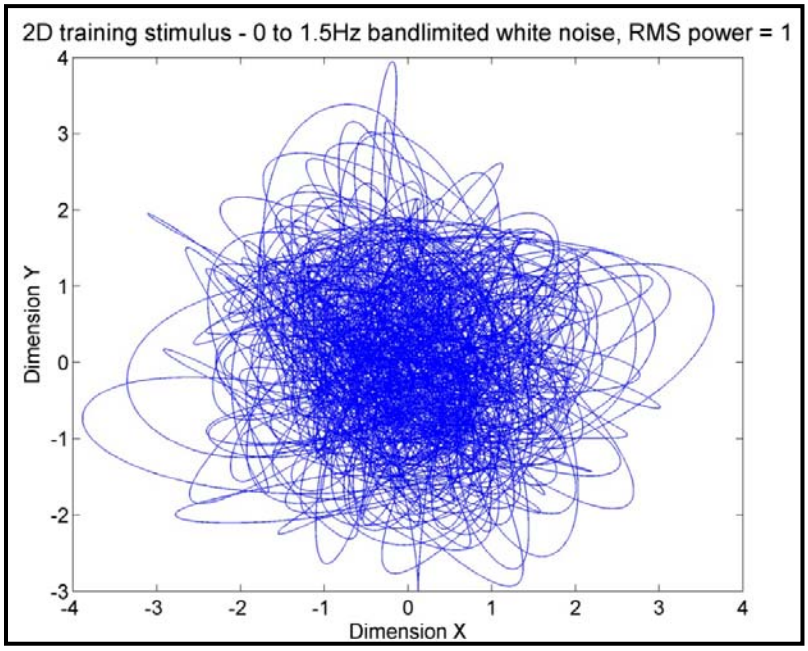


Figure 3.4: (A) X-component and (B) Y-component of the (C) two-dimensional training signal used to optimize the adaptive filter system. A two dimensional 0 – 1.5 Hz bandlimited white noise signal was used to optimize the decoding weights of the Kalman filter using a least squares minimization process. The RMS power content of the training signal was set to 1 for convenience.

A simulated 100 neuron population was set up using the *von Mises* neuronal model described in section 3.1 with maximum response rates between 20 – 80 spikes/sec, with preferred directions uniformly distributed from 0° to 360°.

The firing rates from this 100-neuron population to the white noise training stimulus were computed over 50 ms bins and were fed to the least squares minimization algorithm along with the actual movement amplitudes (in two dimensions), computed as the average over each 50ms interval for each dimension. The optimization was performed using the matrix equations detailed in eq. (3.8), which correspond to the least squares minimization of the Kalman filter prediction-correction equations listed in eq. (3.5 and 3.6).

Signal length N = 250 sec / 50 ms = 5000 bins

$$\mathrm{A} = \begin{pmatrix} x_{1,2} & \cdots & x_{1,N} \\ x_{2,2} & \cdots & x_{2,N} \end{pmatrix} \bullet \begin{pmatrix} x_{1,1} & \cdots & x_{1,N-1} \\ x_{2,1} & \cdots & x_{2,N-1} \end{pmatrix}^{T} \bullet inv\left(\begin{pmatrix} x_{1,1} & \cdots & x_{1,N-1} \\ x_{2,1} & \cdots & x_{2,N-1} \end{pmatrix} \bullet \begin{pmatrix} x_{1,1} & \cdots & x_{1,N-1} \\ x_{2,1} & \cdots & x_{2,N-1} \end{pmatrix}^{T} \right)$$

$$\mathrm{H} = \begin{pmatrix} z_{1,1} & \cdots & z_{1,N} \\ \vdots & \ddots & \vdots \\ z_{100,1} & \cdots & z_{100.N} \end{pmatrix} \bullet \begin{pmatrix} x_{1,1} & \cdots & x_{1,N} \\ x_{2,1} & \cdots & x_{2,N} \end{pmatrix}^{T} \bullet inv\left(\begin{pmatrix} x_{1,1} & \cdots & x_{1,N} \\ x_{2,1} & \cdots & x_{2,N} \end{pmatrix} \bullet \begin{pmatrix} x_{1,1} & \cdots & x_{1,N} \\ x_{2,1} & \cdots & x_{2,N} \end{pmatrix}^{T} \right)$$

$$\mathrm{Q} = \frac{\left(\left(\begin{pmatrix} x_{1,2} & \cdots & x_{1,N} \\ x_{2,2} & \cdots & x_{2,N} \end{pmatrix} - A \bullet \begin{pmatrix} x_{1,1} & \cdots & x_{1,N-1} \\ x_{2,1} & \cdots & x_{2,N-1} \end{pmatrix} \right) \bullet \left(\begin{pmatrix} x_{1,2} & \cdots & x_{1,N} \\ x_{2,2} & \cdots & x_{2,N} \end{pmatrix} - A \bullet \begin{pmatrix} x_{1,1} & \cdots & x_{1,N-1} \\ x_{2,1} & \cdots & x_{2,N-1} \end{pmatrix} \right) \right)^{T}}{(N-1)}$$

$$R = \frac{\left(\begin{pmatrix} Z_{1,1} & \cdots & Z_{1,N} \\ \vdots & \ddots & \vdots \\ Z_{100,1} & \cdots & Z_{100.N} \end{pmatrix} - H \bullet \begin{pmatrix} X_{1,1} & \cdots & X_{1,N} \\ X_{2,1} & \cdots & X_{2,N} \end{pmatrix}\right) \bullet \left(\begin{pmatrix} Z_{1,1} & \cdots & Z_{1,N} \\ \vdots & \ddots & \vdots \\ Z_{100,1} & \cdots & Z_{100.N} \end{pmatrix} - H \bullet \begin{pmatrix} X_{1,1} & \cdots & X_{1,N} \\ X_{2,1} & \cdots & X_{2,N} \end{pmatrix}\right)^T}{N}$$

… (3.8)

In the current implementation, the state transition matrix **A** was given by the identity matrix since the state variables $\dot{\boldsymbol{x}}_{\boldsymbol{intended}} = [v_1, v_2]$ modeling movement velocities were independent. (When the matrix **A** was computationally determined during the training process, it approached a 2x2 Identity matrix.) More generally, however, the state transition matrix defines the relationships between internal states spanning multiple dimensions of movement, e.g., position and velocity, and can contain off-diagonal terms.

3.2.2.b. Corrective Filter

When the statistical properties of the signals at the neuronal-electrode interface change, the Kalman filter weights $\boldsymbol{H}$ are no longer optimized for the sampled neuronal population resulting in improper estimates of intended movement. Therefore, the weights themselves must be adjusted to compensate for the alterations in the neural responses, thereby re-optimizing the decoded movement. The corrective filter achieves this through the use of an external error source (corresponding to a movement error in the same dimensions as the decoded variables) to drive the weight correction. In a physically realized system, the error signal may be derived from external systems that utilize visual and/or sonic modalities to spatially localize the prosthesis or on error signals decoded directly from the brain.

From equation (3.7), changes in the neural responses, $\boldsymbol{z}$, must be compensated for by error-driven changes in the measurement matrix, $\boldsymbol{H}$, to optimize the mapping between intended movement $\dot{\boldsymbol{x}}_{\boldsymbol{intended}}$, and neural responses $\boldsymbol{z}$. Using the current predicted value of the movement velocities $\dot{\boldsymbol{x}}_{\boldsymbol{intended}}$ and the error signal $\dot{\boldsymbol{x}}_e$, $\boldsymbol{H}$ can be iteratively adjusted such that,

$$\boldsymbol{H}_i = \boldsymbol{H}_{i-1} + \eta * \boldsymbol{K}_H * (\boldsymbol{z}_i - \boldsymbol{H}_{i-1} * \dot{\boldsymbol{x}}_{act}) \quad ,$$

$$\dot{\boldsymbol{x}}_{act} = (\dot{\boldsymbol{x}}_{\boldsymbol{intended}} + \dot{\boldsymbol{x}}_e) \, , \qquad \ldots (3.9)$$

where $\boldsymbol{H}_i$ is the corrected weight for the current timestep, $\boldsymbol{H}_{i-1}$ is the erroneous weight from the previous timestep, $\eta = 0.2$ is a scaling factor determined empirically that is applied to enable lower weight changes over each iteration and bound the filter weights. While a faster value of η would help in improving the speed of recovery, a smaller value could improve decoding accuracy. In order to select the value of η, a gradient descent algorithm was implemented that modified the scale by ±5% over a 2.5 second non-overlapping window so that error within this window was reduced. The initial value of η was set to a high value (20). It was observed that the value approached a 0.2 asymptote for these simulations. The scaling factor η does not apply to other adaptive decoding filters implemented here (reoptimizing Kalman and reoptimizing linear filters described in Wu et al 2008) since these filters perform a complete reoptimization of their decoding weights over their 550 second reoptimizing window.

$\boldsymbol{K}_H$ is an adaptive mapping of the error between the predicted versus actual neural responses and therefore carries the dimensions of $\dot{\boldsymbol{x}}_{\boldsymbol{intended}}^{T}$, $\dot{\boldsymbol{x}}_e$ is the error signal given

by the signed difference between the intended ($\dot{\boldsymbol{x}}_{act}$) and predicted ($\dot{\boldsymbol{x}}_{i}$) movements. $\boldsymbol{K}_H$ is obtained over each timestep by computing the Kalman gain factor (*Welch and Bishop – SIGGRAPH 2001)* as follows –

$$\boldsymbol{K}_H = \mathbf{P}_{H,\,i-1} * \boldsymbol{H}_{i-1}^{T} * \text{inv}\,(\boldsymbol{H}_{i-1} * \mathbf{P}_{H,\,i-1} * \boldsymbol{H}_{i-1}^{T} + \boldsymbol{R}), \tag{3.10}$$

where $\mathbf{P}_{H,\,i-1}$ is a measure of the covariance of the estimate of **H** for each timestep given by,

$$\mathbf{P}_{H,\,i} = \mathbf{P}_{H,\,i-1} - \eta * \boldsymbol{K}_H * \dot{\boldsymbol{x}}_{act} * \mathbf{P}_{H,\,i-1}), \tag{3.11}$$

In Eq. 3.9, the term ($\dot{\boldsymbol{x}}_{\boldsymbol{intended}} + \dot{\boldsymbol{x}}_{e}$) represents the actual movement $\dot{\boldsymbol{x}}_{act.}$ This term acts as an external teacher and modulates the changes in $\boldsymbol{H}$ to iteratively minimize the difference term ($z_i - \boldsymbol{H}_{i-1} * \dot{\boldsymbol{x}}_{act}$), where z_i is the altered neural responses in the current time-step (see Section 2.3), which are compared with the internal estimate of the neural responses '$\boldsymbol{H}_{i-1} * \dot{\boldsymbol{x}}_{act}$' to drive iterative changes in $\boldsymbol{H}$.

The adaptive filter system was tested using stationary as well as nonstationary responses from simulated neuron populations. A 100-neuron population was simulated in most cases (unless otherwise specified). Each simulation initialized a new population of neurons so that algorithm performance could be evaluated independent of neuron bias.

A non-recursive reoptimizing linear filter and a non-recursive reoptimizing Kalman filter based on (Wu et al. 2008) were constructed with a reoptimizing window of 550 seconds. In addition, an optimal decoder that represented the optimal performance

for the altered neuron population was implemented by running the optimization on the neuron population after the introduction of the nonstationarity.

For each simulation, the decoded movement was obtained for the adaptive Kalman filter and compared against the movement obtained using the reoptimizing linear filter, the reoptimizing Kalman filter, the optimal decoder and a non-adaptive static Kalman filter. Four nonstationary conditions were simulated – loss of neurons, replacement of neurons, attention modulation and adaptation. The following chapters discuss each type of nonstationarity separately along with their implementation and results.

4 DECODING PERFORMANCE BEYOND THE TRAINED SPACE

As described in Chapter 3, an adaptive neural decoding system based on a Kalman filter design was implemented with an aim to provide accurate decoding in the presence of nonstationary neural signals. A simulated 100 neuron population was set up using the *von mises* neuronal model described in section 3.1 with maximum response rates between 10 to 40 spikes/sec at a speed of 1, with preferred directions uniformly distributed from 0° to 360° and no connectivity between the units. Background responses attributed to noise were limited to 10% of the spiking activity.

An initial training or optimization process was used to establish the decoding coefficients (weights) of this filter using a two dimensional 0 - 1.5 Hz bandlimited white noise signal with a RMS power of 1 (see Figure 3.4). In order to provide a reference for comparison of decoding performance, a 'static' or non-adaptive Kalman filter based decoder was also implemented with the same initial optimized decoding weights as the adaptive filter. The decoding performance of the adaptive filter was also compared with a re-optimizing linear filter and a re-optimizing Kalman filter as described by Wu et al (2008). These adaptive filters were optimized over an initial 550 second time window after which their decoding performance was tested.

Two types of bandlimited white noise movements (1450 seconds long each) were chosen to test decoding performance under stationary conditions (i.e. with no modification of the simulated neural population after optimization) as follows –

1. Bandlimited white noise movements with an RMS power of 1, but with differing frequency bands of 0 - 0.5Hz, 0 - 1Hz, 0 - 1.5Hz, 0 - 2Hz and 0 - 5Hz.
2. Bandlimited white noise movements within a frequency band of 0 - 1 Hz, but with differing RMS powers of 0.5, 1, 2 and 5.

The RMS power specifies the extent (range) of the motion of the prosthetic device, while the frequency range illustrates the speed of the motion. During the optimization process, the subject is asked to perform a series of typical movements while the filter learns the association of this movement to the neuron responses by assigning specific weights. This enables the filter to reliably decode similar movement using these weights when provided with the neuron responses. However, it may or may not be able to accurately decode beyond its trained (optimized) movement space.

The two sets of test signals in this case were designed to evaluate the performance of the different filters in decoding motion beyond the range of movements provided during their optimization phases. This would enable a decoding filter to independently adapt to the novel movements without requiring the subjects to return to the labs for significant re-training of the decoding filter.

For this experiment, a neuron population of 100 neurons was constructed as described earlier in Chapter 3. The training signal was a 250 second two-dimensional 0 - 1.5 Hz bandlimited white noise signal with a RMS power of 1. With this neuron population, five simulations each were run for a white noise test stimulus of 2000 seconds in length for each of the test signal conditions. Root Mean Square Errors

(RMSE) were computed over 10 second windows along the test stimulus length and normalized to the test amplitude.

4.1 Results

Figure 4.1 shows a comparison of normalized root mean square error (NRMSE) computed over the last 1450 seconds of the test stimulus for each of the four filters when the test movement bandwidth was varied in order to allow the reoptimizing filters to reach their optimal decoding state (after 550 seconds) before the advent of the nonstationarity. Within the range of optimized movements (0 - 1 Hz and an RMS power of 1), the decoding accuracy of the filters was good with the adaptive filter errors lowest (0.171 NRMSE). Low decoding errors for movements characterized from 0 - 0.5 Hz were obtained for all the decoding filters as shown in Table 4.1. Beyond the ranged of trained movements, i.e. 0 - 1 Hz, the non-adaptive 'static' filter error increased significantly ($t(4) = -39.75$, $p<0.00001$) with frequency range (bandwidth from 1.5-5 Hz). A similar trend was observed for the re-optimizing linear and re-optimizing Kalman decoding filters with all three filters approaching ~0.8 NRMSE for movements characterized from 0 - 5 Hz.

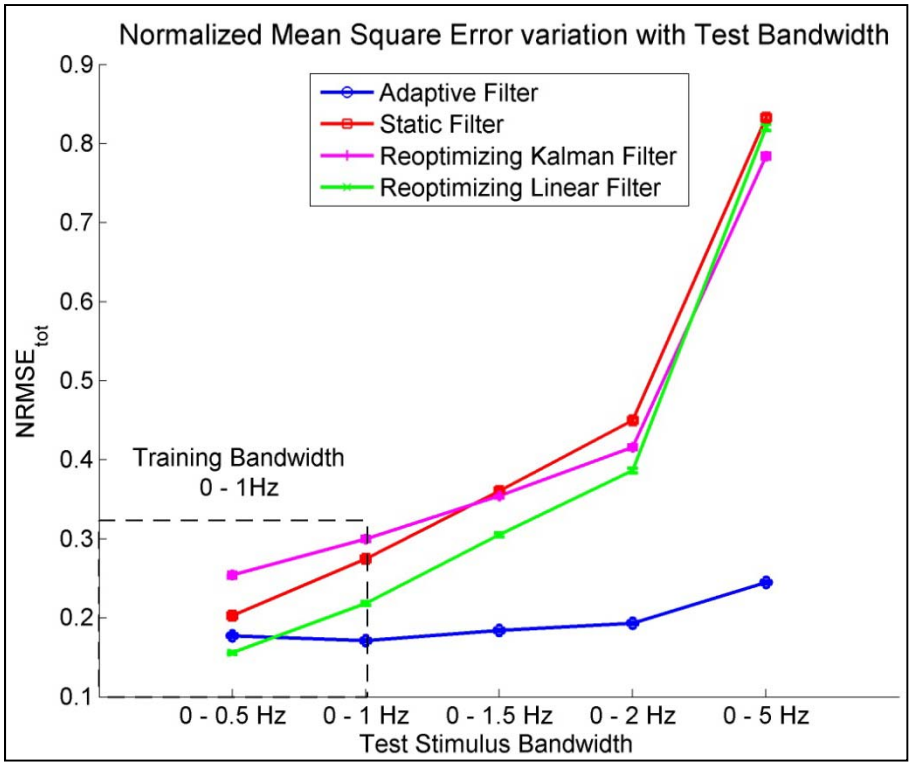


Figure 4.1: **Normalized root mean square error (NRMSE) in response to changing test stimulus bandwidth across five simulations.** NRMSE is shown for the static Kalman (red), adaptive Kalman (blue), reoptimizing Kalman (magenta) and reoptimizing linear filters (green). Errors were computed over a 10 second sliding window. Filter performance was initially optimized for pseudo-random movements with power from 0 - 1 Hz. Decoding performance for the adaptive Kalman filter remained the same beyond the trained frequencies, while decoding errors rose significantly for the static Kalman, reoptimizing Kalman and reoptimizing linear filters. Error bars correspond to ±2 Standard Errors and are within the symbols for all four plots

The decoding accuracy of the adaptive decoding filter did not deteriorate to the extent seen for other decoding filters. The best performance of the adaptive filter occurred for a stimulus bandwidth of 0 - 1 Hz (NRMSE = 0.171) while it's maximum decoding error for the 0 - 5 Hz stimulus (NRMSE = 0.245), is statistically different ($t(4) = -59.41$, $p<1E-6$) but still lower than the other adaptive decoding approaches. The decoding error for the adaptive filter was the lowest among the four filters for the frequency ranges tested (Figure 4.1).

Test Stimulus Bandwidth	Mean Normalized Root Mean Square Decoding Errors			
	Static filter	Re-optimizing Kalman filter	Re-optimizing Linear filter	Adaptive Filter
0 - 0.5 Hz	0.203±0.002	0.254±0.003	0.156±0.001	0.177±0.001
0 - 1 Hz	0.275±0.002	0.299±0.003	0.219±0.002	0.171±0.001
0 - 1.5 Hz	0.361±0.002	0.354±0.002	0.305±0.002	0.184±0.001
0 - 2 Hz	0.449±0.003	0.416±0.003	0.387±0.003	0.193±0.001
0 - 5 Hz	0.832±0.003	0.784±0.003	0.82±0.004	0.245±0.001

Table 4.1 Decoding errors for the static Kalman filter, re-optimizing Kalman filter, re-optimizing linear filter and the adaptive Kalman filter as test stimulus bandwidth is varied.

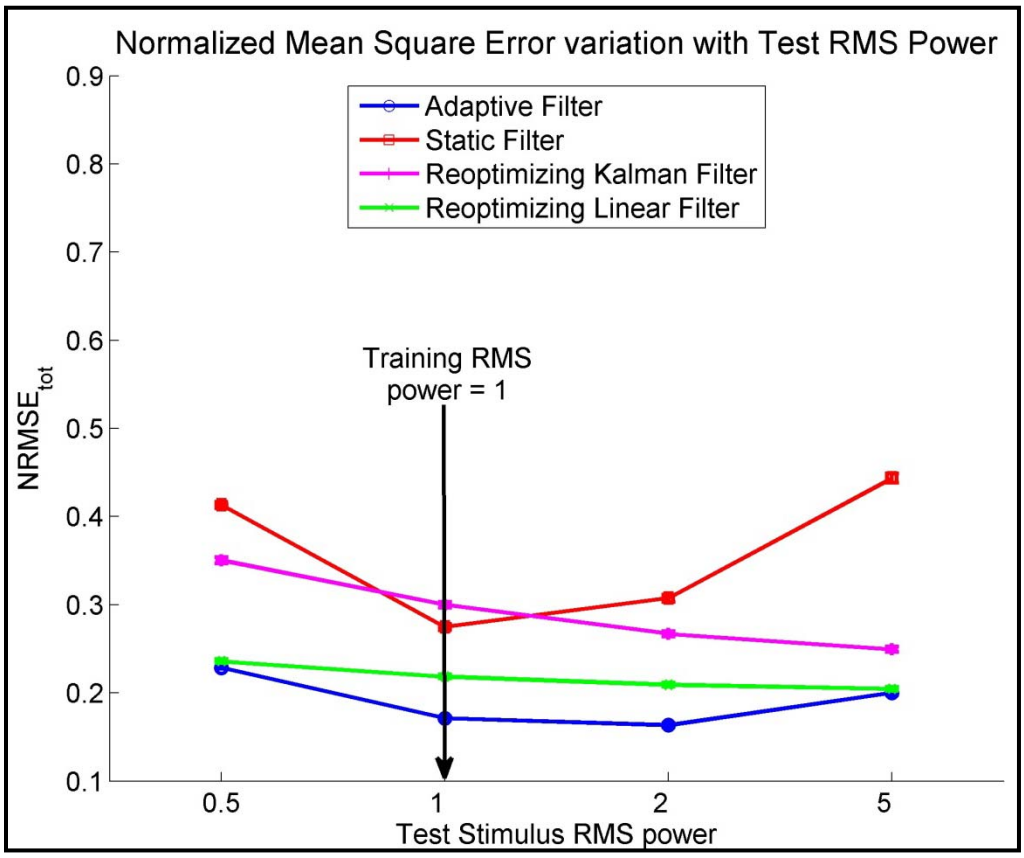


Figure 4.2: **Normalized root mean square error (NRMSE) in response to changing stimulus power.** NRMSE is shown for the static Kalman (red), adaptive Kalman (blue), reoptimizing Kalman (magenta) and reoptimizing linear filters (green) computed over the last 1450 seconds of the test stimulus across five simulations. Errors were computed over a 10 second window. Filter performance was initially optimized for a white noise signal (0 – 1.5 Hz) with an RMS power of 1. Decoding performance for the adaptive Kalman filter across stimulus amplitudes increased for untrained RMS amplitudes (0.5, 2, 5), and

the errors were significantly different (RMS = 0.5: t(4)=-53.15, p<1E-6; 2: t(4)=5.74, p<0.01; 5: t(4)=-16.79, p<1E-4) but still representing the best case decoding for these RMS amplitudes. Error bars correspond to ±2 Standard Errors and are within the symbols for all four plots.

As shown in Figure 4.2, decoding accuracy was also compared for the different decoding filters with varying RMS power for a movement bandwidth of 0 - 1 Hz. The decoding error for the static filter was at its lowest (NRMSE = 0.275) for the trained range of movement amplitudes (RMS = 1). At RMS powers of 0.5, 2 and 5 decoding error was significantly different (RMS = 0.5: t(4)=-79.73, p<1E-6; 2: t(4)=-11.71, p<0.001; 5: t(4)=-48.5, p<1E-5) and increased as shown in the table below.

Test Stimulus RMS power	Mean Normalized Root Mean Square Decoding Errors			
	Static filter	Re-optimizing Kalman filter	Re-optimizing Linear filter	Adaptive Filter
0.5	0.413±0.001	0.35±0.003	0.236±0.002	0.229±0.001
1	0.275±0.002	0.299±0.003	0.219±0.002	0.171±0.001
2	0.307±0.004	0.267±0.002	0.209±0.002	0.163±0.001
5	0.444±0.005	0.249±0.002	0.205±0.002	0.2±0.002

Table 4.2 Decoding errors for the static Kalman filter, re-optimizing Kalman filter, re-optimizing linear filter and the adaptive Kalman filter as test stimulus RMS power is varied.

4.2 Discussion

The non-adaptive static Kalman filter and the adaptive filter were provided with the same initial decoding weights associated with each neuron. However, the adaptive filter performance was better than the optimized Kalman filter even under the best case conditions that fall within the bounds of the training signal – bandwidth = 0 - 1 Hz, movement amplitude RMS power = 1. We attribute this improvement in performance to the continuous reoptimization of the adaptive filter to the instantaneous statistics of the

movement at each time-step. Since the initial optimization process is based on a least squares minimization approach, the decoding weights associated with each neuron aim to minimize movement error over the two-dimensional movement space (encompassed by the training signal shown in Figure 3.3). The adaptive nature of the dual-Kalman approach allows it to induce slight changes to the individual weights based on the movement stimulus at each 50 ms timestep. Therefore, we see greater decoding accuracy than the static filter within the space over which both algorithms were optimized.

For movements that fell outside the statistics of the training stimulus, the static Kalman filter errors were significantly higher than those for the adaptive decoding system. For movements beyond the trained bandwidth 0 - 1 Hz, the decoding accuracy of the re-optimizing linear and the re-optimizing Kalman filters suffered. The re-optimizing linear filter errors were 39.7%, 76.8% and 275% worse than its trained bandwidth error at bandwidths of 0-1.5 Hz, 0-2 Hz and 0-5 Hz respectively, while the errors for the re-optimzing Kalman filter were 18.2%, 38.7% and 161% worse for these bandwidths respectively. Because the reoptimizing filters rely on a least squares minimization technique over a long stimulus window (550 seconds) in order to update the decoding weights, they are not able to update the weights in response to instantaneous variations in the statistics of the movement stimulus. As a result, decoding errors increased with increasing bandwidth of the white noise test signal since they are unable to compensate due to their long adaptive windows.

This result may be significant for deciding the training parameters for a subject with a prosthetic implant. Typically, the range of amplitudes and bandwidth would be

chosen so as to encompass the entire range of movements that would likely be encountered with the device. This may entail longer training times, larger datasets and result in more generalized performance for which errors within specific regions of the movement space are not fully minimized. With the adaptive decoding system, a limited training signal can be used to initialize the system and subsequently adapt to instantaneous changes in the mapping between the input and output spaces while remaining within the range of weights necessary for global optimization.

5 LOSS OF NEURONAL SIGNALS

In invasive neuroprosthetic systems, neural signals are collected from the cortex using an electrode array implant. The aim of the electrode array is to be implanted without significant degradation in the quality of the neuronal signals recorded over the long term. However, a number of phenomena can occur that contribute to neuron signal loss (Lebedev et al. 2006), (Schwartz et al. 2006). Trauma to the neural cells during the implant procedure, resulting from the shape and size of electrode implant, type of insulation material used and depth of insertion, may damage the surrounding neural tissue (Bjornsson et al. 2006), (Polikov et al. 2005).

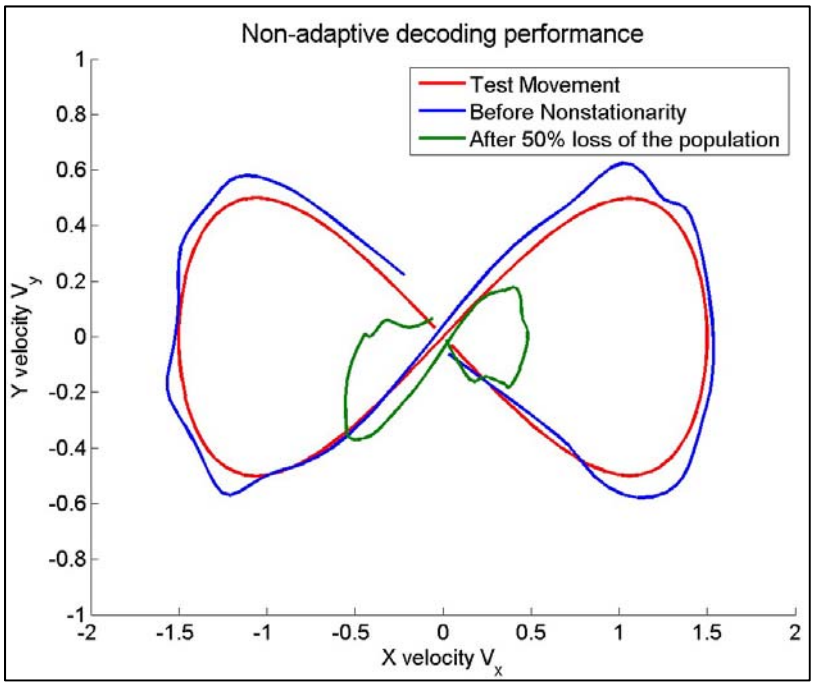
The region surrounding a chronically implanted electrode may see a rise in activated microglia clusters (1 – 3 weeks post implantation) and macrophages that engulf parts of the electrode during Phagocytosis. Glial scar formation is extremely common and results in the electrode being surrounded by glial tissue that may increase the distance between the electrode and the neuron population (Polikov et al. 2005). Both glial scar formation and astrocyte growth (up to 6 – 8 weeks post implantation) may result in displacement of neural tissue thus contributing to an increase in impedance and consequently loss of neural signal recordings. In a study of neural tissue response to chronically implanted electrode arrays, Biran et al. (2005) reported a 40% loss of neurons surrounding the tissue implant within a period of two weeks post implantation.

To examine the impact of such signal loss on decoding performance, we simulated an abrupt loss of 50% of the neuron population (as illustrated in Figure 5.1). A 100-neuron population was simulated with the same properties as described in Chapter 4.

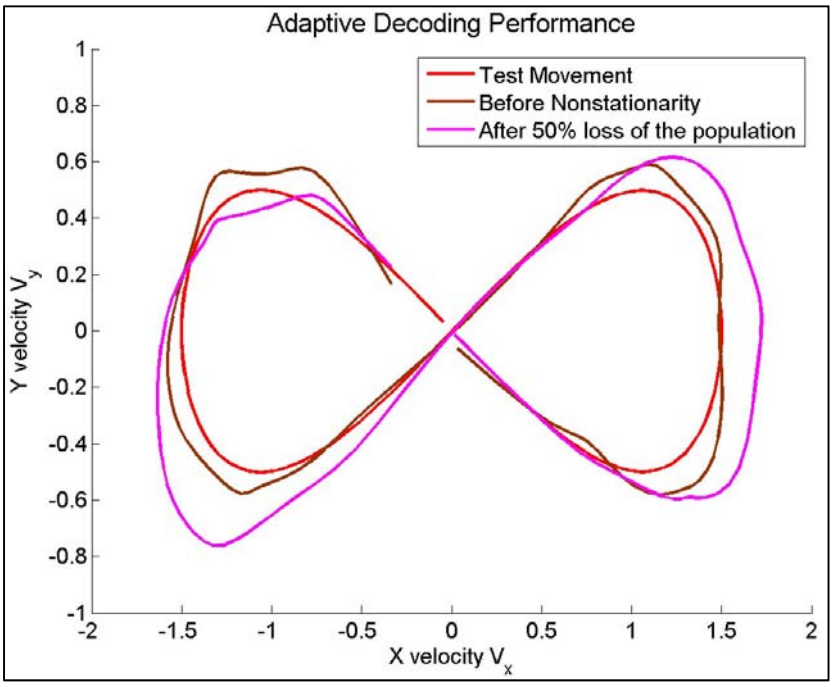
Six hundred and fifty seconds into the simulation, 50 neurons were removed instantaneously from the population, simulating a worst-case neuron loss. The loss was simulated by zeroing out the responses of 50 neurons for all times (t >100 sec.):

$$Rate_i = \begin{cases} 0, & i \in Lost\ population \\ R_i, & i \in Unchanged\ population \end{cases} \quad t > 100s \qquad \ldots (5.1)$$

where R_i is the instantaneous rate response of the i^{th} neuron. Performance for the static Kalman filter (optimized prior to the loss) before and after nonstationarity is shown in Fig. 5.1.



(A)



(B)

Figure 5.1 Effect of 50% neuron loss on non-adaptive decoding performance. (A): Decoded movement from a 100 neuron population for a five second long horizontal 'Figure of 8' movement. The optimal decoded signal (blue) for the static Kalman filter closely approximates the desired movement (red) with a 100-neuron population. Accuracy suffers when half of the population is lost (green). (B): Effect of 50% neuron loss on adaptive algorithm decoding performance. All decoded signals were low pass filtered at 5 Hz for visibility (4^{th} order zero-phase Butterworth filter).

5.1 Results

Decoding accuracy was quantified by computing the root mean square errors normalized to the RMS power of the test movement stimulus (RMS power = 1 in this case). Figure 5.2 shows the normalized root mean square errors (NRMSE) averaged across 20 simulations with a 100 LIF neurons computed over the last 1450 seconds of the test stimulus.

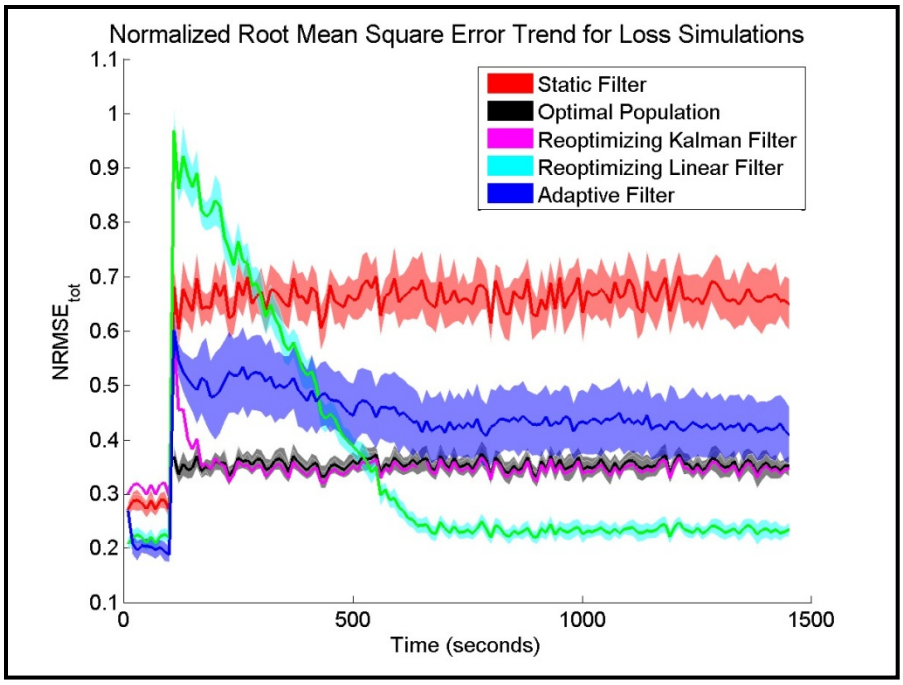


Figure 5.2: **Normalized root mean square error (NRMSE) in response to an instantaneous loss of 50% of the neural populations.** NRMSE is shown for the static Kalman (red), adaptive Kalman (blue), reoptimizing Kalman (magenta) and reoptimizing linear filters (green) averaged across 20 simulations. Errors were computed over a 10 second non-overlapping window. The shaded regions denote the 95% confidence interval in the mean NRMSE across simulations. One hundred seconds into the simulation, 50% of the neuronal population (50 of a 100 neurons) was lost, i.e. no responses were recorded from these neurons. The final error for the reoptimizing linear filter is the lowest among the adapting filters. The reoptimizing Kalman filter error recovers within 100 seconds to the level of the optimal Kalman decoder. The adaptive filter error approaches the optimal decoding but does not recover to that level ($t(198) = 51.02$, $p \ll 0.01$, t-test).

The adaptive filter had the lowest errors (NRMSE=0.189) under stationary conditions (i.e., first 650 seconds of the simulation), consistent with the results in Chapter 4 with the test signal at RMS power of 1 and a bandwidth of 0 – 1 Hz. Since the decoding errors were normalized to the RMS power of the test stimulus, an error of 0.189 signifies 18.9% error in the output movement. The reoptimizing linear filter performed slightly worse at 0.221 NRMSE.

The errors for all decoding filters increased due to the abrupt loss of neurons with the reoptimizing linear filter error the highest at 0.967 NRMSE. Over time the repoptimizing linear filter recovered to 0.227 NRMSE after 580 seconds, which is near its pre-loss level. The reoptimizing Kalman filter error increased to 0.594 NRMSE and recovered very quickly (within 90 seconds) to the level of the optimal decoder (NRMSE = 0.345). The adaptive filter error increased at the onset of neuron loss (NRMSE = 0.601) and recovered to a level of 0.42 NRMSE after 550 seconds.

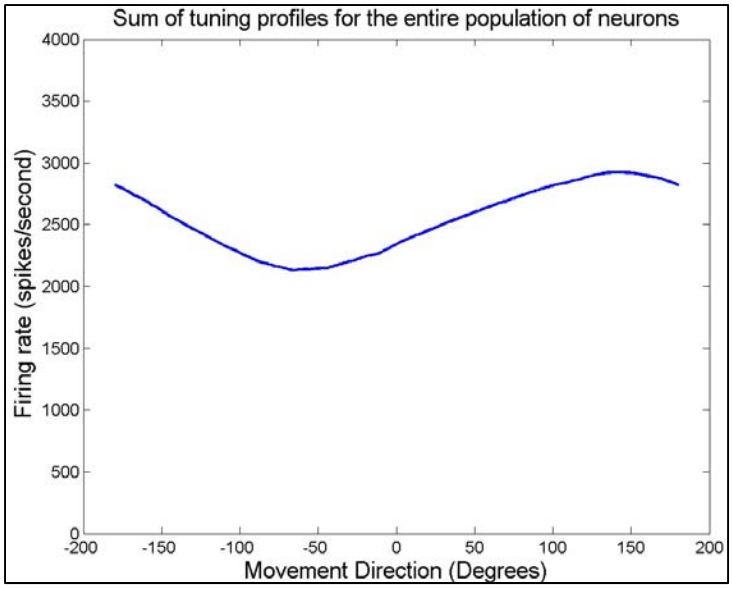
5.2 Discussion

The adaptive filter represented the best decoding case under stationary conditions. After the onset of the non-stationarity which increases the decoding error for all decoding filters, the reoptimizing linear filter slowly recovers (580 seconds) to the lowest decoding error while the reoptimizing Kalman filter quicklyrecovers its performance to the level of the optimal decoder. The adaptive filter performance recovered slowly to a better accuracy after the loss, but did not reach the error level of the optimal decoding.

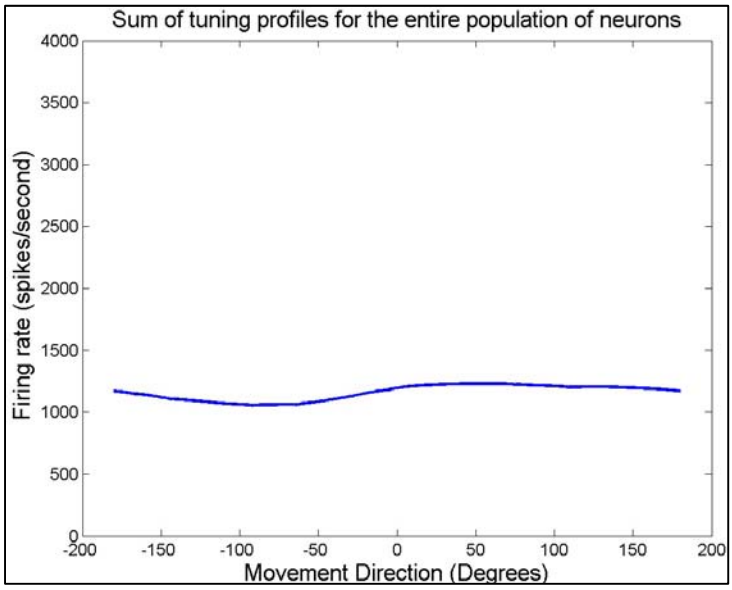
For any decoding system, loss of neurons translates into a loss of information available for use in decoding. The initial optimization process is responsible for determining which neuron firing rates are most useful for decoding the movement space.The adaptive decoding system is based on a Kalman decoding scheme that attaches a single optimal decoding weight to each stimulus dimension for each neuron in order to determine its relative contribution to the overall stimulus. The optimal decoder, with weights optimized to the residual 50 neurons, provides a benchmark for the best-case performance of a Kalman filter. The linear filter, on the other hand, employs multiple

decoding weights for each stimulus dimension and neuron (20 weights/neuron/dimension in this case of a one second filter) that are determined during the initial optimization process. Therefore, the linear filter takes advantage of this history of the neuron responses that is inherent in its' decoding scheme to provide a better recovery performance than the (best-case) optimal Kalman filter. The reoptimizing Kalman does not have access to this history of neuron responses, but operates over the same 550 second reoptimizing time window to optimize the decoding weights and drive the error of the decoding post-nonstationarity to approximate to that of the optimal Kalman decoder.

If the lost neurons were to encode a specific region of the space that may have no (or sparse) representation in the residual 50 neuron population, less information about the encoded movement stimulus would be available to the decoding algorithm. Since the decoding algorithm uses the current neuron response in order to estimate movement, this would increase the error in decoding. However, the remaining neuronal population adequately sampled the movement space in these simulations. Figure 5.3 (A) shows the sum of tuning response profiles for the entire population of neurons for a simulation while Figure 5.3 (B) shows the sum of the tuning profiles for the same neurons after the loss has occurred. No particular movement direction shows a drop in the firing rates compared to the rest. This would indicate that the loss impacted all movement directions. Also, the decoding errors plotted in Figure 5.2 were computed across 20 different simulations with different neuron populations so that neurons lost would not specifically encode for a particular movement space.



(A)



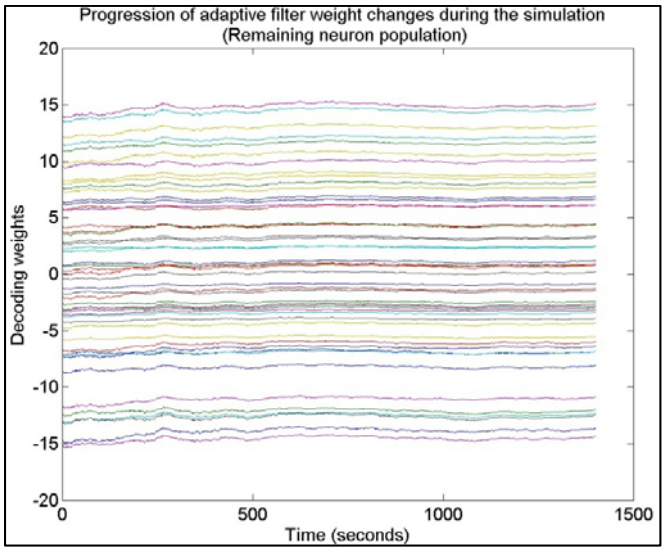
(B)

Figure 5.3: **Population response as a function of movement direction** (A) before and (B) after loss of 50% of the neural population. The shape of the population response as a whole is fairly uniform for (A) and (B). Thus, the lost neurons did not result in a loss of direction information from the encoded movement.

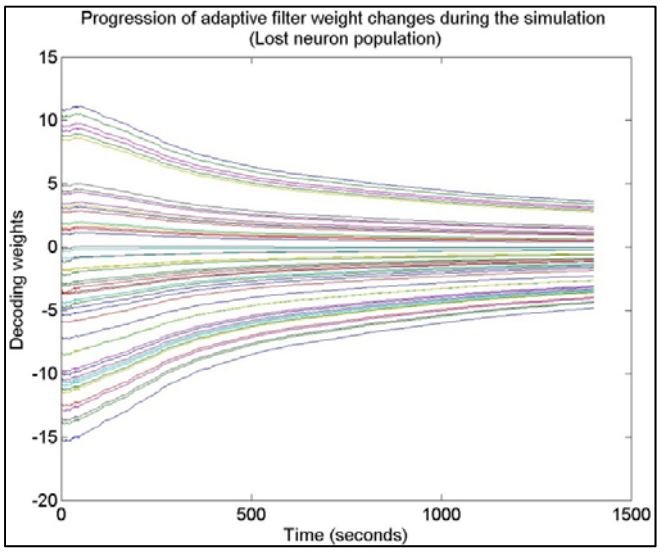
The reoptimizing Kalman filter approximates to the optimal decoder performance and does so quicker than expected due to the form of the re-optimization process. The re-optimizing Kalman filter adjusts its decoding weights at each timestep by minimizing the error over the preceding 550 second window. The rate of weight modification is directly related to the overall error in this window. Since we simulated an abrupt catastrophic nonstationarity that drives the overall error high, the weight reoptimization is influenced very early in the reoptimizing window and therefore, a quick recovery is observed.

Since less information is available about the encoded movement, the decoding performance of the adaptive filter is bound to suffer (see Eliasmith et al. 2002 – Chapter 2). The error increases with lower number of neurons available for decoding movement thus describing an inverse (1/N) relation. It was not expected that the adaptive filter would be able to recover to the level of its initial decoding performance with a 100 neurons since the information input to the system is reduced. The adaptive filter scheme is based on a gradient descent that adjusts the Kalman decoding weights by making instantaneous updates to the weights every 50 ms. The adaptive filter error reduces systematically after the nonstationarity is registered, however, the speed of recovery is slow due to the instantaneous properties of the stimulus when the nonstationarity occurs. As was observed in Chapter 4, the adaptive filter seeks to continuously optimize the decoding weights to the instantaneous region of the movement space at each 50 ms timestep. If the nonstationarity occurs at a movement timestep with high decoding error, the gradient descent scheme can be thrown off its path toward the global error minimum for the system. However, no such weight change was observed in our simulations. The adaptive filter acts to drive down the weights associated with the neurons that are lost

while preserving the weights of the neurons that are still present in the population as shown below in Figure 5.4.



(A)



(B)

Figure 5.4: **Change in decoding weights along one (X) dimension over time for** (A) 50 unaltered neurons and (B) 50 neurons that were lost from a 100 unit neuronal population. The filter weights for the 50 neurons that were lost see a decrease in the

weights associated with them. The adaptive filter retains the weights associated with the remaining 50 neuron population.

The reoptimizing linear filter has more weights per neuron than the Kalman-based decoding filters. In order to minimize the error over its 550-second reoptimizing window, it has to optimize multiple weights associated with each neuron when compared with the reoptimizing Kalman approach that only uses one weight per neuron. The reoptimizing linear filter would not benefit from making large weight changes to individual weights associated with each neuron. It makes incremental changes to each weight for each neuron at every timstep and therefore, while it is able to recover from its high decoding error, it takes ~570 seconds to do so as opposed to the ~100 second recovery shown by the reoptimizing Kalman filter. It shows the largest increase in error due to the nonstationarity but recovers to a lower error than an optimal decoding of 50 neurons with a Kalman filter (which is not in agreement with (Wu et al. 2008) whose linear decoding filters performed worse than comparable Kalman decoding filters).

6 SIMULTANEOUS LOSS AND RECRUITMENT OF NEURONS

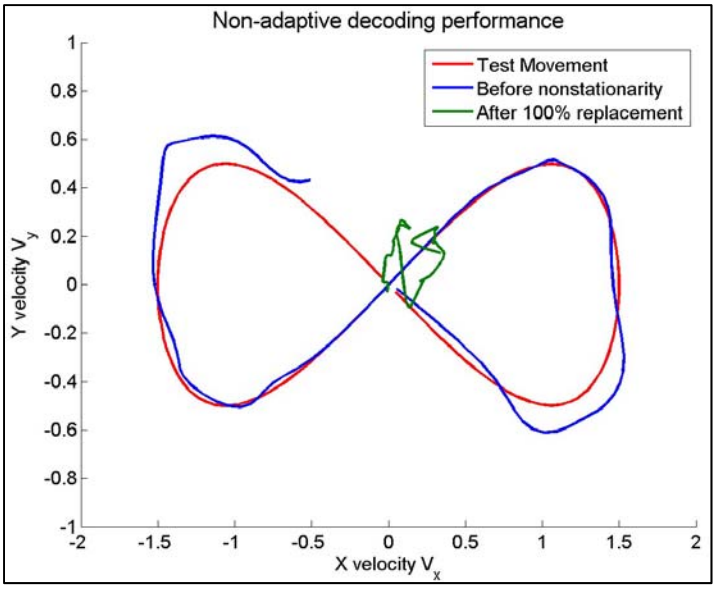
The decoding algorithm relies on a set of decoding weights that are specific and optimized for the subset of the neuron population sampled by the electrode implant. Over time, movement of the electrode array can result in some neural signals being lost, as the array moves away from some neurons, and others being newly acquired with potentially different tuning characteristics to the neurons that were lost. Since the decoding optimization is specific to the decoded movement parameters and the tuning responses of the population, such "drift" in the electrode array is computationally similar to changing the tuning characteristics of the neurons in the sampled subset.

Changes in the shape (tuning characteristics) of recorded neurons have been observed by (Xindong Liu et al. 1999) and more recently by (Suner et al. 2005). In the Liu et al study, the stability of neural recordings was characterized over a number of months. The authors reported that recorded neural activity was unstable up to 4 – 8 weeks post implantation. While neural activity stabilized after this period, slow changes in the recordings were omnipresent. The authors provided evidence for electrode movement through the tissue, which resulted in previously active units being lost. Growth of connective tissue may have contributed to the movement of the electrode array along the cortical surface or into deeper layers of the cerebral cortex.

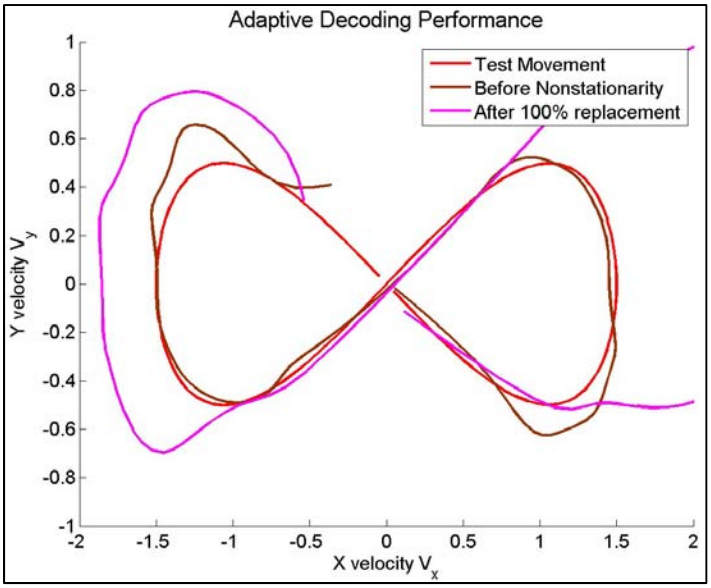
Suner et al (2005) reported similar results in Macaque motor cortex. Over time, signals within individual recording channels disappeared while channels with low (or no) signal strength sometimes started recording signals. The shape of the recorded waveforms

varied across days for the chronic implants. Over a period of 91 days, action potential waveforms retained their shape 38% of the time. While this may suggest electrode movement and/or changes in neuron tuning characteristics, the effects of both are similar with respect to decoding algorithm performance.

For these simulations, a 100-neuron population was simulated with the same properties as described in Chapter 4. To simulate simultaneous loss and recruitment associated with a shift in the electrode array, a population of 100 neurons was abruptly replaced with a 100 *novel* neurons. Both the original and the new populations had the same aggregate response properties (Chapter 3). Figure 6.1 shows an example of the effect on (static) Kalman filter decoding of replacing the entire population. Since the weights were optimized to the original population, replacing each neuron with an 'unknown' neuron randomized the relative contributions of neurons' responses, significantly impairing performance.



(A)



(B)

Figure 6.1: Effect of 100% neuron replacement on non-adaptive decoding performance. (A) Decoded movement from a 100 neuron population for a five second long horizontal 'Figure of 8' movement. The optimal decoded signal (blue) closely approximates the desired movement (red) with weights optimized for a 100-neuron population. Following instantaneous replacement of the entire population (green), accuracy of the decoded movement was poor. The reconstructed signal does not approach the intended movement in either amplitude or direction. (B): Effect of 100% neuron replacement on adaptive algorithm decoding performance. All decoded signals were low pass filtered at 5 Hz for visibility (4th order Butterworth filter).

6.1 Results and Discussion

Figure 6.2 shows the normalized root mean square errors (NRMSE) averaged across 20 simulations computed over the last 1450 seconds of the test stimulus length.

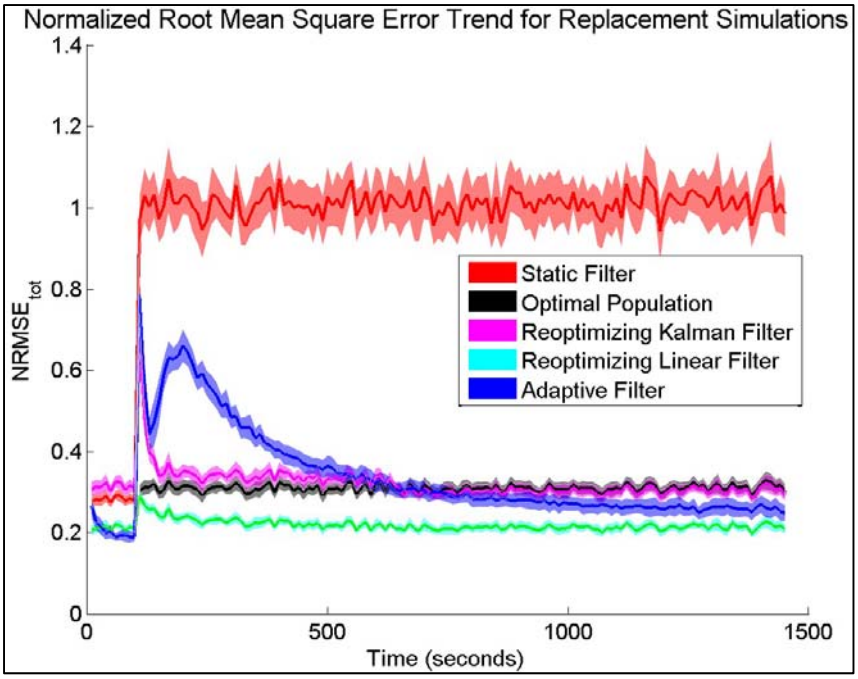


Figure 6.2: **Normalized root mean square error (NRMSE) in response to an instantaneous replacement of 100% of the neural population.** NRMSE is shown for the static Kalman (red), adaptive Kalman (blue), reoptimizing Kalman (magenta) and reoptimizing linear filters (green) averaged across 20 simulations. Errors were computed over a 10 second non-overlapping window. The shaded regions denote the 95% confidence interval across simulations. The entire neuronal population was replaced with novel neurons 100 seconds into the simulation. The reoptimizing linear filter shows the lowest errors when compared with the adaptive and the reoptimizing Kalman filters. The adaptive filter recovers to an error level lower than the reoptimizing Kalman filter. Pre-nonstationary errors for the adaptive filter are similar to the reoptimizing linear filter but better than the reoptimizing Kalman filter.

Following the switch in neuronal population characteristics Six hundred and fifty seconds into the simulation, static Kalman filter errors increased to 1.05 from a pre-nonstationary error of 0.3. This indicates extremely poor decoding accuracy (error > 100%) of the intended movement. During the initial Kalman filter optimization, the optimal decoding weight for each neuron was computed. Since the entire population was replaced, it is unlikely that any of the decoding weights remained optimal with respect to its newly associated neuron. Hence, a catastrophic increase in error is to be expected.

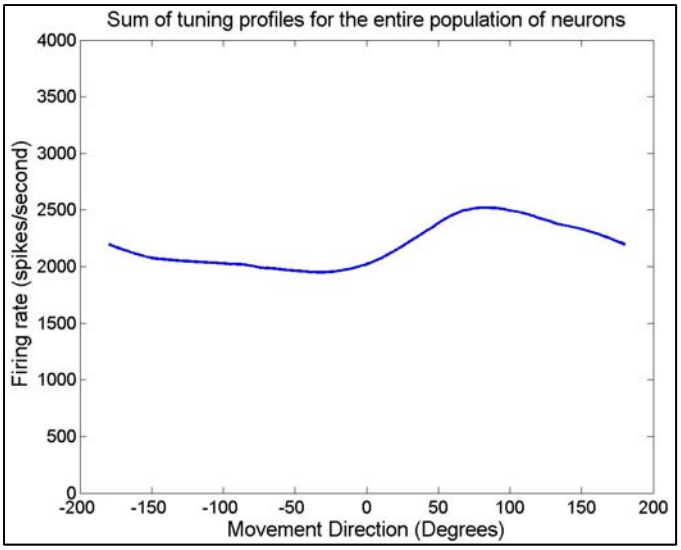
The reoptimizing Kalman filter errors increased to 0.64 and recovered within 50 seconds to 0.36 (NRMSE) which is close to the optimal decoding error (NRMSE=0.32). The error further decreased to match the optimal decoder performance after the switch in population moved beyond the 550 second optimizing window. Total replacement of the neuronal population is an abrupt catastrophic nonstationarity that dominates the overall movement error within the 550 second optimization window. The filter then corrects for the large error by making correspondingly large changes to the weights, which is why a quick recovery to a low error is observed. Once the nonstationarity (at 100 sec) has completely passed through the reoptimizing window, the system is once again optimal and therefore behaves exactly like the optimal Kalman decoder.

The adaptive filter error increased to 0.8 immediately after the population replacement and decreased to the level of the optimal decoder after approximately 500 seconds. The decrease in error was non-monotonic – an initial fast decrease in error to 0.45 after 40 seconds followed by a increase in error to 0.61 NRMSE at 200 seconds, after which the error decreased systematically to its final value (NRMSE = 0.268) after 980 seconds. Following the population replacement, the rate of weight modification was large due to the large initial error. The adaptive filter shows a recovery beyond that of the optimal Kalman decoder, resulting in a NRMSE of 0.25, which is slightly higher than its pre-nonstationarity error (~NRMSE = 0.2). As seen in Chapter 4, the adaptive filter is able to make modifications to the weights based on the not just increased error attributable to nonstationary changes but also that due to the instantaneous properties of the stimulus, thus resulting in a lower error than the optimal decoder which has a set of unaltered decoding weights. To investigate the non-monotonic change, the algorithm was

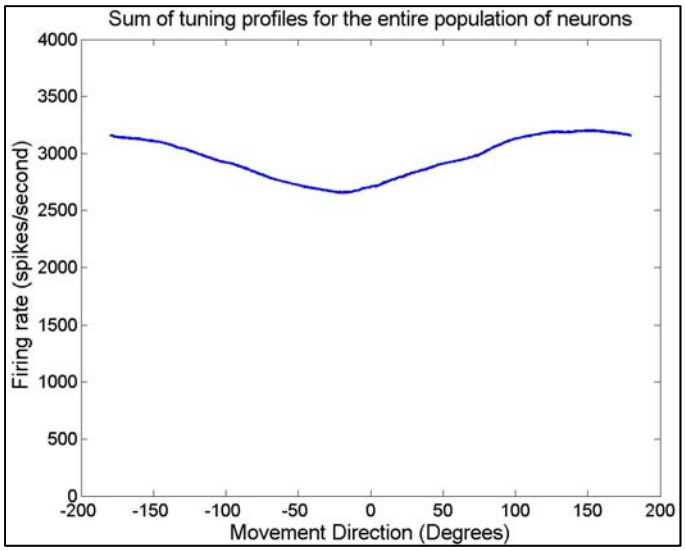
tested with a known set of sinusoids and predefined decoding weights. A monotonic decrease in the RMS error was observed for this test case (see Appendix B).

The reoptimizing linear filter shows the best performance among the adaptive filters for these simulations. It matches the adaptive filter before the introduction of nonstationarity (NRMSE = 0.2) and recovers to this level following full replacement of the population. The nonstationarity itself did not cause a high increase in error; and its recovery occurred within approximately 200 seconds.

The reoptimizing linear filter has more weights per neuron (and hence more degrees of freedom) than the Kalman-based decoding filters. In order to minimize the error over its 550 second reoptimizing window, it has to optimize multiple (20) weights associated with each neuron when compared with the reoptimizing Kalman approach that only uses one weight per neuron. However, its decoding performance does not deteriorate catastrophically. The reoptimization aims to correct the linear decoding weights so as to minimize the decoding error within its reoptimizing time window (550 seconds) given the overall distribution of the neuron firing rates in the encoded movement space. The distribution of firing rates for the initial population is similar to that for the population that replaced it as shown in Figure 6.3 below. Thus, no large loss in decoding accuracy is seen for this filter.



(A)



(B)

Figure 6.3: **Population response profiles (A) before and (B) after complete replacement of the neuronal population.** The shape of the population response profile is largely uniform. Thus, the new neuron population adequately samples the movement space and direction information of the encoded movement is retained.

The timecourse for the weight changes for the adaptive decoding filter for one of the simulations is shown in Figure 6.4. The weight changes are consistent with that

expected for going from one 100 neuron population to an entirely new population of neurons.

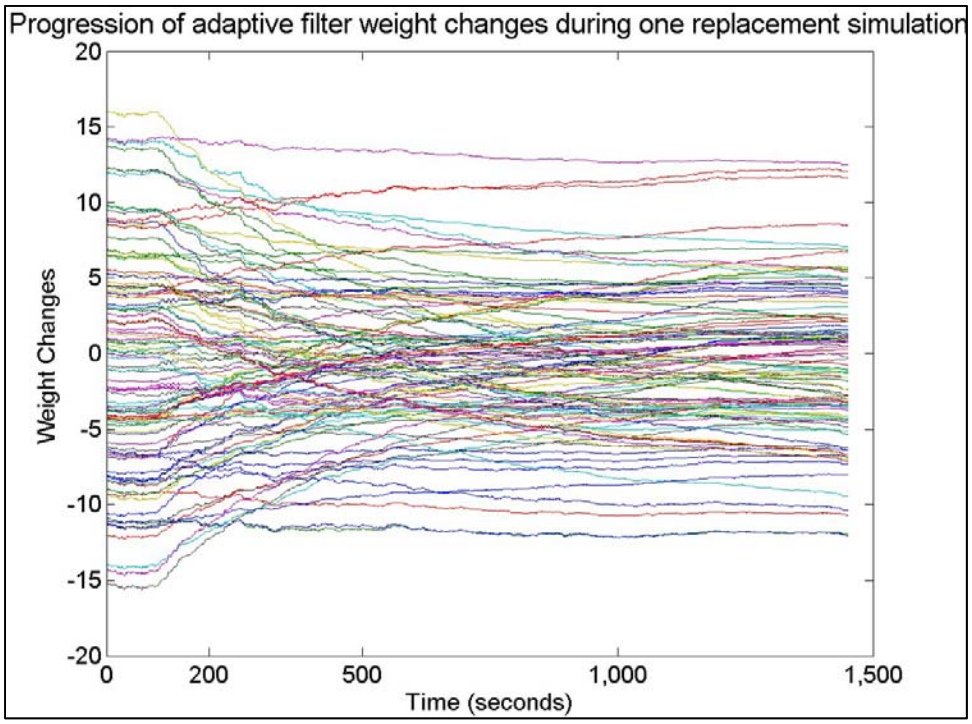


Figure 6.4: **Weight changes along the X-dimension for the adaptive decoding filter for one simulation.**

The replacement of motor neurons may potentially not be as catastrophic as neuron loss alone since the encoded stimulus is still represented by the same number of neurons; therefore the overall information about the stimulus is retained. This affords the opportunity to the adaptive filter to correct for the presence of the novel neurons by changing the values of the decoding weights associated with each neuron.

Point process adaptive filters (Eden et al 2004, Srinivasan et al 2007) have been shown to be resistant to slow changes in neural response properties but it is unclear how such systems would perform under more extreme conditions. For neuron replacement at a rate of one per minute, the adaptive filter proposed by Eden and colleagues (Eden et al

2004), was able to perform well when reconstructing movement direction from a population of 20 neurons but was not able to consistently recover speed of movement. Srinivasan et al 2007 showed similar trends in performance when neurons were replaced at a rate one per minute. When an equivalent rate of replacement was simulated here, there was no observable effect on performance using the adaptive Kalman filter (Figure 6.5). This performance is better than that of the proposed point process adaptive filters described above. Thus, the adaptive Kalman filter is able to recover in the presence of a catastrophic nonstationary replacement of the population instantaneously and shows good performance in the presence of a slower real world (Wu et al 2008) neuron replacement.

Before the introduction of the nonstationarity at 100 seconds into the simulation, the adaptive filter decoding errors are the lowest. The optimal decoding uses the weights that were optimized for the new 100 neuron population and therefore, represents the best case Kalman filter error for this new population. After approximately 950 seconds into the simulation, the adaptive filter errors increase to match the optimal decoding errors that are the best case error for the new 100 neuron population.

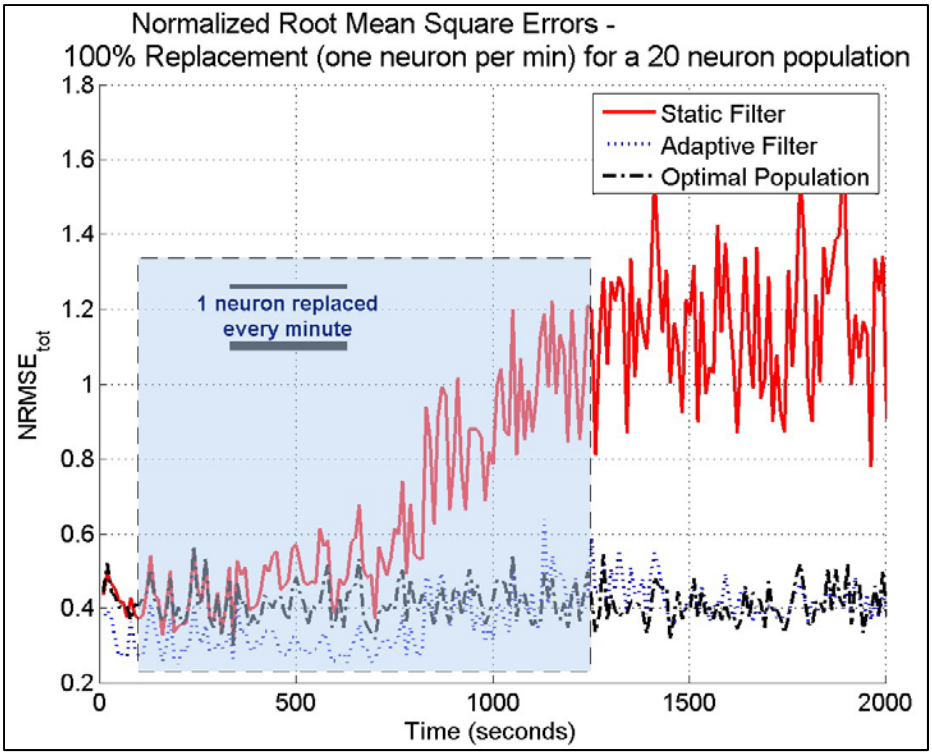


Figure 6.5: **Decoding errors for one simulation with complete replacement of a 20-neuron population at the rate of one neuron per minute.** One hundred seconds into the simulation, one neuron was replaced by a novel neuron every minute. The decoding performance of the adaptive Kalman filter approaches the optimal Kalman decoder after introduction of the nonstationarity.

7 ATTENTION

7.1 Attention Modulation

Attention has been shown to modulate neuron responses in cortical areas including the primary motor cortex. Johansen-Berg et al (2002) showed differential activation of primary motor cortex when subjects were asked to count backwards while performing a movement (button press). The backward counting from a three-digit number was intended to act as a distractor for the subject thus reducing attention to the movement task. The experimenters observed decreased responses in the primary motor cortex when both tasks were performed simultaneously as opposed to the condition when the subjects performed just the movement task.

Attention has also been shown to modulate the tuning curves of neurons in primary visual cortex (Chen et al. 2008), visual area V4 (McAdams et al. 1999), parietal cortex (Quraishi et al. 2007) and motor areas (Binkofski et al. 2002). When attention is allocated to tasks processed by these areas, changes in the amplitude, tuning width, background rate and preferred orientation of neuron responses have been reported (see McAdams et al. 1999). The decoding of movement depends on the neuronal responses from the given population of neurons. Unaccounted for changes to the neuronal parameters described above cause a loss of accuracy due to the decoding being no longer optimized to the attention modulated neuronal responses.

For these simulations, a 100-neuron population was simulated with the same properties as described in Chapter 4. To examine the effects of attention on decoding

performance we modulated the amplitude (i.e. firing rate) of neurons' responses to simulate the effects of attention. The instantaneous firing rates of the simulated neurons were modulated by ±20% (via the driving current $\boldsymbol{J_d}$), using a shifted sine wave signal with a period of five seconds scaled to the range [0.8, 1.2], (McAdams et al 1999 report 26% change in neuron response amplitudes brought about by attention).

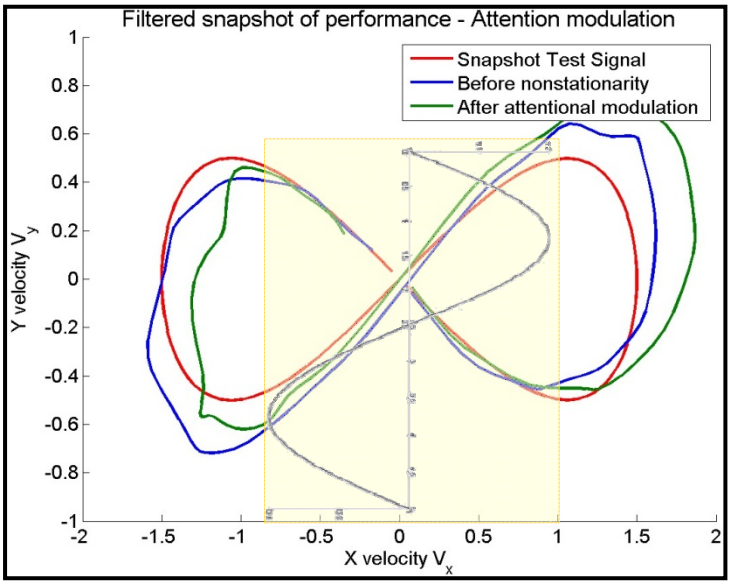


Figure 7.1: Effect of attention gain modulation of the neuron responses on non-adaptive decoding performance. A five second long horizontal 'Figure of 8' movement was decoded from 100 neurons responding to movement in a two-dimensional space. Attention was modeled as a ±20% sinusoidal modulation of neuronal responses with a period of five seconds. A corresponding increase in decoded velocity values is seen on the right (with increased attention) and a decrease on the left (when attention is reduced).

Figure 7.1 compares the decoding performance of a non-adaptive optimal filter with attention modulation of the neuron responses. The filter weights were optimized to responses from the neurons when no attention modulation was present. Responses for the first 2.5 seconds were enhanced (i.e. for the right half of the figure of eight). The increased responses resulted in higher velocity estimates than intended. For the next 2.5 seconds, responses were suppressed (left half) and a corresponding drop in decoded

velocity estimates was obtained. It is important to note that since the weights of the filter were optimized without attentional modulation, a loss in accuracy is seen when attention is included. In either case, the decoded estimates incorporate higher errors than the optimal decoding of the velocity from the neural responses.

7.2 Results and Discussion

The normalized root mean square errors (NRMSE) for a simulation containing 100 LIF neurons responding to a 0 - 1 Hz bandlimited white noise movement with a RMS power of 1 computed over the last 1450 seconds of the test stimulus length are shown in Figure 7.2.

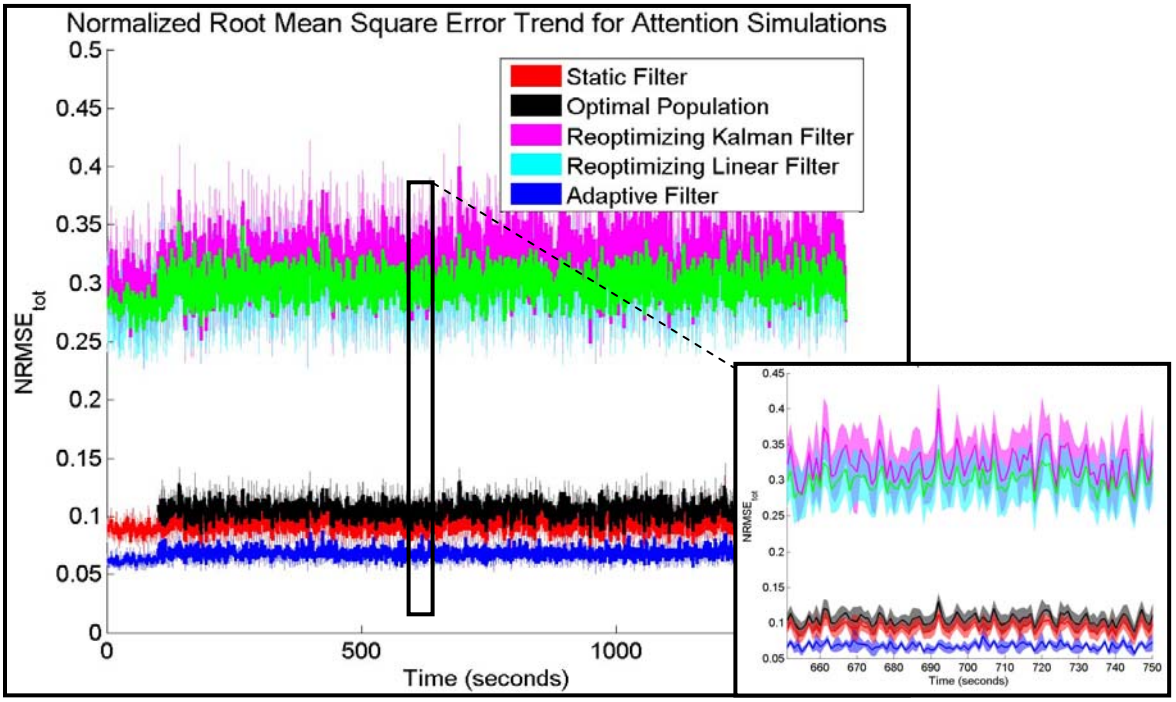


Figure 7.2: **Normalized root mean square error (NRMSE) in response to attentional modulation of neuron firing rates.** NRMSE is shown for the static Kalman (red), adaptive Kalman (blue), reoptimizing Kalman (magenta) and reoptimizing linear filters (light blue/green) averaged across 20 loss simulations. Inset: A 100 second section illustrating the change in error over each second. Since the period of attentional

modulation was set to 5 seconds, errors were computed over a 1 second non-overlapping window to capture the effect of attention on decoding error. The shaded regions illustrate the 95% confidence in the mean NRMSE across 20 simulations. Attention modulation of neuron responses began 100 seconds into the simulation. The optimal population decoding represents the performance of a decoding filter with weights optimized to the attention modulated responses. The adaptive filter performance is better than both the reoptimizing Kalman filter and the reoptimizing linear filter.

As shown in Figure 7.3, the errors for all decoding filters show periodicity at 0.2 Hz and 0.4 Hz due to the 5-second long attention modulation. The reoptimizing Kalman filter and the reoptimizing linear filter have the highest concentration of error at 0.2 Hz and 0.4 Hz respectively. Adaptive Kalman filter errors also show the periodicity, but the errors are the lowest of all the adapting filters.

The attentional modulation was initiated six hundred and fifty seconds into the simulation. Attention modulation does not seem to have as large an effect as the other nonstationarities. With an instantaneous 50% loss of the population (see Chapter 5), the adaptive filter errors rose to 0.601 NRMSE while the reoptimizing linear filter errors and the reoptimizing Kalman filter errors increased to 0.967 and 0.594 NRMSE respectively. With attentional modulation, the errors were 0.072, 0.352, and 0.379 NRMSE for the adaptive Kalman, reoptimizing linear filter and the reoptimizing Kalman filters respectively.

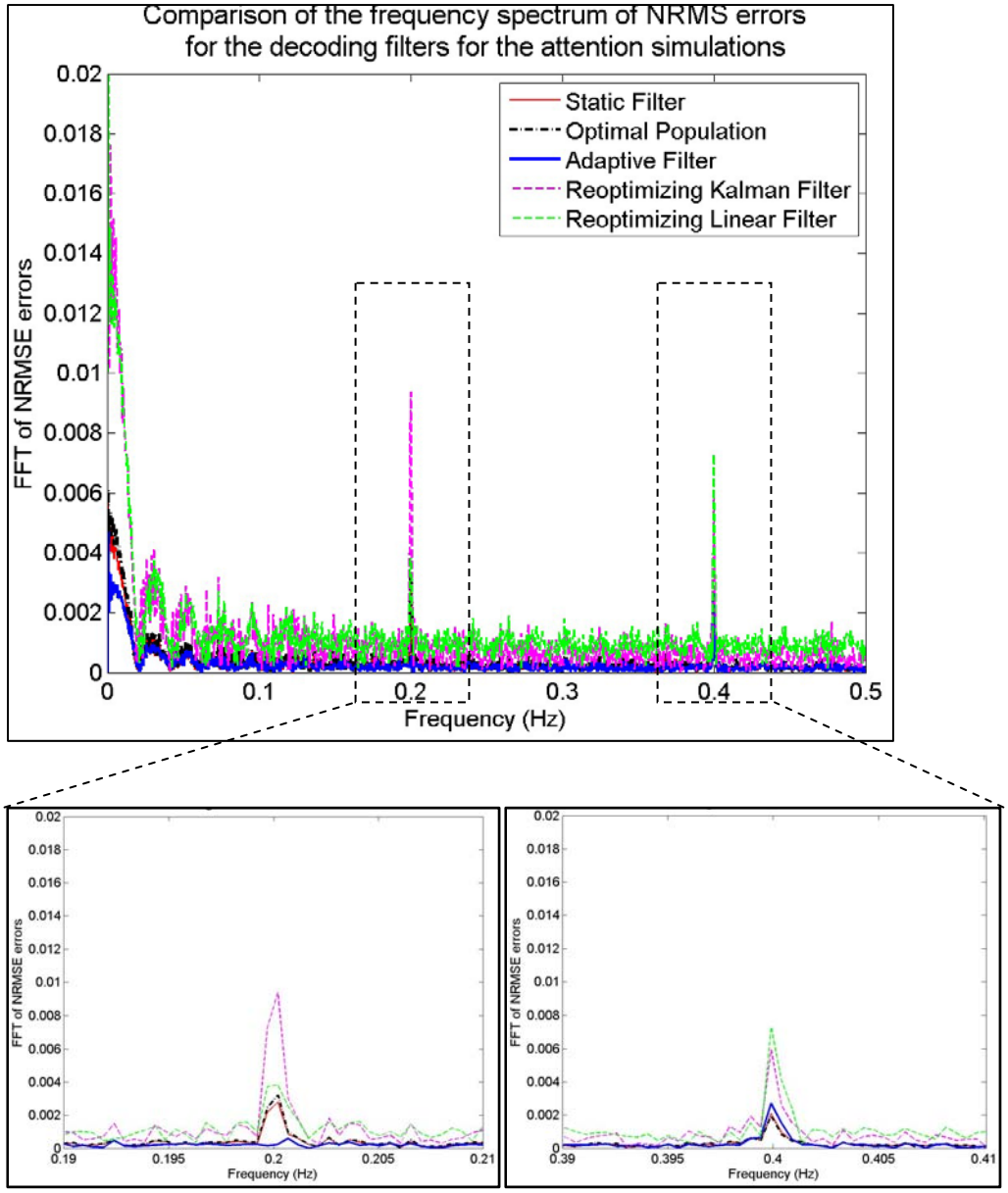


Figure 7.3: **Frequency spectrum of the normalized root mean square errors (NRMSE).** Peaks of the errors for all filters are seen at 0.2 Hz and 0.4 Hz. Inset (left): Errors at 0.2 Hz. Inset (right): Errors at 0.4 Hz.

The reoptimizing linear filter suffers a small increase in error from 0.278 to 0.352 NRMSE. The reoptimizing Kalman filter shows an increase in error from 0.288 to 0.379 NRMSE. No discernable recovery for the reoptimizing Kalman and the reoptimizing linear filter is seen. Both the reoptimizing linear filter and the reoptimizing Kalman filters

modify their decoding weights at each timestep by minimizing the error over their reoptimizing window, which is 550 seconds long for both filters. The rate of weight modification is directly related to the overall movement error in this window. The modulation of neural responses due to attention is periodic over 5 seconds which is a smaller time scale when compared to the window length. Therefore, the attentional modulation does not drive the error within this 550 second window high enough to influence the weight reoptimization. The attentional modulation is symmetric (as illustrated in Fig 7.1 and therefore the net signed error over this window length is small. This results in an inherent uncertainty that is present and remains the same within each successive 550 second window. When compared to an instantaneous 50% loss of the population, since the attention modulation is 20% of the neuronal responses, the effect of attention is not as catastrophic and more importantly, the reoptimizing filters see a large error in their reoptimizing window that they try to minimize over successive iterations. Due to attention modulation, the relation between the rate responses, weights and stimulus location is no longer constant. Thus, it is difficult for a reoptimizing filter to adjust to the attention induced neuron reponse changes and no recovery is seen.

The adaptive filter scheme, on the other hand, is based on a gradient descent scheme that influences the Kalman decoding weights and it makes instantaneous updates to the weights every 50 ms. This allows the weights to adjust to attention modulations that occur over longer timescales (e.g., seconds). Therefore, the adaptive filter error trend reduces after the introduction of attentional modulation. The adaptive filter continuously reoptimizes its decoding weights to the instantaneous region of the movement space at each 50 ms timestep (see Chapter 3). Its final error is the lowest of all the decoding filters

(as seen in Figure 7.2) and due to its ability to adjust to the instantaneous properties of the stimulus, it outperforms the optimal decoding filter.

8 ADAPTATION

8.1 Neuronal Adaptation

Neurons that are exposed to the same constant or time-varying stimulus over a period of time adapt to the strength of the stimulus thus resulting in reduced neuron spiking (Connors et al. 1990). During chronic implantation of an electrode at the brain machine interface, neuronal adaptation may cause the neuron responses to decrease over time, effectively changing the tuning characteristics of the neurons. Thus decoding algorithms optimized with dynamic stimuli may no longer be optimal when faced with repetitive or slowly time-varying stimuli. Because adaptation produces a change in the neuron tuning characteristics (e.g. firing rate), the decoding performance of a non-adaptive linear decoder would be inaccurate.

Spike frequency adaptation, commonly seen in ‘regular-spiking’ neurons (Connors et al. 1990) defines the adaptive behavior of the neuron once a spike is generated. In these neurons, after-hyperpolarization causes an increase in the membrane conductance following each action potential. This causes an increased difference between the threshold voltage and the resting potential thus increasing the time to reach threshold and generate a subsequent spike. Thus a drop in the spiking frequency (firing rate in spikes/sec) of the neuron is seen when constant stimuli are presented over a period of 50 – 600 ms.

8.2 Adaptive LIF Neurons

The properties of neuronal adaptation can be approximated using an Adaptive Leaky Integrate and Fire (Adaptive LIF) neuron model (Eliasmith et al. 2002; Koch 1998). In the adaptive LIF neuron, a voltage dependant resistance is added to the normal LIF neuron, which acts to increase the interval between successive action potentials. Because this variable resistance is in parallel with the resistive – capacitive circuit of the LIF model (Figure 8.1), it reduces the current available to the capacitor to integrate to the threshold voltage Vth.

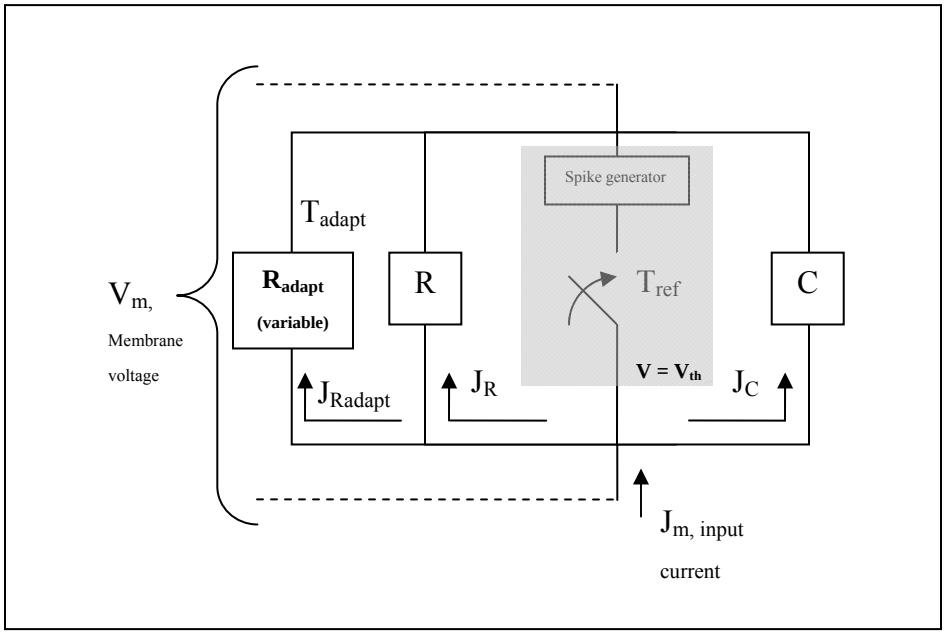


Figure 8.1: Adaptive Leaky Integrate and Fire Neuron. The resistive-capacitive circuit of the LIF neuron model, is placed in parallel with a variable shunt resistance, R_{adapt}, whose value varies dynamically in response to a stimulus in the neuron's preferred direction that excites an action potential (Koch 1999; Eliasmith et al. 2002).

The variation of the adaptive neuron resistance $\boldsymbol{R_{adapt}}$ is described as follows. On the occurrence of an action potential (spike), $\boldsymbol{R_{adapt}}$ is decreased by a fixed value $\boldsymbol{R_{dec}}$, i.e.

$$\boldsymbol{R_{adapt} = R_{adapt} - R_{dec}}$$

When there is no input (or no spike is generated), $\boldsymbol{R_{adapt}}$ increases exponentially towards its resting state, i.e.

$$\boldsymbol{R_{adapt} = R_{adapt} + dR_{adapt}},$$

$$\frac{\boldsymbol{dR_{adapt}}}{\boldsymbol{dt}} = \frac{\boldsymbol{R_{adapt}}}{\boldsymbol{\tau_{adapt}}} \quad \ldots (7.1)$$

In Chapter 3 (equation (3.4)), we saw that the value of the threshold current $\boldsymbol{J_{th}}$ is set by the leakage resistance $\boldsymbol{R}$. For an adaptive LIF neuron, this resistance is in parallel with the adaptive resistance $\boldsymbol{R_{adapt}}$; therefore a change in $\boldsymbol{R_{adapt}}$ produces a change in the RC time constant of the neuron thus impacting its ability to produce spikes at its maximum response even when it encodes stimulus at its preferred direction.

8.3 Simulation

A 100-neuron population was simulated with the same properties as described in Chapter 4 to investigate the effects of adaptation on the decoding accuracy. The decoding of movement through a linear filter is directly related to the optimized weights and the neural responses encoding that movement. The rates for the static Kalman decoding filter were optimized for non-adapting neurons. The reduction in neuronal responses with adaptation, can reduce the effective gain of the decoded response. This effect can be seen

in Figure 8.3 (green). For the decoded movement shown, the parameters were set as shown in Table 8.1 below:

Simulation parameter	Symbol	Value
Adaptive Resistance	$\boldsymbol{R_{adapt}}$	20
Time constant for adaptation	$\boldsymbol{\tau_{adapt}}$	50 – 600 ms
Drop in resistance due to spiking	$\boldsymbol{R_{dec}}$	5

Table 8.1 Adaptive Leaky Integrate and Fire (LIF) neuron parameters used in the simulation. The resistance $\boldsymbol{R_{dec}}$contributes to the adaptive response of the neuron when encoding the stimulus at its preferred direction. The adaptive resistance and time constant control the rate of recovery of the neuron from its adaptive response to its resting state.

The time constant for adaptation is described by Liu and Wang (Liu et al. 2001) to be within the range 50 – 600 ms. Since the value of leakage resistance (in Chapter 3) is set to 1, Radapt is set to a comparatively high value (20) for the simulation and the drop in resistance is set to five to produce a noticeable change due to adaptation. These values were chosen so as to see a significant change in the neuron responses due to adaptation (as illustrated below in Figure 8.2) within the time course that was chosen for the simulations.

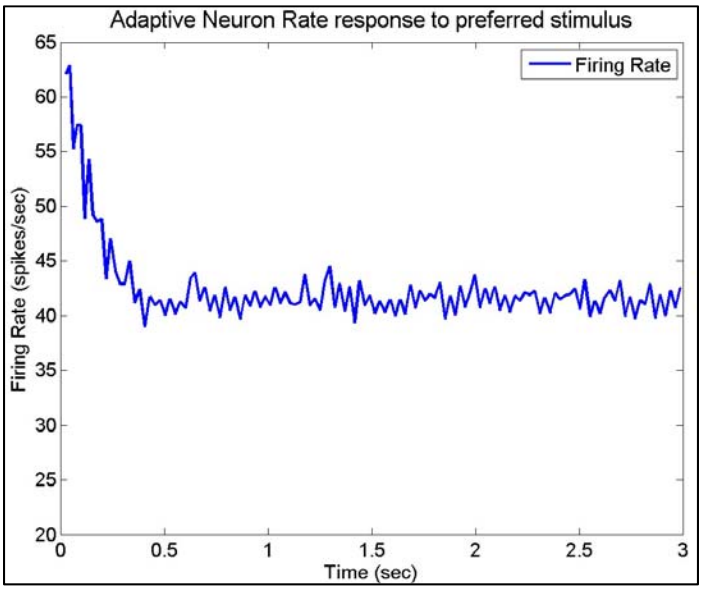


Figure 8.2: Effect of adaptation on the spike activity of a sample neuron. The firing rate of a neuron with a maximum firing rate of 80 spikes/sec at its preferred stimulus direction decreases when this preferred direction is present in the movement signal over a period of 3 seconds.

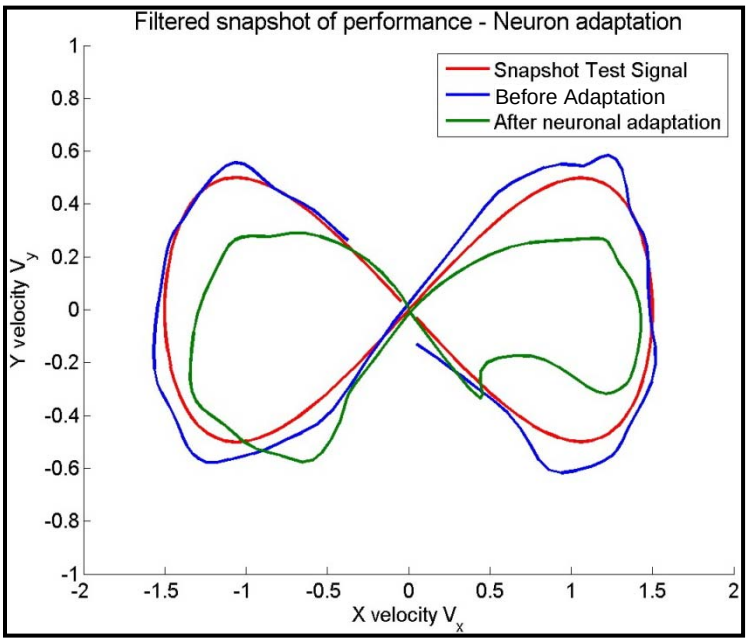


Figure 8.3 Effect of neuronal adaptation on non-adaptive decoding performance. A five second long horizontal 'Figure of eight' movement was decoded from 100 neurons responding to movement in a two-dimensional space. A loss in decoding accuracy was

seen with neurons adapting to the movement signal (green) as compared to decoding before adaptation (blue). Decoded signals were low-pass filtered at 5 Hz (4th order Butterworth filter).

8.4 Results and Discussion

The normalized root mean square errors (NRMSE) for a simulation containing 100 LIF neuron responding to a 0 - 1 Hz bandlimited white noise movement with a RMS power of 1 computed over the last 1450 seconds of the test stimulus length are shown in Figure 8.4.

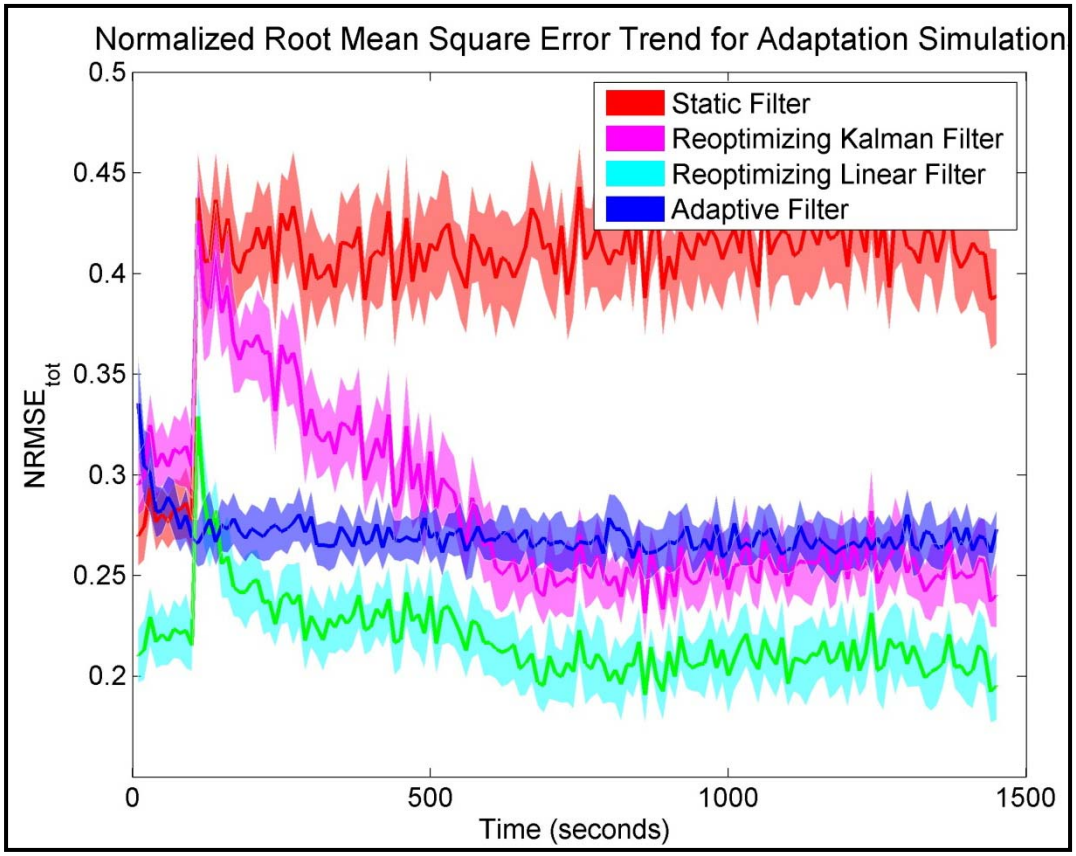


Figure 8.4: **Normalized root mean square error (NRMSE) in response to adaptation of the neurons to a bandlimited white noise stimulus with a RMS power of 1.** NRMSE is shown for the static Kalman (red), adaptive Kalman (blue), reoptimizing Kalman (magenta) and reoptimizing linear filters (light blue/green) averaged across 20 simulations. Optimal filter errors were exactly the same as the static Kalman errors and not shown here for clarity. Errors were computed over a 10 second non-overlapping

window. The shaded regions illustrate the 95% confidence in the mean NRMSE across 20 simulations. One hundred seconds into the simulation all neurons began adapting to the movement stimulus.

Neuron adaptation was introduced six hundred and fifty seconds into the simulation by setting R_{adapt} to a nonzero value. The static Kalman filter suffers an increase in error from 0.277 to 0.437 NRMSE. As described in Chapter 3, in the Kalman filter, the movement is decoded via the internal state variables using the product of the neural firing rates with its optimized Kalman coefficients. During adaptation, neural firing rates drop, such that the decoding weights are no longer optimal. This in turn results in decreased amplitude of the decoded movement.

The reoptimizing Kalman filter approaches the same level of error as the static Kalman filter following the onset of adaptation, but recovers to 0.247 NRMSE after 650 seconds. The effect of adaptation of the neural responses is not as catastrophic as the loss of 50% of the population. As seen in Chapter 5, an instantaneous loss of 50% of the population causes the decoding error to increase by approximately 200% while the change seen here is approximately 50%. The rate of change of the weights depends on the decoding error seen by the filter. Therefore, the rate of modification of the weights is comparatively slower and we see a slower recovery in the case of neuronal adaptation.

The reoptimizing linear filter, on the other hand, starts with a low pre-nonstationarity error of 0.210 NRMSE that increases to 0.329 NRMSE with the introduction of adaptation and recovers to pre-adaptation levels at 0.215 NRMSE after 650 seconds. The reoptimizing linear filter has more weights per neuron than the Kalman-based decoding filters. In order to minimize the error over its 550 second reoptimizing window, it has to optimize multiple weights associated with each neuron. It

is able to combat the effects of adaptation because it has more (20) weights optimized to each neuron. The effect of the slow change in the neuronal response due to adaptation is tempered by the multiple weights associated with that neuron since the estimated movement is a matrix product of the weights and the neuron response. This has an averaging effect on the computation of the predicted movement and thus, it shows low overall error when compared to the other filters. It reaches its optimal error about 650 seconds into the simulation (like the reoptimizing Kalman) at which point its weights are optimized to the adaptive responses.

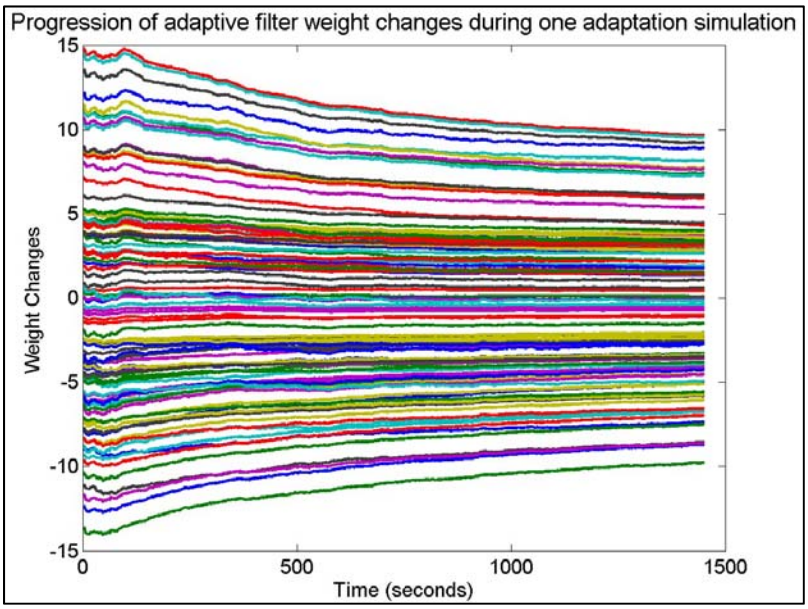


Figure 8.5: Progression of changes to the individual weights associated with each neuron for the movement along one (X) dimension for the population of adaptive neurons.

The adaptive filter has a pre-nonstationarity error at 0.268 NRMSE and is resistant to the effects of neuronal adaptation and ends up at 0.26 NRMSE. The adaptive filter optimizes its weights over each 50 ms time step and to the instantaneous properties of the test stimulus and the neuronal responses. The adaptation effects modeled here had

a time scale of between 50 to 600 ms, as discussed earlier. Compared to a sudden loss of 50% of the population or replacement of the entire population over 50 ms, the effects of the adaptation are not as drastic. The filter weights for the adaptive neurons see a small decrease in the weights associated with them as shown in Figure 8.5.

Compared to the loss of neurons scenario, the adaptation does not impact the loss of space being sampled (i.e. retains the same tuning widths), only the amplitudes of the responses that the neurons generate. The adaptive filter operates over each 50 ms time bin to change its weights to counter this small effect on the amplitudes brought about by adaptation and thus, is able to maintain its level of optimal error. Thus, the errors for the reoptimizing linear filter are the lowest among the adapting filters and the performance of the adaptive filter is not affected by the neuronal adaptation.

9 DISCUSSION AND CONCLUSION

This thesis presents an adaptive neural decoding system based on a Kalman filter that was designed to be resistant to the occurrence of nonstationary neural signals. Filter decoding performance was compared to a non-adaptive system and several alternative adaptive decoding algorithms (reoptimizing linear filter and reoptimizing Kalman filter), proposed in the literature.

The algorithm was implemented using simulated motor cortical neurons encoding intended movement velocity. The decoded movement and therefore the performance (NRMS errors) was described within the velocity space. Other approaches (Wu et al 2008) estimated movement from neuron responses as decoded velocity as well as position. If intended position were to be included in the simulations here, it would not impact the accuracy of velocity decoding as long as the number of neurons encoding for velocity is retained. The decoded velocity information could be used toward estimating intended position more accurately, potentially reducing errors in decoded position.

The white noise signals used for training the algorithms and for testing the decoding performance were bandlimited to approximate the range of limb movement frequencies (0 - 1.5Hz). In a real world scenario, the intended movement would likely not have a uniform power distributed along all frequencies. A single bandlimited signal was used here since it provides the most generalization across the space of possible movements. We have shown the proposed algorithm to be resistant to changes in the movement bandwidth (Chapter 4), therefore, its performance would be retained under real world conditions.

9.1 Decoding Beyond the Trained Movement Constraints

The adaptive filter performance was the best among all the filters implemented under the trained RMS power (of 1) and bandwidth constraints (0 – 1.5 Hz) of the optimizing signal. Also, although the performance of the adaptive filter was significantly different (t-test for bandwidth 0-5 Hz, $t(4) = -59.41$, $p<1E-6$) when the frequency range of decoded movements exceeded the optimized bandwidth, it represented the lowest decoding errors. For decoding movements beyond the adaptive filter's trained RMS power, decoding accuracy remained high. This performance was better than comparable adaptive algorithms such as the reoptimizing linear filter and reoptimizing Kalman filter (both described in Wu. et al 2008). The decoding accuracy for the adaptive Kalman filter was the highest for the RMS power variations of 0.5, 1, 2 and 5 in the test signal. For the BW changes, the adaptive Kalman filter had the lowest decoding errors for the test bandwidths of 0-1Hz, 0-1.5Hz, 0-2Hz and 0-5Hz. The reoptimizing linear filter had the best decoding accuracy for the test bandwidth of 0 -0.5Hz.

In order to achieve low decoding errors in the case of a non-adaptive system, it would be necessary to perform the initial optimization of the weights using all possible movements with a wide range of frequencies and amplitudes. This would result in a longer duration for the training sessions and greater inconvenience to the subjects. Even with sufficient training, as seen during our simulations, the static filter decoding accuracy may suffer when compared that of the adaptive filter since it is unable to optimize to the instantaneous stimulus properties such as current amplitude and frequency of the movement (velocity).

9.2 Nonstationary Conditions

The simulation results for the nonstationary conditions show that the adaptive decoding filter is capable of recovering from catastrophic changes in the neural signals to maintain accurate decoding of the intended movement. With some approaches, full recovery to events such as neuron replacement can require hours (Rotermund et al 2006). The time taken for recovery was 12 minutes for a 50% loss of neural signals and 3 minutes for full replacement of neural signals.

For catastrophic nonstationary changes such as loss of 50% of the neurons and replacement of 100% of the neuron population, the reoptimizing Kalman and the reoptimizing linear filters show better decoding accuracy and faster recovery than the proposed adaptive Kalman filter. Since these adaptive systems depend on minimizing error in their reoptimizing window (550 seconds), their rate of recovery for a large error change is better than the proposed adaptive filter. These adaptive approaches are better suited to catastrophic nonstationary effects such as loss and replacement of neurons since they are more sensitive to the large error that is produced.

However, for nonstationarities such as attention modulation and adaptation, the induced error at each timestep is small. The time scale of the induced changes (~ 5 seconds for attention and 50-600 ms for adaptation) allows the adaptive Kalman filter to modify its weights over each iteration to combat the nonstationary effects. This allows the gradient descent approach of the adaptive Kalman algorithm to make changes to the weights over each successive iteration and combat the increased error. The adaptive Kalman filter decoding is resistant to both these nonstationarities and no increase in error

is observed. For the reoptimizing Kalman filter and the reoptimizing linear filters, the timescale of these nonstationarities is smaller than their reoptimizing window of 550 seconds. Those approaches make a change to the weights to reduce the error over a 550 second window and therefore, they are not able to achieve optimal decoding. The proposed adaptive Kalman filter is thus better suited to combating nonstationarities of attention and adaptation of neurons.

Gage et al. (2005) have previously proposed an adaptive Kalman filtering approach that is similar to the reoptimizing Kalman filter approach outlined here. The key difference between the two approaches lies in the method for re-optimization (windowed vs. instantaneous) and the requirements on the type of error signal used by the system. In the adaptive Kalman filter developed by Gage and colleagues, the system is intermittently re-optimized using the standard least-square optimization over a sliding temporal window. The temporal history used in re-optimizing the system places a lower bound on the speed at which the system can recover by requiring that nonstationary changes in the signal move beyond the re-optimization window. However, it was observed in that study that the reoptimizing filters had error trends that did not conform to this idea (the reoptimizing Kalman filter recovered ~100 seconds for a loss of 50% of the population). The total error over the window that a reoptimizing filter tries to minimize determines the rate of change of its weights. Higher error results in faster changes and thus faster recovery to the minimum error. For catastrophic changes (neuron loss, replacement) that induce a high error into this window, a quicker recovery is therefore observed.

Point process adaptive filters (Eden et al 2004a, 2004b; Srinivasan et al 2007) have been shown to be resistant to slow changes in the neural response properties but it is

unclear how such systems would perform under more extreme conditions. For neuron replacement at a rate of one per minute, the adaptive filter proposed by Eden and colleagues (2005), was able to reconstruct movement direction from a population of 20 neurons but was not able to consistently recover speed of movement. Srinivasan et al (2007) showed similar trends in performance when neurons were replaced at a rate one per minute. When an equivalent rate of replacement was simulated here, there was no observable effect on performance using the adaptive filter proposed here (see Chapter 6).

Use of least-squares optimization for obtaining the decoding weights also requires that the error signals be explicitly represented in units that define the movement space. Such information is generally not available outside of a laboratory setting posing challenges for real-world implementation. Error information could likely be extracted from other cortical areas and neural populations, although the same issues inherent in decoding non-stationary signals would affect the decoded estimates of error.

The adaptive decoding algorithm described here uses a gradient descent scheme to update its weights. With this type of system it is possible to use more reliable "qualitative" measures of error (e.g., signed/direction of error, relative error, quantized 'levels' error) to guide weight changes along with a gain adjustment to optimize the speed of convergence based on the type of error information available. Thus, having an exact error signal is not an explicit requirement of the adaptive decoding algorithm described here. Future work will examine the ability of the system to adapt using more generalized error signals that do not explicitly encode error within the movement space.

9.3 Computational Requirements

In conjunction with good performance, the practical application of adaptive decoding systems will ultimately require their implementation in a portable system. Current adaptive algorithms have shown considerable promise for the reliable decoding of neural signals at the brain machine interface; however, they often have high computational demands (Rotermund et al 2006, Srinivasan et al, 2007) that may not be suited to a portable implementation.

For the initial optimization, the computational cost associated with the Kalman-based decoders is given by $\boldsymbol{O\,(N^3)}$, where N is the size of the matrices and O denotes order of the operation. The cost is due to the estimation of the decoding weights during the least squares optimization process using matrix sizes of

- NxN → (100 x 100), N is the number of neurons in the population
- NxNt → (100 x 5000), Nt is the number of 50 ms bins in the 250 second movement stimulus

Since N denotes the number of neurons in the population, the least squares optimization process yields a (N x N x N) size matrix multiplication operation that dominates the order O of operations. The number of steps required for these computations, would therefore, be dominated by a N^3 term. The big-O notation for these operations, by definition, would be given by $\boldsymbol{O\,(N^3)}$.

While the computational cost for the linear filter would also be given by $\boldsymbol{O\,(N^3)}$, since it requires 20 additional decoding weights, its cost is 20 times higher than a Kalman

based approach. This is not relevant in a computational system such as a desktop computer, but for a portable implementation with more limited computational resources, this could potentially impact real-time implementation. Since it reoptimizes at each timestep using an optimization technique over a 550 second time window, the filter carries a high computational cost during each operation (NxNtx20) → (100 x 11000 x 20) at each iteration (11000 neuron response bins of 50ms each in a 550 second window). Since the reoptimizing Kalman filter uses a window of length 550 seconds as well, the computational cost associated with it operating at each timestep is given by (NxNt) → (100 x 11000).

The adaptive Kalman filter proposed here requires information only from the previous timestep to obtain the current estimate. After its initial optimization, the computational cost per iteration is given by (N x N x N) for estimation of the corrected decoding weight (see eq (3.8) and (3.9)). Thus, the maximum cost for a hundred neuron population would be (100 x 100 x 100). Thus, it has lower computational requirements that make it amenable to a portable implementation with current technology.

9.4 Conclusion

The aim of the project was to identify the sources of nonstationarity associated with prostheses during the long term and create an algorithm that would combat any errors in decoding attributable to these sources. In addition, the alogirthm was compared to other approaches in literature in terms of decoding accuracy and recovery time.

The proposed adaptive filter was able to reliably decode movement outside the movement attributes such as movement range and speed that it was trained over. Its performance was better than comparable approaches and thus, the algorithm can be employed for decoding under non-stationary conditions without requiring frequent and cumbersome retraining.

For catastrophic nonstationary effects such as loss of 50% of the population and replacement of the entire population of sampled neurons, the filter recovery was slower and did not recover to an optimal error when compared to other proposed approaches such as the reoptimizing linear and Kalman filters. The catastrophic effects were simulated as a worst case. When the rate of the impact of the nonstationarity was lessened (for e.g. 1 neuron replaced per minute), the adaptive filter was able to retain its decoding performance and approached an optimal error within 50 ms of the impact.

The filter recovered its performance for nonstationary changes that are not as drastic, such as attention and adaptation and results were comparable to other approaches or better. The smaller timescale over which these nonstationarities occur allow the filter to recover to a lower error in its decoding.

This would suggest that in addition to a very good performance under stationary conditions, the adaptive filter would be able to combat slow replacement, attention and adaptation in a practical implementation. The filter was evaluated to meet certain design criteria to achieve such an implementation:

- Real-time performance

  As per the requirements, the algorithm was able to decode neuron response rates computed over 50 ms time bins and provide movement estimates over each bin.

- Accuracy

  The algorithm reached the specified accuracy levels of 10-20% while decoding movement stimuli using a stationary population of neurons. After nonstationary impact, the algorithm was able to recover to decode with better accuracy than comparable approaches for Attention (21.7%) and Adaptation (24.7%) while worse for Loss (18.9%) and Replacement (25%).

- Time to recovery

  The algorithm had a quicker time to recovery for nonstationarities such as Attention (130 seconds) and Adaptation (110 seconds), while for catastrophic nonstationarities such as Loss (550 seconds) and Replacement of neurons (980 seconds), the time to recovery was much slower than comparable approaches.

- Number of computations

  The computations required by the proposed algorithm for adaptive decoding resulted in a computational cost of (100 x 100 x 100) or $\boldsymbol{O\,(N^3)}$, which is less than comparable approaches such as the reoptimizing Kalman filter by a factor of 110 and the linear filter by a factor of 110x20 = 2200. This is more relevant in portable implementations due to limited computational power and thus the

proposed algorithm is amenable to a portable implantation than comparable approaches.

9.5 Future Directions

A future implementation of this algorithm would be realized in an embedded system producing the control signals for limb prostheses. Furture steps would include identifying the specifications of such as computational system, and creating a prototype implementation. Since the prototyping language used here in this case is MATLAB, implementing the algorithm in a faster compiled environment (C, embedded C) would lend itself well to a real-time portable implementation.

Also, the reliance on the absolute error signal used by the algorithm to adapt to the nonstationarity could be investigated. An error signal analogue that carries direction and not amplitude information could be potentially employed. The sources of movement error that can be tapped into in order to get the desired error signal could also be investigated.

BIBLIOGRAPHY

Amirikian B, Georgopoulos AP. Directional tuning profiles of motor cortical cells. Neurosci Res. 2000;36:73-79.

Biran, R., Martin, D.C., and Tresco, P.A. (2005). Neuronal cell loss accompanies the brain tissue response to chronically implanted silicon microelectrode arrays. Exp. Neurol. 195, 115–126.

Bjornsson CS, Smith KL, Lin G, Abdul-Karim MA, LeBlanc D, Turner JN, Roysam B, Shain W. Damage to the neurovascular unit during prosthetic device insertion: vascular casting and quantitative analysis, 35th Annual Meeting of Society for Neuroscience, Washington, DC, 2005

Connors BW, Gutnick MJ. Intrinsic firing patterns of diverse neocortical neurons. Trends Neurosci. 1990 Mar;13(3):99-104.

Chen Y, Martinez-Conde S, Macknik SL, Bereshpolova Y, Swadlow HA, Alonso JM. Task difficulty modulates the activity of specific neuronal populations in primary visual cortex. Nat Neurosci. 2008 Aug;11(8):974-82. Epub 2008.

Eden U, Truccolo W, Fellows M, Donoghue J, Brown E. Reconstruction of hand movement trajectories from a dynamic ensemble of spiking motor cortical neurons. Conf Proc IEEE Eng Med Biol Soc. 2004;6:4017-4020.

Eden UT, Frank LM, Barbieri R, Solo V, Brown EN. Dynamic analysis of neural encoding by point process adaptive filtering. Neural Comput. 2004 May;16(5):971-98

Eliasmith C, Anderson CH. Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems. The MIT Press; 2002.

Gage, G.J.; Otto, K.J.; Ludwig, K.A.; Kipke, D.R. , Co-adaptive Kalman filtering in a naive rat cortical control task, Engineering in Medicine and Biology Society, 2004. IEMBS '04. 26th Annual International Conference of the IEEE , vol.2, no., pp.4367-4370

Gage GJ, Ludwig KA, Otto KJ, Ionides EL, Kipke DR. Naive coadaptive cortical control. J Neural Eng. 2005;2:52-63.

Hochberg LR, Serruya MD, Friehs GM, et al. Neuronal ensemble control of prosthetic devices by a human with tetraplegia. Nature. 2006;442:164-171.

Issacs RE, Weber DJ, Schwartz AB, Work toward realtime control of a cortical neural prosthesis, IEEE Trans. On Rehab. Eng. 8, pp 196-198, 2000

Johansen-Berg H, Matthews PM, Attention to movement modulates activity in sensori-motor areas, including primary motor cortex. Exp Brain Res. 2002 Jan;142(1):13-24. Epub 2001 Nov 13.

Kalaska JF, Neuroscience: Brain control of a helping hand, Nature 453, 994-995 (19 June 2008)

Lebedev MA, Nicolelis MAL, Brain-machine interfaces: past, present and future, Trends in Neurosciences, Volume 29, Issue 9, September 2006, Pages 536-546

Liu YH, Wang XJ. Spike-frequency adaptation of a generalized leaky integrate-and-fire model neuron. J Comput Neurosci. 2001 Jan-Feb;10(1):25-45.

Maybeck PS. Stochastic Models, Estimation and Control. New York : Academic Press; 1979.

Maynard EM, Nordhausen CT, Normann RA, The Utah Intracortical Electrode Array: A recording structure for potential brain-computer interfaces, Electroencephalography and Clinical Neurophysiology, Volume 102, Issue 3, March 1997, Pages 228-239

McAdams CJ, Maunsell JH. Effects of attention on the reliability of individual neurons in monkey visual cortex. Neuron 1999;23:765–773.

Moran DW, Schwartz AB. Motor Cortical Activity During Drawing Movements: Population Representation During Spiral Tracing, J Neurophysiol. 1999 Nov;82(5):2693-704

Paninski, L., Fellows, M., Hatsopoulos, N., and Donoghue, J. P. (2001). Temporal tuning properties for hand position and velocity in motor cortical neurons, J. of Neurophysiology.

Polikov, V.S., Tresco, P.A., and Reichart, W.M. (2005). Response of brain tissue to chronically implanted neural electrodes. J. Neurosci. Methods 148, 1–18.

Quraishi S, Heider B, Siegel RM. Attentional modulation of receptive field structure in area 7a of the behaving monkey. Cereb Cortex. 2007 Aug;17(8):1841-57. Epub 2006 Oct 31.

Rotermund D, Ernst UA, Pawelzik KR. Towards on-line adaptation of neuro-prostheses with neuronal evaluation signals. Biol Cybern. 2006;95:243-257.

Schwartz AB. Cortical neural prosthetics. Annu Rev Neurosci. 2004;27:487-507.

Schwartz AB, Cui XT, Weber DJ, Moran DW. Brain-controlled interfaces: Movement restoration with neural prosthetics. Neuron. 2006;52:205-220.

Srinivasan L, Eden UT, Mitter SK, Brown EN. General-purpose filter design for neural prosthetic devices. J Neurophysiol. 2007;98:2456-2475.

Suner, S.; Fellows, M.R.; Vargas-Irwin, C.; Nakata, G.K.; Donoghue, J.P, Reliability of signals from a chronically implanted, silicon-based electrode array in non-human primate primary motor cortex, Neural Systems and Rehabilitation Engineering, IEEE Transactions on , vol.13, no.4, pp.524-541, Dec. 2005

Swindale NV. Orientation tuning curves: Empirical description and estimation of parameters. Biol Cybern. 1998;78:45-56.

Welch G, Bishop G. An Introduction to the Kalman Filter; 2006.

Wessberg J, Stambaugh CR, Kralik JD, Beck PD, Laubach M, Chapin JK, Kim J, Biggs SJ, Srinivasan MA, Nicolelis MA: Real-time prediction of hand trajectory by ensembles of cortical neurons in primates. Nature 2000, 408:361-365.

Wu W, Inferring hand motion from multi-cell recordings in motor cortex using a Kalman filter", presented at SAB'02-Workshop on Motor Control in Humans and Robots:

On the Interplay of Real Brains and Artificial Devices, Edinburgh, Scotland (UK), August 10, 2002.

Wu W, Gao Y, Bienenstock E, Donoghue JP, Black MJ. Bayesian population decoding of motor cortical activity using a Kalman filter. Neural Comput. 2006;18:80-118.

Wu W, Hatsopoulos, N.G, Real-Time Decoding of Nonstationary Neural Activity in Motor Cortex, Neural Systems and Rehabilitation Engineering, IEEE Transactions on , vol.16, no.3, pp.213-222, June 2008.

Xindong Liu; McCreery, D.B.; Carter, R.R.; Bullara, L.A.; Yuen, T.G.H.; Agnew, W.F.; , Stability of the interface between neural tissue and chronically implanted intracortical microelectrodes, Rehabilitation Engineering, IEEE Transactions on , vol.7, no.3, pp.315-326, Sep 1999.

Appendix A

MATLAB® code for the neuron model, decoding algorithms and simulations follows.

DecodingSimulation.m

```
close all; clear; clc
dbstop if error
tic

% Decoding simulation for the adaptive decoding filter

for NumberSim = 1:20
    close all; clear;

    [Sim, Stim] = InitializeNewSim;
    newSim = 1;
    validSim = 1;


    if validSim          %RUN THE SIMULATION

        %       Initialize Local Simulation Parameters
        rand('state',Sim.RSeed);                    %#ok<RAND> %Set
seed for random number generator
        nBins = Sim.FR.FiltLength/Sim.FR.tRateInt;  %Number of temporal
intervals comprising the linear filter


        % START SIMULATION

        for i = 1:length(Sim.nUnits)             %for each population
            for j = 1:Sim.nRuns
                t0 = clock;     %Initialize timer

                N = Sim.nUnits(i);
                Sim.Pop(i).nUnits = N;

                %Create neural population
                switch(Sim.phiEnc_func)
                    case 'GaussTuningResp'
                        if newSim
                            %Initialize Gaussian tuned neurons
                            if Sim.nDim == 1
                                Sim.Pop(i).Spref = (max(Stim.sRange)-
min(Stim.sRange))*rand(N,Sim.nDim) + min(Stim.sRange);
                                Sim.Pop(i).Ssig =
0.11*(2*rand(N,Sim.nDim)-1) + 0.16;    %for linear rep.
                                Sim.Pop(i).SmaxLin = max(Stim.sRange);
                            else
                                Sim.Pop(i).Spref = (max(Stim.sRange)-
min(Stim.sRange))*rand(N,Sim.nDim-1) + min(Stim.sRange);
                                Sim.Pop(i).Ssig =
0.34.*(2*rand(N,Sim.nDim-1)-1) + pi/4;    %for 2D polar rep.
                                Sim.Pop(i).Tau = Sim.Tau;
                                Sim.Pop(i).SmaxLin =
Stim.Training.maxMag;
```

```
                        end
                        [Sim.Pop(i).LIFparams, Sim.Pop(i).noiseVar,
Sim.Pop(i).maxResp] = InitGaussLIFNeurons(Stim.sRange, N,
Sim.Pop(i).Spref, Sim.Pop(i).Ssig, Sim.maxRespRange, ...
                            Sim.tauRefRange, Sim.tauRCRange,
Sim.V_th, Sim.R_leak, Sim.error, 1);
                    end

                case 'LinearTuningResp'
                    if newSim
                        %Initialize Linearly tuned neurons
                        Sim.Pop(i).Sint = (max(Stim.sRange)-
min(Stim.sRange))*rand(N,1) + min(Stim.sRange);  %Randomly place x-
intercepts across input range

                        [Sim.Pop(i).LIFparams, Sim.Pop(i).noiseVar,
Sim.Pop(i).maxResp] = InitLinearLIFNeurons(Sin, N, Sim.Pop(i).Sint,
Sim.maxRespRange, ...
                            Sim.tauRefRange, Sim.tauRCRange,
Sim.V_th, Sim.R_leak, Sim.error);
                    end

                    if Sim.nDim > 1
                        Sim.Pop(i).prefAngle = rand(1,N)*2*pi;
%Randomly select each neurons preferred direction (for multi-
dimensional stimulus representations)
                        phiEnc = [cos(Sim.Pop(i).prefAngle);
sin(Sim.Pop(i).prefAngle)];  %Compute normalized encoding weights based
on the preferred direction
                    else
                        phiEnc = ones(1,N);
%For the 1D case the preferred direction is +-1 cand is already
incorporated into the neuron's response.
                    end

                case 'CosineTuningResp'
                    if newSim
                        %Initialize Cosine tuned neurons
                        if Sim.nDim == 1
                            Sim.Pop(i).Spref = (max(Stim.sRange)-
min(Stim.sRange))*rand(N,Sim.nDim) + min(Stim.sRange);
                            Sim.Pop(i).SmaxLin = max(Stim.sRange);
                        else
                            Sim.Pop(i).Spref = (max(Stim.sRange)-
min(Stim.sRange))*rand(N,Sim.nDim-1) + min(Stim.sRange);
                            Sim.Pop(i).Tau = Sim.Tau;
                            Sim.Pop(i).SmaxLin =
Stim.Training.maxMag;
                        end
                        [Sim.Pop(i).LIFparams, Sim.Pop(i).noiseVar,
Sim.Pop(i).maxResp] = InitCosineLIFNeurons(Stim.sRange, N,
Sim.Pop(i).Spref, Sim.maxRespRange, ...
                            Sim.tauRefRange, Sim.tauRCRange,
Sim.V_th, Sim.R_leak, Sim.error, 1);
                    end
```

```
                    case 'vonMisesTuningResp'
                        if newSim
                            %Initialize von Mises tuned neurons
                            if Sim.nDim == 1
                                Sim.Pop(i).Spref = (max(Stim.sRange)-
min(Stim.sRange))*rand(N,Sim.nDim) + min(Stim.sRange);
                                Sim.Pop(i).SmaxLin = max(Stim.sRange);
                            else
                                Sim.Pop(i).Spref = (max(Stim.sRange)-
min(Stim.sRange))*rand(N,Sim.nDim-1) + min(Stim.sRange);
                                Sim.Pop(i).Spref =
round(Sim.Pop(i).Spref.*(180/pi)); % Round off the preferred direction
to the nearest degree
                                Sim.Pop(i).Spref =
Sim.Pop(i).Spref.*(pi/180);
                                Sim.Pop(i).SmaxLin =
Stim.Training.maxMag;
                            end
                            clear kappa
                            [Sim.Pop(i).LIFparams, Sim.Pop(i).noiseVar,
Sim.Pop(i).maxResp, a_S, kappa, halfwidth] =
InitvonMisesLIFNeurons(Stim.sRange, N, Sim.Pop(i).Spref,
Sim.maxRespRange, ...
                                Sim.tauRefRange, Sim.tauRCRange,
Sim.V_th, Sim.R_leak, Sim.error, 1);
                            Sim.Pop(i).kappa = kappa;
                            Sim.Pop(i).halfwidth = halfwidth;
                        end
                    otherwise
                        error('Invalid stimulus tuning profile
specified');
                end

                % GENERATE THE TRAINING SIGNAL
                tt = 0:Stim.FR.dt:Stim.Training.FR.T;

                Amps_training = zeros(Sim.nDim, length(tt));

                switch (Stim.Training.type)
                    case 'Constant'
                        Sin_training = Stim.Test.mag*ones(2,
length(tt));
                    case 'Figure 8'
                        theta = linspace(-pi/4, 3/4*pi, length(tt));
                        Sin_training = [1.5*cos(2*theta);
1*cos(2*theta).*sin(2*theta)]; %(April 12, 2007 - shifted center back
to (0,0))
                        Sin_training =
repmat(Sin_training,1,Stim.Test.FR.tst_runs);
                    case 'Circle'
                        thetaTemp = -
pi:Stim.Test.degreepert*(pi/180):pi;
                        Ntheta = length(thetaTemp);
                        theta = repmat(thetaTemp, 1,
floor(length(tt)/Ntheta));
```

```
                        theta = cat(2, theta,
thetaTemp(1:mod(length(tt),Ntheta)));
                        Sin_training = [Stim.Test.radius*cos(theta);
Stim.Test.radius*sin(theta)];
                    case 'White Noise'
                        for f = 1:Sim.nDim
                            [Sin_training(f,:),Amps_training(f,:)] =
genSignal(Stim.Training.FR.T,Stim.FR.dt,Stim.Training.rms,Stim.Training
.bandwidth,Sim.RSeed*pi*f);  %#ok<AGROW> %Increment random seed in
deteministic way across multiple dimensions when RandomSeed >0
                            %pi multiple in randomSeed used to ensure
                            %different amplitude coeff in generaiton of
random training and test signals
                        end
                end

                Sin_mag_training = sqrt(sum(Sin_training.^2,1));
                Ind95pctrain = ceil(0.95*size(Sin_mag_training, 2));
                Sin_mag_training_ascend = sort(Sin_mag_training,
'ascend');
                Sim.Pop(i).SmaxLin =
Sin_mag_training_ascend(Ind95pctrain);

                nRateStepsT =
floor(Stim.Training.FR.T/Sim.FR.tRateInt);
                ndtperBin = Sim.FR.tRateInt/Sim.FR.dt;
                LIFinit_training.V = zeros(1,Sim.nUnits);
                LIFinit_training.EndRefPeriod = zeros(1,Sim.nUnits);
                LIFinit_training.jitterSig = [];

                SUrateResp_training=zeros(Sim.nUnits,nRateStepsT);
                sSUCenters_training=zeros(2,nRateStepsT);


                t_A = 0:Sim.FR.tRateInt:Stim.Training.FR.T;
                if strcmp(Sim.Nonstatdecision, 'Yes') &&
(strcmp(Sim.NonStatType, 'Attention') || strcmp(Sim.NonStatType,
'AttentionReplacement'))
                    AttnSig = sin(2*pi*(1/Sim.AttnPeriod)*t_A);
                    AttnSig = (AttnSig + abs(min(AttnSig)));
                    AttnSig = AttnSig./max(AttnSig);
                    AttnSig = Sim.AttentionMod(1) +
(Sim.AttentionMod(2)-Sim.AttentionMod(1)).*AttnSig;
                else
                    AttnSig = ones(1, length(t_A));
                end

                Gadapt_training = Sim.Gadapt;

                for cnt=1:nRateStepsT
                    [SUrateResp_training(:,cnt),
sSUCenters_training(:,cnt), LIFinit_training, Gadapt_training,
spikeTimes_training, GadaptTemp_training] =
GetNeuronFiringRatesIterative_G(Sim, Stim, Sin_training(:,((cnt-
```

```
1)*ndtperBin)+1:cnt*(ndtperBin)), LIFinit_training, Sim.nUnits,
AttnSig(cnt), Gadapt_training);
                end

                % PLOT TRAINING STIMULUS
                figure, plot (Sin_training(1,:), Sin_training(2,:),
'LineWidth', 2);
                set(gca, 'FontSize', 14), legend('Training stimulus')
                title ('Training Stimulus', 'FontSize', 16);
                xlabel('X velocity V_x', 'FontSize', 14)
                ylabel('Y velocity V_y', 'FontSize', 14)
                drawnow;

                % OPTIMIZATION PROCESS
                if strcmp(Sim.Nonstatdecision, 'Yes') &&
(strcmp(Sim.NonStatType, 'Replacement') || strcmp(Sim.NonStatType,
'AttentionReplacement'))
                    Nusable = N - Sim.nchangedpop; % NO OF UNCHANGED
UNITS IN THE POPULATION
                else
                    Nusable = N;
                end

                [AdaptiveFilter.static.Asu, AdaptiveFilter.static.Hsu,
AdaptiveFilter.static.Wsu, AdaptiveFilter.static.Qsu] =
GetDecodingWeights(sSUCenters_training,
SUrateResp_training(1:Nusable,:));

                % Generate a new bandlimited white noise stimulus for
TESTING
                t = 0:Stim.FR.dt:Stim.Test.FR.T;    %Time at each
sample

                Amps = zeros(Sim.nDim, length(t));

                switch (Stim.Test.type)
                    case 'Constant'
                        Sin_tst = Stim.Test.mag*ones(2, length(t));
                    case 'Figure 8'
                        theta = linspace(-pi/4, 3/4*pi, length(t));
                        Sin_tst = [1.5*cos(2*theta);
1*cos(2*theta).*sin(2*theta)];
                        Sin_tst =
repmat(Sin_tst,1,Stim.Test.FR.tst_runs);
                    case 'Circle'
                        thetaTemp = -pi:Stim.Test.degreepert:pi;
                        Ntheta = length(thetaTemp);
                        theta = repmat(thetaTemp, 1,
floor(length(t)/Ntheta));
                        theta = cat(2, theta,
thetaTemp(1:mod(length(t),Ntheta)));
                        Sin_tst = [Stim.Test.radius*cos(theta);
Stim.Test.radius*sin(theta)];
                    case 'White Noise'
                        for f = 1:Sim.nDim
```

```
                            [Sin_tst(f,:),Amps(f,:)] =
genSignal(Stim.Test.FR.T,Stim.FR.dt,Stim.Test.rms,Stim.Test.bandwidth,S
im.RSeed*pi*f);  %Increment random seed in deteministic way across
multiple dimensions when RandomSeed >0
                            %pi multiple in randomSeed used to ensure
different amplitude coeff in generaiton of random training and test
signals
                            clear Amps
                        end
                end

                % PLOT TEST STIMULUS
                figure, plot (Sin_tst(1,:), Sin_tst(2,:),'r',
'LineWidth', 2);
                title ('Test Stimulus', 'FontSize', 16);
                set(gca, 'FontSize', 14), legend('Test like stimulus')
                xlabel('X velocity V_x', 'FontSize', 14)
                ylabel('Y velocity V_y', 'FontSize', 14)
                drawnow;

                % GENERATE A OPTIMIZING SIGNAL WITH PROPERTIES SIMILAR
TO THE TEST
                tt = 0:Stim.FR.dt:Stim.Training.FR.T;

                Amps_tst_like = zeros(Sim.nDim, length(tt));

                switch (Stim.Test.type)
                    case 'Constant'
                        Sin_tst_like = Stim.Test.mag*ones(2,
length(tt));
                    case 'Figure 8'
                        theta = linspace(-pi/4, 3/4*pi, length(tt));
                        Sin_tst_like = [1.5*cos(2*theta);
1*cos(2*theta).*sin(2*theta)];
                        Sin_tst_like =
repmat(Sin_tst_like,1,Stim.Test.FR.tst_runs);
                    case 'Circle'
                        thetaTemp = -
pi:Stim.Test.degreepert*(pi/180):pi;
                        Ntheta = length(thetaTemp);
                        theta = repmat(thetaTemp, 1,
floor(length(tt)/Ntheta));
                        theta = cat(2, theta,
thetaTemp(1:mod(length(tt),Ntheta)));
                        Sin_tst_like = [Stim.Test.radius*cos(theta);
Stim.Test.radius*sin(theta)];
                    case 'White Noise'
                        for f = 1:Sim.nDim
                            [Sin_tst_like(f,:),Amps_tst_like(f,:)] =
genSignal(Stim.Training.FR.T,Stim.FR.dt,Stim.Test.rms,Stim.Test.bandwid
th,Sim.RSeed*pi*f);  %Increment random seed in deteministic way across
multiple dimensions when RandomSeed >0
                            %pi multiple in randomSeed used to ensure
different amplitude coeff in generaiton of random training and test
signals
                            clear Amps_tst_like
```

```
                    end
                end

                nRateStepsT =
floor(Stim.Training.FR.T/Sim.FR.tRateInt);
                ndtperBin = Sim.FR.tRateInt/Sim.FR.dt;
                LIFinit_tst_like.V = zeros(1,Sim.nUnits);
                LIFinit_tst_like.EndRefPeriod = zeros(1,Sim.nUnits);
                LIFinit_tst_like.jitterSig = [];
                SUrateResp_tst_like=zeros(Sim.nUnits,nRateStepsT);
                sSUCenters_tst_like=zeros(2,nRateStepsT);
                Radapt = Sim.Radapt;

                t_A = 0:Sim.FR.tRateInt:Stim.Training.FR.T;
                if strcmp(Sim.Nonstatdecision, 'Yes') &&
(strcmp(Sim.NonStatType, 'Attention') || strcmp(Sim.NonStatType,
'AttentionReplacement'))
                    AttnSig = sin(2*pi*(1/Sim.AttnPeriod)*t_A);
                    AttnSig = (AttnSig + abs(min(AttnSig)));
                    AttnSig = AttnSig./max(AttnSig);
                    AttnSig = Sim.AttentionMod(1) +
(Sim.AttentionMod(2)-Sim.AttentionMod(1)).*AttnSig;
                else
                    AttnSig = ones(1, length(t_A));
                end

                Gadapt_tst_like = Sim.Gadapt;

                for cnt=1:nRateStepsT
                    [SUrateResp_tst_like(:,cnt),
sSUCenters_tst_like(:,cnt), LIFinit_tst_like, Gadapt_tst_like,
spikeTimes_tst_like, GadaptTemp_tst_like] =
GetNeuronFiringRatesIterative_G(Sim, Stim, Sin_tst_like(:,((cnt-
1)*ndtperBin)+1:cnt*(ndtperBin)), LIFinit_tst_like, Sim.nUnits,
AttnSig(cnt), Gadapt_tst_like);
                end

                if strcmp(Sim.Nonstatdecision, 'Yes')
                    if strcmp(Sim.NonStatType, 'Loss')
                        [AsuTestLikeSig, HsuTestLikeSig,
WsuTestLikeSig, QsuTestLikeSig] =
GetDecodingWeights(sSUCenters_tst_like,
SUrateResp_tst_like(1:(Sim.nUnits-Sim.nchangedpop),:));
                    elseif strcmp(Sim.NonStatType, 'Replacement') ||
strcmp(Sim.NonStatType, 'AttentionReplacement')
                        [AsuTestLikeSig, HsuTestLikeSig,
WsuTestLikeSig, QsuTestLikeSig] =
GetDecodingWeights(sSUCenters_tst_like,
SUrateResp_tst_like(Sim.nchangedpop+1:Sim.nUnits,:));
                    else
                        [AsuTestLikeSig, HsuTestLikeSig,
WsuTestLikeSig, QsuTestLikeSig] =
GetDecodingWeights(sSUCenters_tst_like, SUrateResp_tst_like);
                    end
                else
```

```
                    [AsuTestLikeSig, HsuTestLikeSig, WsuTestLikeSig,
QsuTestLikeSig] = GetDecodingWeights(sSUCenters_tst_like,
SUrateResp_tst_like);
                end

                % PLOT TEST LIKE STIMULUS FOR TRAINING
                figure, plot (Sin_tst_like(1,:), Sin_tst_like(2,:),'k',
'LineWidth', 2);
                title ('Test Like Stimulus', 'FontSize', 16);
                set(gca, 'FontSize', 14), legend('Test like stimulus')
                xlabel('X velocity V_x', 'FontSize', 14)
                ylabel('Y velocity V_y', 'FontSize', 14)
                drawnow;

                nRateSteps =
floor(Stim.Test.FR.tst_runs*Stim.Test.FR.T/Sim.FR.tRateInt);

                ndtperBin = Sim.FR.tRateInt/Sim.FR.dt;

                LIFinit.V = zeros(1,N);
                LIFinit.EndRefPeriod = zeros(1,N);
                LIFinit.jitterSig = [];

                LIFinitcat.V = [];
                LIFinitcat.EndRefPeriod = [];
                LIFinitcat.jitterSig = [];

                clear SUrateResp;
                SUrateResp=zeros(N,nRateSteps);
                SUrateRespTemp=zeros(N,nRateSteps);
                sSUCenters=zeros(Sim.nDim,nRateSteps);

                % STATIC FILTER INITIALIZATIONS
                sx = repmat(struct('A', 0, 'B', 0, 'H',
zeros(size(AdaptiveFilter.static.Hsu(:,1))), 'Q', 0, 'R',
zeros(size(AdaptiveFilter.static.Qsu)), 'P', 0, 'u', 0), 1,
nRateSteps);
                sy = repmat(struct('A', 0, 'B', 0, 'H',
zeros(size(AdaptiveFilter.static.Hsu(:,2))), 'Q', 0, 'R',
zeros(size(AdaptiveFilter.static.Qsu)), 'P', 0, 'u', 0), 1,
nRateSteps);

                sx(1).A = AdaptiveFilter.static.Asu(1,1);
                sx(1).B = 0;
                sx(1).H = AdaptiveFilter.static.Hsu(:,1);
                sx(1).Q = AdaptiveFilter.static.Wsu(1,1);
                sx(1).R = AdaptiveFilter.static.Qsu;

                sx(1).P =(sx(1).H\sx(1).R)/sx(1).H'; %P =
inv(H)*R*inv(H')
                sx(1).u = 0;
                sxscale = 1;

                sy(1) = [];
                sy(1).A = AdaptiveFilter.static.Asu(2,2);
```

```
                sy(1).B = 0;
                sy(1).H = AdaptiveFilter.static.Hsu(:,2);
                sy(1).Q = AdaptiveFilter.static.Wsu(2,2);
                sy(1).R = AdaptiveFilter.static.Qsu;

                sy(1).P = (sy(1).H\sy(1).R)/sy(1).H'; %P =
inv(H)*R*inv(H')
                sy(1).u = 0;
                syscale = 1;

                % STATIC FILTER INITIALIZATIONS FOR THE REMAINING
POPULATION OF
                % NEURONS WITH TEST LIKE TRAINING SIGNAL

                sxTstLk = repmat(struct('A', 0, 'B', 0, 'H',
zeros(size(HsuTestLikeSig(:,1))), 'Q', 0, 'R',
zeros(size(QsuTestLikeSig)), 'P', 0, 'u', 0), 1, nRateSteps);
                syTstLk = repmat(struct('A', 0, 'B', 0, 'H',
zeros(size(HsuTestLikeSig(:,2))), 'Q', 0, 'R',
zeros(size(QsuTestLikeSig)), 'P', 0, 'u', 0), 1, nRateSteps);

                sxTstLk(1).A = AsuTestLikeSig(1,1);
                sxTstLk(1).B = 0;
                sxTstLk(1).H = HsuTestLikeSig(:,1);
                sxTstLk(1).Q = WsuTestLikeSig(1,1);
                sxTstLk(1).R = QsuTestLikeSig;
                sxTstLk(1).P
=(sxTstLk(1).H\sxTstLk(1).R)/sxTstLk(1).H'; %P = inv(H)*R*inv(H')
                sxTstLk(1).u = 0;
                sxTstLkscale = 1;

                syTstLk(1).A = AsuTestLikeSig(2,2);
                syTstLk(1).B = 0;
                syTstLk(1).H = HsuTestLikeSig(:,2);
                syTstLk(1).Q = WsuTestLikeSig(2,2);
                syTstLk(1).R = QsuTestLikeSig;
                syTstLk(1).P =
(syTstLk(1).H\syTstLk(1).R)/syTstLk(1).H'; %P = inv(H)*R*inv(H')
                syTstLk(1).u = 0;
                syTstLkscale = 1;


                % ADAPTIVE FILTER INITIALIZATIONS
                clear statex;
                clear statey;

                [AdaptiveFilter adaptiveKalman]=
InitAdaptiveFilter(AdaptiveFilter);

                AdaptiveFilter.Hsaveoff =
zeros(size(AdaptiveFilter.static.Hsu,1),Sim.nDim*nRateSteps);

                flag = 0;
                errstep = 1;
                step = 1;
```

```
                xscalecat = [];
                yscalecat = [];
                changedpopcat = [];

                Ksx = [];
                Ksy = [];
                Kallcat = [];
                K2xycat = [];
                statex = zeros(1,nRateSteps);
                statey = zeros(1,nRateSteps);
                timecount = zeros(1,nRateSteps);
                normx = zeros(1,nRateSteps);
                normy = zeros(1,nRateSteps);
                times = zeros(1,nRateSteps);

                if strcmp(Sim.NonStatType, 'Loss')
                    indchangedpopLoss = Nusable -
Sim.neuronsEachTime+1;
                    indchangedpopReplace = zeros(1, Nusable);
                end
                if strcmp(Sim.NonStatType, 'Replacement') ||
strcmp(Sim.NonStatType, 'AttentionReplacement')
                    indchangedpopReplace =
Nusable+1:Sim.neuronsEachTime:N;
                    indchangedpopLoss = zeros(1, Nusable);
                end

                indchangedpopNusable = 1:Sim.neuronsEachTime:Nusable;
                replaceIndex = 0;

                errorx = zeros(1,
(AdaptiveFilter.errorwindow/Sim.FR.tRateInt)+1);
                errory = zeros(1,
(AdaptiveFilter.errorwindow/Sim.FR.tRateInt)+1);
                errorstatx = zeros(1,
(AdaptiveFilter.errorwindow/Sim.FR.tRateInt)+1);
                errorstaty = zeros(1,
(AdaptiveFilter.errorwindow/Sim.FR.tRateInt)+1);
                errorxTstLk = zeros(1,
(AdaptiveFilter.errorwindow/Sim.FR.tRateInt)+1);
                erroryTstLk = zeros(1,
(AdaptiveFilter.errorwindow/Sim.FR.tRateInt)+1);

                rmserrx = zeros(1,
nRateSteps/(AdaptiveFilter.errorwindow/Sim.FR.tRateInt));
                rmserry = zeros(1,
nRateSteps/(AdaptiveFilter.errorwindow/Sim.FR.tRateInt));
                rmserrstatx = zeros(1,
nRateSteps/(AdaptiveFilter.errorwindow/Sim.FR.tRateInt));
                rmserrstaty = zeros(1,
nRateSteps/(AdaptiveFilter.errorwindow/Sim.FR.tRateInt));
                rmserrxTstLk = zeros(1,
nRateSteps/(AdaptiveFilter.errorwindow/Sim.FR.tRateInt));
                rmserryTstLk = zeros(1,
nRateSteps/(AdaptiveFilter.errorwindow/Sim.FR.tRateInt));
```

```
                % TAKE A SNAPSHOT OF PERFORMANCE AT THE BEGINNING
                % ADAPTIVE
                [snapStartx, snapStarty] = Kalmansnapshot('Figure 8',
Sim, Stim, Nusable, [adaptiveKalman.adaptfilt3x.H
adaptiveKalman.adaptfilt3y.H], AdaptiveFilter,
adaptiveKalman.adaptfilt1x.P, [adaptiveKalman.adaptfilt1x.R
adaptiveKalman.adaptfilt1y.R], flag, 0, 0, 0, 1);
                Snapshotfigs(1)= gcf;
                % STATIC
                [snapstatStartx, snapstatStarty] =
Kalmansnapshot('Figure 8', Sim, Stim, Nusable,
AdaptiveFilter.static.Hsu, AdaptiveFilter, sx(1).P,
[AdaptiveFilter.static.Qsu AdaptiveFilter.static.Qsu], flag, 0, 0, 0,
1);
                Snapshotfigs(2)= gcf;

                t_A = 0:Sim.FR.tRateInt:Stim.Test.FR.T;
                if strcmp(Sim.Nonstatdecision, 'Yes') &&
(strcmp(Sim.NonStatType, 'Attention') || strcmp(Sim.NonStatType,
'AttentionReplacement'))
                    AttnSig = sin(2*pi*(1/Sim.AttnPeriod)*t_A);
                    AttnSig = (AttnSig + abs(min(AttnSig)));
                    AttnSig = AttnSig./max(AttnSig);
                    AttnSig = Sim.AttentionMod(1) +
(Sim.AttentionMod(2)-Sim.AttentionMod(1)).*AttnSig;
                else AttnSig = ones(1, length(t_A));
                end

                Gadapt = Sim.Gadapt;

                for cnt=1:nRateSteps
                    [SUrateResp(:,cnt), sSUCenters(:,cnt), LIFinit,
Gadapt, spikeTimes, GadaptTemp] = GetNeuronFiringRatesIterative_G(Sim,
Stim, Sin_tst(:,((cnt-1)*ndtperBin)+1:cnt*(ndtperBin)), LIFinit,
Sim.nUnits, AttnSig(cnt), Gadapt);
                    SUrateRespTemp(:,cnt) = SUrateResp(:,cnt);
                end

                for cnt=1:nRateSteps
                    % INTRODUCTION OF NONSTATIONARITY
                    if strcmp(Sim.Nonstatdecision, 'Yes')
                        if sum(cnt == Sim.NonStatTime/Sim.FR.tRateInt)
                            replaceIndex = replaceIndex + 1;
                            if strcmp(Sim.NonStatType, 'Replacement')
|| strcmp(Sim.NonStatType, 'AttentionReplacement')
                                if Sim.nchangedpop ==
Sim.neuronsEachTime
                                    SUrateResp(1:Sim.neuronsEachTime ,
cnt:end) = SUrateResp(Sim.neuronsEachTime+1:end, cnt:end);
                                else

SUrateResp(indchangedpopNusable(replaceIndex), cnt:end) =
SUrateResp(indchangedpopReplace(replaceIndex), cnt:end);
                                end
                                flag = 2;
```

```
                            elseif strcmp(Sim.NonStatType, 'Loss')
                                flag = 1;
                                SUrateResp(Nusable:-
1:indchangedpopLoss,cnt:end) = 0;
                                Nleft = indchangedpopLoss - 1;
                            end
                            if sum(size(Sim.NonStatTime)) ~=2
                                indchangedpopLoss = indchangedpopLoss -
Sim.neuronsEachTime;
                            end
                        end
                        % TAKE A SNAPSHOT OF PERFORMANCE AT THE
INTRODUCTION OF NONSTATIONARITY
                        if strcmp(Sim.NonStatType, 'Replacement') ||
strcmp(Sim.NonStatType, 'Loss') || strcmp(Sim.NonStatType,
'AttentionReplacement')
                            if cnt ==
(Sim.NonStatTime(end))/Sim.FR.tRateInt
                                % ADAPTIVE
                                [snapNonstatx, snapNonstaty] =
Kalmansnapshot('Figure 8', Sim, Stim, Nusable,
[adaptiveKalman.adaptfilt3x.H adaptiveKalman.adaptfilt3y.H],
AdaptiveFilter, adaptiveKalman.adaptfilt1x.P,
[adaptiveKalman.adaptfilt1x.R adaptiveKalman.adaptfilt1y.R], flag,
indchangedpopNusable(replaceIndex), indchangedpopReplace(replaceIndex),
indchangedpopLoss, cnt);
                                Snapshotfigs(3)= gcf;
                                % STATIC
                                [snapstatNonstatx, snapstatNonstaty] =
Kalmansnapshot('Figure 8', Sim, Stim, Nusable,
AdaptiveFilter.static.Hsu, AdaptiveFilter, sx(cnt).P,
[AdaptiveFilter.static.Qsu AdaptiveFilter.static.Qsu], flag,
indchangedpopNusable(replaceIndex), indchangedpopReplace(replaceIndex),
indchangedpopLoss, cnt);
                                Snapshotfigs(4)= gcf;
                            end
                        end
                    end
                end

                %                 save LongLongSim

                for cnt=1:nRateSteps
                    % STATIC FILTER
                    sx(cnt).z = SUrateResp(1:Nusable,cnt);
                    sy(cnt).z = SUrateResp(1:Nusable,cnt);
                    if(cnt == 1) % PROVIDE INITIAL BEST ESTIMATES FOR
THE KALMAN FILTER
                        sx(1).x = sx(1).H\sx(1).z;
                        sy(1).x = sy(1).H\sx(1).z;
                    end
                    [sx(cnt+1), Kx] = kalmanf(sx(cnt),sxscale);
                    [sy(cnt+1), Ky] = kalmanf(sy(cnt),syscale);

                    % STATIC FILTER - Test for optimal pop
                    if strcmp(Sim.Nonstatdecision, 'Yes')
```

```
                        if strcmp(Sim.NonStatType, 'Loss')
                            sxTstLk(cnt).z = SUrateResp(1:(Sim.nUnits-
Sim.nchangedpop),cnt);
                            syTstLk(cnt).z = SUrateResp(1:(Sim.nUnits-
Sim.nchangedpop),cnt);
                        elseif strcmp(Sim.NonStatType, 'Replacement')
|| strcmp(Sim.NonStatType, 'AttentionReplacement')
                            sxTstLk(cnt).z =
SUrateResp(Sim.nchangedpop+1:Sim.nUnits,cnt);
                            syTstLk(cnt).z =
SUrateResp(Sim.nchangedpop+1:Sim.nUnits,cnt);
                        else
                            sxTstLk(cnt).z = SUrateResp(1:Nusable,cnt);
                            syTstLk(cnt).z = SUrateResp(1:Nusable,cnt);
                        end
                    else
                        sxTstLk(cnt).z = SUrateResp(1:Nusable,cnt);
                        syTstLk(cnt).z = SUrateResp(1:Nusable,cnt);
                    end

                    if(cnt == 1) % PROVIDE INITIAL BEST ESTIMATES FOR
THE KALMAN FILTER
                        sxTstLk(1).x = sxTstLk(1).H\sxTstLk(1).z;
                        syTstLk(1).x = syTstLk(1).H\syTstLk(1).z;
                    end
                    [sxTstLk(cnt+1), KxT] =
kalmanf(sxTstLk(cnt),sxTstLkscale);
                    [syTstLk(cnt+1), KyT] =
kalmanf(syTstLk(cnt),syTstLkscale);

                    % ADAPTIVE FILTER
                    if (cnt == 1) % PROVIDE INITIAL BEST ESTIMATES FOR
THE KALMAN FILTER
                        adaptiveKalman.adaptfilt1x.x =
adaptiveKalman.adaptfilt1x.H\SUrateResp(1:Nusable,1);
                        adaptiveKalman.adaptfilt1y.x =
adaptiveKalman.adaptfilt1y.H\SUrateResp(1:Nusable,1);
                    end

                    [adaptiveKalman, AdaptiveFilter, statex(cnt),
statey(cnt), Kall, K2xy] =
adaptKalmanIterate(adaptiveKalman,AdaptiveFilter,sSUCenters(:,cnt),SUra
teResp(1:Nusable,cnt),cnt);
                    AdaptiveFilter.Hsaveoff(:,2*cnt-1:2*cnt) =
[adaptiveKalman.adaptfilt3x.H adaptiveKalman.adaptfilt3y.H];
                    xscalecat = cat(2,xscalecat,AdaptiveFilter.xscale);
                    yscalecat = cat(2,yscalecat,AdaptiveFilter.yscale);

                    % ERROR CALCULATIONS - relative errors
                    % Normalized Mean Square Errors
                    errorx(step) = (sSUCenters(1,cnt) - statex(cnt));
                    errory(step) = (sSUCenters(2,cnt) - statey(cnt));
                    errorstatx(step) = (sSUCenters(1,cnt) - sx(cnt).x);
                    errorstaty(step) = (sSUCenters(2,cnt) - sy(cnt).x);
                    errorxTstLk(step) = (sSUCenters(1,cnt) -
sxTstLk(cnt).x);
```

```
                    erroryTstLk(step) = (sSUCenters(2,cnt) -
syTstLk(cnt).x);

                    if
mod(cnt,(AdaptiveFilter.errorwindow/Sim.FR.tRateInt)) == 0 % CALCULATE
THE NRMS ERROR FOR EACH ERROR WINDOW
                        rmserrx(errstep) = sqrt(mean((errorx).^2));
                        rmserry(errstep) = sqrt(mean((errory).^2));

                        rmserrstatx(errstep) =
sqrt(mean((errorstatx).^2));
                        rmserrstaty(errstep) =
sqrt(mean((errorstaty).^2));

                        rmserrxTstLk(errstep) =
sqrt(mean((errorxTstLk).^2));
                        rmserryTstLk(errstep) =
sqrt(mean((erroryTstLk).^2));

                        errstep = errstep+1;
                        step = 1;
                    end
                    if
mod(cnt,((AdaptiveFilter.errorwindow*10)/Sim.FR.tRateInt)) == 0
                        sprintf('%d of %d seconds done !!! (Simulation
Time)', round(cnt*Sim.FR.tRateInt),
round(Stim.Test.FR.T*Stim.Test.FR.tst_runs))
                        sprintf('Sim running for %d seconds !!! (Real
Time)', round(toc))
                    end
                    step = step +1;
                end

                % TAKE A SNAPSHOT OF PERFORMANCE AT THE END

                if strcmp(Sim.Nonstatdecision, 'Yes') &&
strcmp(Sim.NonStatType, 'Loss')
                    % ADAPTIVE
                    [snapEndx, snapEndy] = Kalmansnapshot('Figure 8',
Sim, Stim, Nusable, [adaptiveKalman.adaptfilt3x.H
adaptiveKalman.adaptfilt3y.H], AdaptiveFilter,
adaptiveKalman.adaptfilt1x.P, [adaptiveKalman.adaptfilt1x.R
adaptiveKalman.adaptfilt1y.R], flag,
indchangedpopNusable(replaceIndex), indchangedpopReplace(replaceIndex),
indchangedpopLoss, cnt);
                    Snapshotfigs(5)= gcf;
                    % STATIC
                    [snapstatEndx, snapstatEndy] =
Kalmansnapshot('Figure 8', Sim, Stim, Nusable,
AdaptiveFilter.static.Hsu, AdaptiveFilter, sx(cnt).P,
[AdaptiveFilter.static.Qsu AdaptiveFilter.static.Qsu], flag,
indchangedpopNusable(replaceIndex), indchangedpopReplace(replaceIndex),
indchangedpopLoss, cnt);
                    Snapshotfigs(6)= gcf;
                else
                    % ADAPTIVE
```

```
                    [snapEndx, snapEndy] = Kalmansnapshot('Figure 8',
Sim, Stim, Nusable, [adaptiveKalman.adaptfilt3x.H
adaptiveKalman.adaptfilt3y.H], AdaptiveFilter,
adaptiveKalman.adaptfilt1x.P, [adaptiveKalman.adaptfilt1x.R
adaptiveKalman.adaptfilt1y.R], flag, 0, 0, 0, cnt);
                    Snapshotfigs(5)= gcf;
                    % STATIC
                    [snapstatEndx, snapstatEndy] =
Kalmansnapshot('Figure 8', Sim, Stim, Nusable,
AdaptiveFilter.static.Hsu, AdaptiveFilter, sx(cnt).P,
[AdaptiveFilter.static.Qsu AdaptiveFilter.static.Qsu], flag, 0, 0, 0,
cnt);
                    Snapshotfigs(6)= gcf;
                end

                AdaptiveFilter.statex = statex;
                AdaptiveFilter.statey = statey;
                AdaptiveFilter.Kall = Kall;

                sxplot = zeros(1,nRateSteps-1);
                syplot = zeros(1,nRateSteps-1);
                sxTstLkplot = zeros(1,nRateSteps-1);
                syTstLkplot = zeros(1,nRateSteps-1);

                for cnt=1:nRateSteps % for extracting the array from
the struct
                    sxplot(cnt)=sx(cnt).x;
                    syplot(cnt)=sy(cnt).x;
                    sxTstLkplot(cnt)=sxTstLk(cnt).x;
                    syTstLkplot(cnt)=syTstLk(cnt).x;
                end

                AdaptiveFilter.sxplot = sxplot;
                AdaptiveFilter.syplot = syplot;
                AdaptiveFilter.sxTstLkplot = sxTstLkplot;
                AdaptiveFilter.syTstLkplot = syTstLkplot;

                %Scale the errors with the RMS power of the TEST signal

                AdaptiveFilter.nrmserrx =
rmserrx./sqrt(mean((sSUCenters(1,:)).^2));
                AdaptiveFilter.nrmserry =
rmserry./sqrt(mean((sSUCenters(2,:)).^2));
                AdaptiveFilter.nrmserrstatx =
rmserrstatx./sqrt(mean((sSUCenters(1,:)).^2));
                AdaptiveFilter.nrmserrstaty =
rmserrstaty./sqrt(mean((sSUCenters(2,:)).^2));
                AdaptiveFilter.nrmserrxTstLk =
rmserrxTstLk./sqrt(mean((sSUCenters(1,:)).^2));
                AdaptiveFilter.nrmserryTstLk =
rmserryTstLk./sqrt(mean((sSUCenters(2,:)).^2));
                AdaptiveFilter.nrmsError = sqrt((rmserrx).^2 +
(rmserry).^2)./sqrt(mean(sSUCenters(1,:).^2 + sSUCenters(2,:).^2));
                AdaptiveFilter.nrmsErrorStat = sqrt((rmserrstatx).^2 +
(rmserrstaty).^2)./sqrt(mean(sSUCenters(1,:).^2 + sSUCenters(2,:).^2));
```

```
                AdaptiveFilter.nrmsErrorTstLk = sqrt((rmserrxTstLk).^2
+ (rmserryTstLk).^2)./sqrt(mean(sSUCenters(1,:).^2 +
sSUCenters(2,:).^2));

                % CONSTRUCT ALL THE PLOTS
                figure
                hold on
                grid on
                plot
(sSUCenters(1,1:Stim.Test.FR.T/Sim.FR.tRateInt),sSUCenters(2,1:Stim.Tes
t.FR.T/Sim.FR.tRateInt),'r-', 'LineWidth', 2);
                plot (sxplot(1:Stim.Test.FR.T/Sim.FR.tRateInt-
1),syplot(1:Stim.Test.FR.T/Sim.FR.tRateInt-1),'m-.', 'LineWidth', 2);
                plot
(statex(1:Stim.Test.FR.T/Sim.FR.tRateInt),statey(1:Stim.Test.FR.T/Sim.F
R.tRateInt),'--', 'LineWidth', 2);
                title (['Reconstruction of the static and adaptive
filters without nonstationarity - signal length = ',
num2str(Stim.Test.FR.T), ' seconds'], 'FontSize', 16)
                set(gca, 'FontSize', 14), legend('Original Signal',
'Static Filter Reconstruction', 'Adaptive Filter Reconstruction');
                xlabel('X velocity V_x', 'FontSize', 14)
                ylabel('Y velocity V_y', 'FontSize', 14)
                drawnow;
                FigHandle(1)=gcf;
                figure
                hold on
                grid on
                plot (sSUCenters(1,:),sSUCenters(2,:),'r-',
'LineWidth', 2);
                plot (sxplot,syplot,'m-.', 'LineWidth', 2);
                plot (statex,statey,'--', 'LineWidth', 2);
                title (['Reconstruction of the static and adaptive
filters with induced nonstationarity - signal length = ',
num2str(Stim.Test.FR.T), ' seconds'], 'FontSize', 16)
                set(gca, 'FontSize', 14), legend('Original Signal',
'Static Filter Reconstruction', 'Adaptive Filter Reconstruction');
                xlabel('X velocity V_x', 'FontSize', 14)
                ylabel('Y velocity V_y', 'FontSize', 14)
                drawnow;
                FigHandle(2)=gcf;
                timeRecon = (1:length(sxplot))*Sim.FR.tRateInt;
                timeReconPlus = (1:length(sxplot)+1)*Sim.FR.tRateInt;
                figure, hold on, grid on,
plot(timeRecon,sSUCenters(1,:),'r', 'LineWidth', 2), plot
(timeRecon,sxplot,'m', 'LineWidth', 2), plot(timeRecon,statex,
'LineWidth', 2)
                title ('Reconstruction of the static and adaptive
filters along x with induced nonstationarity', 'FontSize', 16);
                set(gca, 'FontSize', 14), legend('Original Signal',
'Static Filter Reconstruction', 'Adaptive Filter Reconstruction');
                xlabel('Time (seconds)', 'FontSize', 14)
                ylabel('X velocity V_x', 'FontSize', 14)
                drawnow;
                FigHandle(3)=gcf;
                figure, hold on, grid on, plot
(timeRecon,sSUCenters(2,:),'r', 'LineWidth', 2), plot
```

```
(timeRecon,syplot,'m', 'LineWidth', 2), plot(timeRecon,statey,
'LineWidth', 2)
                title ('Reconstruction of the static and adaptive
filters along y with induced nonstationarity', 'FontSize', 16);
                set(gca, 'FontSize', 14), legend('Original Signal',
'Static Filter Reconstruction', 'Adaptive Filter Reconstruction');
                xlabel('Time (seconds)', 'FontSize', 14)
                ylabel('Y velocity V_y', 'FontSize', 14)
                drawnow;
                FigHandle(4)=gcf;
                timenrmserr =
(1:length(AdaptiveFilter.nrmserrstatx))*AdaptiveFilter.errorwindow;
                figure, hold on, grid on, plot (timenrmserr,
AdaptiveFilter.nrmserrstatx, 'r', 'LineWidth', 2), plot (timenrmserr,
AdaptiveFilter.nrmserrx, '--', 'LineWidth', 2), plot (timenrmserr,
AdaptiveFilter.nrmserrxTstLk, 'k-.', 'LineWidth', 2); % plot
(timenrmserr, AdaptiveFilter.nrmserrRemPopx, 'g:', 'LineWidth', 2),
                title (['NRMS errors along X Test RMS power = ',
num2str(Stim.Test.rms)], 'FontSize', 16)
                set(gca, 'FontSize', 14), legend('Static Filter',
'Adaptive Filter', 'Optimal Population');
                xlabel('Time (seconds)', 'FontSize', 14)
                ylabel('NRMSE_x', 'FontSize', 14)
                drawnow;
                FigHandle(5)=gcf;
                figure, hold on, grid on, plot (timenrmserr,
AdaptiveFilter.nrmserrstaty, 'r', 'LineWidth', 2), plot (timenrmserr,
AdaptiveFilter.nrmserry, '--', 'LineWidth', 2), plot (timenrmserr,
AdaptiveFilter.nrmserryTstLk, 'k-.', 'LineWidth', 2); % plot
(timenrmserr, AdaptiveFilter.nrmserrRemPopy, 'g:', 'LineWidth', 2),
                title (['NRMS errors along Y Test RMS power = ',
num2str(Stim.Test.rms)], 'FontSize', 16)
                set(gca, 'FontSize', 14), legend('Static Filter',
'Adaptive Filter', 'Optimal Population');
                xlabel('Time (seconds)', 'FontSize', 14)
                ylabel('NRMSE_y', 'FontSize', 14)
                drawnow;
                FigHandle(6)=gcf;
                figure, hold on, grid on, plot (timenrmserr,
AdaptiveFilter.nrmsErrorStat, 'r', 'LineWidth', 2), plot (timenrmserr,
AdaptiveFilter.nrmsError, '--', 'LineWidth', 2), plot (timenrmserr,
AdaptiveFilter.nrmsErrorTstLk, 'k-.', 'LineWidth', 2), % plot
(timenrmserr, AdaptiveFilter.nrmsErrorRemPop, 'g:', 'LineWidth', 2),
plot (timenrmserr, AdaptiveFilter.nrmsErrorvalue, 'LineWidth', 4);
                title (['Total NRMS errors Test RMS power = ',
num2str(Stim.Test.rms)], 'FontSize', 16)
                set(gca, 'FontSize', 14), legend('Static Filter',
'Adaptive Filter', 'Optimal Population');
                xlabel('Time (seconds)', 'FontSize', 14)
                ylabel('NRMSE_t_o_t', 'FontSize', 14)
                drawnow;
                FigHandle(7)=gcf;
                figure, plot(timenrmserr,
xscalecat(1:AdaptiveFilter.errorwindow/Sim.FR.tRateInt:nRateSteps),
'LineWidth', 2) % Downsampled to match AdaptiveFilter.errorwindow
                title ('Scale changes along X', 'FontSize', 16)
                set(gca, 'FontSize', 14), legend('Scale variation');
```

```
                xlabel('Time (seconds)', 'FontSize', 14)
                ylabel('Scale_x', 'FontSize', 14)
                FigHandle(8)=gcf;
                figure, plot(timenrmserr,
yscalecat(1:AdaptiveFilter.errorwindow/Sim.FR.tRateInt:nRateSteps),
'LineWidth', 2) % Downsampled to match AdaptiveFilter.errorwindow
                set(gca, 'FontSize', 14), legend('Scale variation');
                title ('Scale changes along Y', 'FontSize', 16)
                xlabel('Time (seconds)', 'FontSize', 14)
                ylabel('Scale_y', 'FontSize', 14)
                FigHandle(9)=gcf;
                if strcmp(Stim.Test.type, 'Figure 8')
                    figure
                    hold on
                    grid on
                    plot (sSUCenters(1,end+1-
(Stim.Test.FR.T/Sim.FR.tRateInt):end),sSUCenters(2,end+1-
(Stim.Test.FR.T/Sim.FR.tRateInt):end),'r-', 'LineWidth', 2);
                    plot(sxplot, syplot,'m-.', 'LineWidth', 2);
                    plot (statex(end+1-
(Stim.Test.FR.T/Sim.FR.tRateInt):end),statey(end+1-
(Stim.Test.FR.T/Sim.FR.tRateInt):end),'g--', 'LineWidth', 2);
                    title (['Final Reconstruction of the static and
adaptive filters with induced nonstationarity - signal length = ',
num2str(Stim.Test.FR.T), ' seconds'], 'FontSize', 16)
                    set(gca, 'FontSize', 14), legend('Original Signal',
'Static Filter', 'Adaptive Filter');
                    xlabel('X velocity V_x', 'FontSize', 14)
                    ylabel('Y velocity V_y', 'FontSize', 14)
                    drawnow;
                    FigHandle(10)=gcf;
                end
            end %j
        end %i


        % TIME TAKEN BY THE SIMULATION
        totalTime = toc;
        TotalTime = sprintf('%d seconds', round(totalTime));
        display(TotalTime)

        % SAVE SIMULATION DATA
        a = date;
        load('runcount.mat');
        runcount = runcount+1;
        if strcmp(Sim.Nonstatdecision, 'Yes')
            foldernameFig = ([a(1:6), ' ', Sim.NonStatType,  ' Run ',
num2str(runcount)]);
        else
            foldernameFig = ([a(1:6), ' ', 'Stationary',  ' Run ',
num2str(runcount)]);
        end
        save runcount runcount
        mkdir(foldernameFig);
        chdir(foldernameFig);
```

```
        save(foldernameFig, 'SUrateResp_training',
'sSUCenters_training', 'Sim', 'Stim', 'SUrateResp', 'sSUCenters',
'AdaptiveFilter', 'xscalecat', 'yscalecat', 'TotalTime'); %,
'Zdiffall', 'ZstatdiffX', 'ZstatdiffY', 'ZTstLkdiffY', 'ZTstLkdiffX',
'ZRemPopdiffY', 'ZRemPopdiffX'); %, 'spikeTrains'); 'phiSU'

        % SAVE FIGURES
        saveas(FigHandle(1), [foldernameFig ' - Static and adaptive
filter reconstruction without nonstationarity'], 'fig');
        saveas(FigHandle(2), [foldernameFig ' - Static and adaptive
filter reconstruction with nonstationarity induced at
',num2str(Sim.begofnonstat),' seconds'], 'fig');
        saveas(FigHandle(3), [foldernameFig ' - X recon'], 'fig');
        saveas(FigHandle(4), [foldernameFig ' - Y recon'], 'fig');
        saveas(FigHandle(5), [foldernameFig ' - X nrms error'], 'fig');
        saveas(FigHandle(6), [foldernameFig ' - Y nrms error'], 'fig');
        saveas(FigHandle(7), [foldernameFig ' - Total nrms error'],
'fig');
        saveas(FigHandle(8), [foldernameFig ' - X Scale'], 'fig');
        saveas(FigHandle(9), [foldernameFig ' - Y Scale'], 'fig');
        if strcmp(Stim.Test.type, 'Figure 8')
            saveas(FigHandle(10), [foldernameFig ' - Final static and
adaptive filter reconstruction'], 'fig');
        end
        for i = 1:length(Snapshotfigs)
            if Snapshotfigs(i)~=0
                saveas(Snapshotfigs(i), [foldernameFig ' - Snapshot '
num2str(i)], 'fig');
            end
        end
        cd ..;

    end
end
```

InitializeNewSim.m

```
function [Sim, Stim] = InitializeNewSim()

Sim.RSeed = sum(100*clock); %162    %Set Random Seed

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Initialize simulation parameters
Sim.nDim = 2;                  %Number of stimulus dimensions
represented across the population
Sim.nUnits = 200;           %Specify number of units for the simulated
populations - FOR ENTIRE POP CHANGE --> Sim.nUnits = 2* Sim.nchangedpop
Sim.nRuns = 1;                 %Number of simulation to run for each
population size
Sim.tAvgWindow = 0.1;          %Temporal averaging window (sec) for
Error statistics
Sim.neuronsPerElect = 3;       %Number of neurons per electrode
Sim.errorwindow = 10; % seconds

Sim.ReOptTimeWindow = 550; % seconds
Sim.LinReOptTimeWindow = 550;

% NONSTATIONARITY INITIALIZATIONS
Sim.Nonstatdecision = 'Yes'; % 'Yes' or 'No'
Sim.begofnonstat = 650; % TIME AT WHICH THE CHANGE BEGINS IN SECONDS
Sim.periodofnonstat = 1;% PERIOD BETWEEN TWO SUCCESSIVE ALTERATIONS
Sim.neuronsEachTime = 100; % NUMBER OF NEURONS ALTERED AT EACH TIME
INSTANT
Sim.nchangedpop = 100; % NUMBER OF NEURONS THAT ARE ALTERED
Sim.endofnonstat = Sim.begofnonstat +
(Sim.periodofnonstat*(Sim.nchangedpop/Sim.neuronsEachTime-1)); % TIME
AT WHICH THE CHANGE ENDS IN SECONDS
Sim.NonStatType = 'Replacement'; % TYPE OF NONSTATIONARITY - 'Loss' OR
'Replacement'OR 'Adaptation' OR 'Attention' OR 'AttentionReplacement'
Sim.NonStatTime =
Sim.begofnonstat:Sim.periodofnonstat:Sim.endofnonstat;

% Adaptation
Sim.TauAdapt = 0.49*(0.05 + round((0.6-0.05)*rand(Sim.nUnits,
1)*1000)/1000); % 0.055*ones(Sim.nUnits, 1); % 50 - 600 ms --> F_adap =
0.51, T_adap = (1 - F_adap)*T_ca... T_ca = 50 - 600 ms
Sim.Radapt = 20*ones(Sim.nUnits, 1); % Larger than R_leak
Sim.Gadapt = zeros(Sim.nUnits, 1);
Sim.Rdec = ones(Sim.nUnits, 1); % 5*R_leak

%Population Temporal-specific parameters
Sim.PT.dt = 0.00025;                    %Time step (sec)
%Define PSC linear filter for decoding
Sim.PT.tauPSC = 0.02*rand(1,Sim.nUnits)+0.01;   %Heterogeneous taus
[10,30]ms - 10-19-06
%Sim.PT.tauPSC = 0.015;%0.005;           %PSC time constant (sec)
Sim.PT.fOrderPSC = 0;                   %Filter Order

%Firing Rate-specific parameters
```

```
Sim.FR.dt = 0.001;%0.00025;             %Time step (sec)
Sim.FR.FiltLength = 1;                  %length of linear rate filter
(s)
Sim.FR.tRateInt = 0.05;                 %Temporal window used to est
firing rate from spike train

% Attention
Sim.AttnPeriod = 5; %seconds
Sim.AttentionMod = [0.8 1.2]; % range of modulation produced by
attention

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Initialize neuron population parameters
Sim.phiEnc_func ='vonMisesTuningResp'; %'LinearTuningResp'
'GaussTuningResp' 'CosineTuningResp' 'vonMisesTuningResp'
Sim.error = 0.1;                    %Percentage error in neuron
response due to noise
Sim.maxRespRange =  [20 80];  %[100 300];    %Range of max. responses
(spikes/s)
Sim.V_th = 1;
Sim.R_leak = 1;
Sim.tauRefRange = [0.002 0.005];     %Set range for refactory periods
across the neural population
Sim.tauRCRange = [0.01 0.03];        %Set range for RC-time constants
across the neural population
Sim.Tau = 1;

%Specify phase shift in signal representation (in time steps)
introduced by
%convolution with the PSC filter in the reconstruction. Used to adjust
time
%series for computation on MSE.
%phShift = uint32(round(Sim.tauPSC/Sim.dt));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Initialize signal parameters
Stim.PT.dt = Sim.PT.dt;
Stim.FR.dt = Sim.FR.dt;

%TRAINING
Stim.Training.type = 'White Noise'; %'White Noise'; % '2D Plane';
%'Spiral Sampling';
switch (Stim.Training.type)
    case 'Spiral Sampling'
        Stim.sRange = (-2:0.001:2)*pi/2;     %Signal Range
        Stim.Training.maxRad = 200*pi;
        Stim.Training.minRad = 0;
        Stim.Training.maxMag = 2;
        Stim.Training.FR.T = 200;
    case 'White Noise'
        Stim.sRange = -1:1/180:1;             %Signal Range - sampling
per degree = 360 samples
        if Sim.nDim == 2
            Stim.sRange = Stim.sRange.*pi./max(Stim.sRange);
        end
```

```
        Stim.Training.randomSeed = Sim.RSeed;%0;%99;     %RandomSeed>0
resets the random number generator, =0 selects new state, <0 uses
existing state
        Stim.Training.PT.T = 10;             %Length of PT training
signal in seconds
        Stim.Training.FR.T = 2.5*Sim.nUnits;    %
(ceil(Sim.nUnits*Sim.nDim/100))*100;  %Length of FR training signal in
seconds = %120 of df (p180 = 450, p120 = 300, p100 = 250, p80 = 200,
p40 = 100, p20 = 50)
        Stim.Training.upperBandLimit = 1.5;%5;    %High frequency
cutoff for white noise signal
        Stim.Training.lowerBandLimit = 0;    %Low frequency cutoff for
white noise signal
        Stim.Training.rms = 1;%1;               %RMS signal level
        Stim.Training.maxMag = sqrt(2); % Changed to 2 - Aug 13 2008
%max(abs(Stim.sRange));
        Stim.Training.bandwidth = [Stim.Training.lowerBandLimit
Stim.Training.upperBandLimit]*2*pi;
    otherwise
        error('Invalid type for training stimulus');
end

%TEST
Stim.Test.type = 'White Noise'; % 'Figure 8', 'White Noise', 'Circle',
'Constant'
Stim.Test.PT.T = 1.0;                  %Length of PT test signal in
seconds
Stim.Test.FR.T = 2000.0;                 %Length of FR test signal in
seconds
Stim.Test.FR.tst_runs = 1;            % Runs of replicating test
stimulus
switch (Stim.Test.type)
    case 'Figure 8'
        Stim.Test.maxRad = 40*pi;
        Stim.Test.minRad = 0;
        Stim.Test.maxMag = 2;
    case 'White Noise'
        Stim.Test.randomSeed = 0;%6546546;       %RandomSeed>0 resets
the random number generator
        Stim.Test.upperBandLimit = 1;      %High frequency cutoff for
white noise signal
        Stim.Test.lowerBandLimit = 0;        %Low frequency cutoff for
white noise signal
        Stim.Test.rms = 1;                %RMS signal level
        Stim.Test.bandwidth = [Stim.Test.lowerBandLimit
Stim.Test.upperBandLimit]*2*pi;
    case 'Circle'
        Stim.Test.rms = Stim.Training.rms; % Set the radius of the
        Stim.Test.radius = Stim.Test.rms;  % circle
        Stim.Test.degreepert = 1/200; % Aribitary step along the
circumference
    case 'Constant'
        Stim.Test.theta = [-pi/2 pi/2];
        Stim.Test.mag = 1/sqrt(2);
        Stim.Test.rms = 1;
end
```

InitvonMisesLIFNeurons.m

```
function [LIFparams, noiseVar, maxResp, a_S, kappa, halfwidth] =
InitvonMisesLIFNeurons(S, N, Spref, maxRespRange, tauRefRange,
tauRCRange, V_th, R_leak, error, figNum)

% [LIFparams, noiseVar, maxResp, a_S, kappa, halfwidth] =
InitvonMisesLIFNeurons(S, N, Spref, maxRespRange, tauRefRange,
tauRCRange, V_th, R_leak, error, figNum);
%
% Initializes the LIF parameters for a population of Gaussian tuned
neurons.
%
%-------INPUTS--------
% "S" is an 1xNt vector containing the range of representative values
over
%       which the neurons should encode a signal (e.g., -2:0.1:2).
% "N" specifies the number of neurons to initialize.
% "Spref" is a MxN matrix specifying the location of the M-dimensional
mean
%   for each neuron's tuning curve within the range specified by S. For
%   Cosine tuned neurons the mean corresponds to the neuron's preferred
%   stimulus.
% "maxRespRange" is a 1x2 vector specifying the range of maximum
responses (spikes/s)
%   for the population of neurons [maxresp_low maxresp_high]. Each
neuron's
%   maximum response is selected randomly from the range.
% "tauRefRange" is a 1x2 vector specifying the range of refractory
times (sec)
%   for the population of neurons. Each neuron's refractory time is
selected
%   randomly from the range.
% "tauRCRange" is a 1x2 vector specifying the range of RC time
constants (sec)
%   for the population of neurons. Each neuron's RC time constant is
selected
%   randomly from the range.
% "V_th" specifies the voltage threshold used to determine when an
action
%   potential occurs. Curently this value is applied to all neurons.
% "R_leak" specifies the leakage resistance across the neurons' cell
membrane.
%   Curently this value is applied to all neurons.
% "error" specifies the percentage error in neuron response due to
noise
%   for preferred stimulus. The value is specified as a ratio relative
to the
%   neuron's maximum response.
% "figNum" specifies the figure number to display a plot of the tuning
%   curves for the population of neurons. If figNum = 0, no figure is
displayed.
%
%-------OUTPUTS--------
% "LIFparams" is a MxN matrix containing the LIF parameters specific to
```

```
%      each neuron. Each row specifies values for a specifi LIF
parameter
%       (1,1:N) -> Refractory periods (sec)
%       (2,1:N) -> RC time-constants (sec)
%       (3,1:N) -> Gains of driving input
%       (4,1:N) -> Bias currents (amps)
%       (5,1:N) -> Threshold voltages (volts)
%       (6,1:N) -> Leakage resistances (ohms)
%       (7;9;11;etc,1:N) -> Preferred stimulus (mean of Cosine tuning).
%                           One row per dimension
% "noiseVar" is a 1xN vector of noise variances (spikes/s) for the
initialized neurons.
% "maxResp"  of maximum responses (spikes/s) for the initialized
neurons.
% "a_S" represents the tuning curves for the entire population of
neurons.
% "kappa" is a 1xN vector of constants related to the tuning half-width
of the neuron
% "halfwidth" is a 1xN vector of tuning halfwidths for the entire
% population of neurons.

% vonMises tuning width - consistent with Amirikian and Georgopulos
(2000)
% Jan 24, 2008
% Tushar Dharampal
% Integrative Neural Systems Lab
halfwidth = zeros(N,1);
kappa = zeros(N,1);

kappaRange = 0.01:0.01:5; % Empirical range
deltaRange = acosd((log(exp(2.*kappaRange)+1)-log(2)-
kappaRange)./kappaRange);

v1 = find(deltaRange >= 30 & deltaRange < 45);
v2 = find(deltaRange >= 45 & deltaRange < 60);
v3 = find(deltaRange >= 60 & deltaRange < 75);
v4 = find(deltaRange >= 75 & deltaRange < 90);

pop1 = round((6/30)*N);  % 30 - 45 degrees
pop2 = round((11/30)*N); % 46 - 60
pop3 = round((8/30)*N);  % 61 - 75
pop4 = N-(pop1+pop2+pop3);  % 76 - 89

for i = 1:pop1
    index = ceil(rand()*length(v1));
    kappa(i) = kappaRange(v1(index));
    halfwidth(i) = deltaRange(v1(index));
end

for i = pop1+1:pop1+pop2
    index = ceil(rand()*length(v2));
    kappa(i) = kappaRange(v2(index));
    halfwidth(i) = deltaRange(v2(index));
end
```

```
for i = pop1+pop2+1:pop1+pop2+pop3
    index = ceil(rand()*length(v3));
    kappa(i) = kappaRange(v3(index));
    halfwidth(i) = deltaRange(v3(index));
end

for i = pop1+pop2+pop3+1:pop1+pop2+pop3+pop4
    index = ceil(rand()*length(v4));
    kappa(i) = kappaRange(v4(index));
    halfwidth(i) = deltaRange(v4(index));
end

clear i;

nDim = size(Spref,2);
Nt = length(S);
J_th = V_th/R_leak;
tauRef = (tauRefRange(2) - tauRefRange(1))*rand(N,1) + tauRefRange(1);
%Set refactory period randomly for each neuron
tauRC = (tauRCRange(2) - tauRCRange(1))*rand(N,1) + tauRCRange(1);
%Set RC-time constant randomly for each neuron

maxResp = (maxRespRange(2) - maxRespRange(1))*rand(N,1) +
maxRespRange(1);
noiseVar = maxResp.*error;                  %Compute noise variance for
each neurons (spikes/s)

v = find(maxResp > 1./tauRef);  %Look for max resp. values that violate
the absolute refractory period
while ~isempty(v)
    maxResp(v) = (maxRespRange(2) - maxRespRange(1))*rand(length(v),1)
+ maxRespRange(1);
    v = find(maxResp > 1./tauRef);
end

%Compute alpha and Jbias for von Mises tuning response based on
%the neuron's preferred stimulus, tuning variance, maximum response,
and baseline noise (expressed as % of max response).
minResp = noiseVar;
J_bias = J_th*(1./(1-exp((tauRef.*minResp-1)./(tauRC.*minResp))));
J_bias_sigma = abs(J_bias - (J_th*(1./(1-
exp((tauRef.*(minResp+sqrt(noiseVar))-
1)./(tauRC.*(minResp+sqrt(noiseVar)))))))));
alpha = J_th*(1./(1-exp((tauRef.*maxResp-1)./(tauRC.*maxResp)))) -
J_bias;


LIFparams(1,:) = tauRef';  %Refractory period
LIFparams(2,:) = tauRC';   %RC time-constant
LIFparams(3,:)= alpha';   %Gain of driving input
LIFparams(4,:) = J_bias';  %Bias current
LIFparams(5,:) = V_th*ones(1,N);    %Threshold voltage
LIFparams(6,:) = R_leak*ones(1,N);  %Leakage resistance
LIFparams(7:7+nDim-1,:) = Spref';   %preferred direction of neuron
```

```
LIFparams(end+1,:) = J_bias_sigma; % S.D of variation in the J_bias
values

if figNum > 0
    figure(figNum);
    clf;
    offset = 1./exp(kappa);
    scale = (exp(kappa)-offset)*ones(1,Nt);
    Jin =
alpha*ones(1,Nt).*((exp(kappa*ones(1,Nt).*cos(angle_mod(ones(N,1)*S(1,:
),Spref(:,1)*ones(1,Nt)))) - offset*ones(1,Nt))./scale);
    Jin = Jin + J_bias*ones(1,Nt);
    a_S = 1./((tauRef*ones(1,length(S)))-
(tauRC*ones(1,length(S))).*log(1-J_th./Jin));
    plot(S(1,:)*180/pi,a_S)
    title ('Tuning curves for the entire population of neurons',
'FontSize', 16);
    xlabel ('Preferred Direction (Degrees)', 'FontSize', 16);
    ylabel ('Firing rate (spikes/second)', 'FontSize', 16);
    drawnow;
    figure
    plot(S(1,:)*180/pi,sum(a_S))
    title ('Sum of tuning profiles for the entire population of
neurons', 'FontSize', 16);
    xlabel ('Preferred Direction (Degrees)', 'FontSize', 16);
    ylabel ('Firing rate (spikes/second)', 'FontSize', 16);
    if(N<=100)
        axis([-200 200 0 4000])
    else
        axis([-200 200 0 8000])
    end
    drawnow;
    if(find(~isfinite(sum(a_S, 2))))
        error('a_S has a NaN')
    end
end
```

GetNeuronFiringRatesIterative_G.m

```
function [SUrateResp, sSUCenters, LIFinit, Gadapt, spikeTimes,
GadaptTemp] = GetNeuronFiringRatesIterative_G(Sim, Stim, Sin, LIFinit,
NumNeurons, AttnSig, Gadapt)
% [SUrateResp, sSUCenters, LIFinit, Gadapt, spikeTimes, GadaptTemp] =
GetNeuronFiringRatesIterative_G(Sim, Stim, Sin, LIFinit, NumNeurons,
AttnSig, Gadapt)

Generate the firing rates for the provided stimulus for the given
neuron population

%-------INPUTS--------
% "Sim" is the structure that holds the simulation specific parameters.
% "Stim" is the structure that holds the stimulus specific parameters.
% "Sin" is the provided stimulus along two-dimensions.
% "LIFinit" is a structure that carries over the current state of each
neuron for the next
%   call to genLIFSpikes_iterate. It contains the fields:
%       ".V" is a 1xN vector containing the final voltage for each
neuron
%       ".EndRefPeriod" is a 1xN vector containing the ending time for
each neuron's refractory
%           period relative to the local time for the next function
call.
%       ".jitterSig" is a 1xN vector containing the standard deviations
of the random temporal jitters
%           applied to the timing of each neuron's spikes.
% "NumNeurons" is the number of neurons in the population.
% "AttnSig" is a vector that represents the attention signal over the
% length of the stimulus
% "Gadapt" is a vector representing the current adaptive conductance
for
% each neuron.
%
%-------OUTPUTS--------
% "SUrateResp" is a vector that holds the binned rates for each neuron
over the
% current timestep.
% "sSUCenters" is a 2x1 vector tjat holds the averaged stimulus along
two
% dimensions for each timestep.
% "LIFinit" is a structure that carries over the current state of each
neuron for the next
%   call to genLIFSpikes_iterate. It contains the fields:
%       ".V" is a 1xN vector containing the final voltage for each
neuron
%       ".EndRefPeriod" is a 1xN vector containing the ending time for
each neuron's refractory
%           period relative to the local time for the next function
call.
%       ".jitterSig" is a 1xN vector containing the standard deviations
of the random temporal jitters
%           applied to the timing of each neuron's spikes.
% "Gadapt" is a vector representing the current adaptive conductance
for
```

```
% each neuron.
% "spikeTimes" is a NxP matrix containing the times for each action
potential.
%   The dimension P is specified by the neurons with the most spikes
=max(spikeCount).
%   For neurons with fewer spikes (Q; Q<P) the row of spike times is
padded
%   with P-Q zeros to complete the matrix.
% "GadaptTemp" is a matrix used to hold the Gadapt values between
function
% calls.


i=1; % ONE POPULATION

Nt = length(Sin);

%Parse Global Stuctures
LIFparams = Sim.Pop(i).LIFparams(:,1:NumNeurons);
noiseVar = Sim.Pop(i).noiseVar(1:NumNeurons);
maxResp = Sim.Pop(i).maxResp(1:NumNeurons);

scale = Sim.Pop(i).SmaxLin;

% Incorporate attention responses
if strcmp(Sim.Nonstatdecision, 'Yes') && (strcmp(Sim.NonStatType,
'Attention') || strcmp(Sim.NonStatType, 'AttentionReplacement'))% &&
strcmp(SigType, 'Test')
   scale = scale./AttnSig;
end

%Convert signal to polar form for Gaussian tuned neurons
switch(Sim.phiEnc_func)
    case 'GaussTuningResp'
        if Sim.nDim == 1
            Sin_mag = max(Stim.sRange)*ones(1,Nt);
            Sin_angle = Sin;
        else
            Sin_mag = sqrt(sum(Sin.^2,1));
            Sin_angle = atan2(Sin(2,:), Sin(1,:));
        end
        Sin_plr = {Sin_mag, Sin_angle};
        respParam = {Sim.Pop(i).Spref(1:NumNeurons),
Sim.Pop(i).Ssig(1:NumNeurons), scale};

    case 'LinearTuningResp'
        Sin_plr = {Sin_temp};
        respParam = {phiEnc'};

    case 'CosineTuningResp'
        if Sim.nDim == 1
            Sin_mag = max(Stim.sRange)*ones(1,Nt);
            Sin_angle = Sin;
        else
            Sin_mag = sqrt(sum(Sin.^2,1));
```

```
            Sin_angle = atan2(Sin(2,:), Sin(1,:));
        end
        Sin_plr = {Sin_mag, Sin_angle};
        respParam = {Sim.Pop(i).Spref(1:NumNeurons), scale};

    case 'vonMisesTuningResp'
        if Sim.nDim == 1
            Sin_mag = max(Stim.sRange)*ones(1,Nt);
            Sin_angle = Sin;
        else
            Sin_mag = sqrt(sum(Sin.^2,1));
            Sin_angle = atan2(Sin(2,:), Sin(1,:));
        end
        Sin_plr = {Sin_mag, Sin_angle};
        respParam = {Sim.Pop(i).Spref(1:NumNeurons),
Sim.Pop(i).kappa(1:NumNeurons), scale};
    otherwise
        error('Invalid stimulus tuning profile specified');
end
clear Sin_temp;

%Compute alpha and Jbias for rectified linear tuning response based on
%the neuron's x-intercept and maximum response.
alpha = LIFparams(3,1:NumNeurons)';
J_bias = LIFparams(4,1:NumNeurons)';

J_bias_sigma = LIFparams(end,1:NumNeurons)';

%Compute driving current and corresponding neuron reponses for the
input signal
J_d = CalcDrivingCurrent(alpha, Nt, Sim.phiEnc_func, respParam,
Sin_plr);%, expa);

J_in = J_d + (J_bias*ones(1,Nt)+J_bias_sigma*randn(1,Nt)); % Add
variability to the J_bias values with a S.D. of J_bias_sigma - August
24 2007

[spikeCount, spikeTimes, LIFinit, Gadapt, GadaptTemp] =
genLIFSpikes_iterate_G(J_in, Stim.FR.dt, LIFparams, noiseVar, maxResp,
LIFinit, Gadapt, Sim.TauAdapt, Sim);

SUrateResp = spikeCount/Sim.FR.tRateInt;

sSUCenters = (mean(Sin,2))';
```

genLIFSpikes_iterate_G.m

```
function [spikeCount,spikeTimes,LIFinit,Gadapt,GadaptTemp] =
genLIFSpikes_iterate_G(J_in, dt, LIFparams, noiseVar, maxResp, LIFinit,
Gadapt, TauAdapt, Sim)
% [spikeCount,spikeTimes,LIFinit,Gadapt,GadaptTemp] =
genLIFSpikes_iterate_G(J_in, dt, LIFparams, noiseVar, maxResp, LIFinit,
Gadapt, TauAdapt, Sim)
%
% Compute timing of action potentials for a population of Leaky
Integrate and Fire (LIF)
% neurons based on the integrated input current received by each neuron
% including optional adaptation of neuron responses.
%
%-------INPUTS--------
% "J_in" is an NxNt matrix containig the input current received by N
neurons
%       for each of Nt time points.
% "dt" is the interval between time points expressed in sec.
% "LIFparams" is a MxN matrix containing the LIF parameters specific to
%      each neuron. The parameters matrix is generated automatically
using
%      the functions InitGaussLIFNeurons or InitLinearLIFNeurons to
%      generate neurons with Gaussian or linear tuning curves
respectively.
%      The parameters specific to each row are
%       (1,1:N) -> Refractory periods (sec)
%       (2,1:N) -> RC time-constants (sec)
%       (3,1:N) -> Gains of driving input
%       (4,1:N) -> Bias currents (amps)
%       (5,1:N) -> Threshold voltages (volts)
%       (6,1:N) -> Leakage resistances (ohms)
% "noiseVar" is a 1xN vector of noise variances (spikes/s). This vector
is
%   generated automatically together with LIFparams as part of the
neuron
%   initialization.
% "maxResp" is a 1xN vector of maximum responses (spikes/s). his vector
is
%   generated automatically together with LIFparams as part of the
neuron
%   initialization.
% "LIFinit" is a structure that carries over the current state of each
neuron for the next
%   call to genLIFSpikes_iterate. It contains the fields
%       ".V" is a 1xN vector containing the final voltage for each
neuron
%       ".EndRefPeriod" is a 1xN vector containing the ending time for
each neuron's refractory
%           period relative to the local time for the next function
call.
%       ".jitterSig" is a 1xN vector containing the standard deviations
of the random temporal jitters
%           applied to the timing of each neuron's spikes.
```

```
% "Gadapt" is a vector representing the current adaptive conductance
for
% each neuron.
% "TauAdapt" is a vector representing the time period of adaptation for
% each neuron.
% "Sim" is the structure that holds the simulation specific parameters.
%
%-------OUTPUTS--------
% "spikeCount" is a 1xN vector containing the total number of spikes
generated
%   during the input sequence for each neuron.
% "spikeTimes" is a NxP matrix containing the times for each action
potential.
%   The dimension P is specified by the neurons with the most spikes
=max(spikeCount).
%   For neurons with fewer spikes (Q; Q<P) the row of spike times is
padded
%   with P-Q zeros to complete the matrix.
% "LIFinit" is a structure that carries over the current state of each
neuron for the next
%   call to genLIFSpikes_iterate. It contains the fields
%       ".V" is a 1xN vector containing the final voltage for each
neuron
%       ".EndRefPeriod" is a 1xN vector containing the ending time for
each neuron's refractory
%           period relative to the local time for the next function
call.
%       ".jitterSig" is a 1xN vector containing the standard deviations
of the random temporal jitters
%           applied to the timing of each neuron's spikes.
% "Gadapt" is a vector representing the current adaptive conductance
for
% each neuron.
% "GadaptTemp" is a matrix used to hold the Gadapt values between
function
% calls.

% Created 4-1-06 (Scott Beardsley)
%
% Modification History:
%

%Initialize LIF paramters
N = size(LIFparams,2);
tauRef = LIFparams(1,:)';  %Refractory period
tauRC = LIFparams(2,:)';   %RC time-constant
alpha = LIFparams(3,:)';   %Gain of driving input
J_bias = LIFparams(4,:)';  %Bias current
V_th = LIFparams(5,:)';    %Threshold voltage
R_leak = LIFparams(6,:)';  %Leakage resistance

spikeCount = zeros(1,N);

spikeTimes = zeros(N, ceil(Sim.FR.tRateInt/Sim.FR.dt));

T = (size(J_in, 2)-1)*dt;   %Total Time
```

```
RateSS = 0.49*Sim.Pop.maxResp;
B = ((1./tauRC).*(tauRef - 1./RateSS));
Jm = alpha + J_bias;
Q = 1./Jm;

As = (B.^2)/2;
Bs = (B + (Q./exp(B)));
Cs = (1 - ((1-Q)./exp(B)));
Gad = (-Bs + sqrt(Bs.^2 - 4.*As.*Cs))./(2.*As);

Ginc = (1 - ((1 - dt./TauAdapt).^(1./(dt.*RateSS)))).*Gad;

if isempty(LIFinit.jitterSig)
    resJitter = dt/4;            %Jitter in spike timing due to
resolution of the time step
    maxJitterSig = (1./maxResp - 1./(maxResp+sqrt(noiseVar)))/4;
    jitterSig = resJitter.*ones(1,N); % Changed to increase variability
in the Inter Spike Intervals
    z = find(maxJitterSig>resJitter);   %Find neurons whose temporal
jitter due to noise exceeds the time step resolution
    if ~isempty(z)
        jitterSig(z) = maxJitterSig(z); %Use the larger source of
jitter (i.e., jitter due to noise) for the above neurons
    end
    LIFinit.jitterSig = jitterSig;
end

GadaptTemp = zeros(N, size(J_in,2)-1);

for j = 1:N
    V(1) = LIFinit.V(j);
    endRefPeriod = LIFinit.EndRefPeriod(j);

    for i = 2:size(J_in,2) %Loop over the length of the signal J_in(t)
        if i*dt > endRefPeriod
            if strcmp(Sim.Nonstatdecision, 'Yes') &&
strcmp(Sim.NonStatType, 'Adaptation')
                V(i) = V(i-1)-(V(i-1)+ V(i-1)*(R_leak(j)*Gadapt(j)) -
J_in(j, i-1)*R_leak(j))/tauRC(j)*dt; % Adaptive LIF neuron Voltage
            else
                V(i) = V(i-1)-(V(i-1) - J_in(j, i-
1)*R_leak(j))/tauRC(j)*dt; % Normal LIF neuron Voltage
            end
            if V(i)>=V_th(j)
                tJitter = (randn*LIFinit.jitterSig(j));   %Incorporate
noise as variability in spike timing
                tSpike = (i-1)*dt + tJitter;
                if (spikeCount(j) ~= 0 && tSpike <=
(spikeTimes(j,spikeCount(j)) + tauRef(j)))
                    tSpike = (spikeTimes(j,spikeCount(j)) + tauRef(j));
                end
                if  tSpike <= T && tSpike >= dt
                    spikeCount(j) = spikeCount(j) + 1;
                    spikeTimes(j,spikeCount(j)) = tSpike;
```

```
                    endRefPeriod = spikeTimes(j,spikeCount(j)) +
tauRef(j);
                end
                V(i) = 0;
                if strcmp(Sim.Nonstatdecision, 'Yes') &&
strcmp(Sim.NonStatType, 'Adaptation') %&& (Radapt(j) > Sim.R_leak)


                    Gadapt(j) = Gadapt(j) + Ginc(j);
                end
            else
                if strcmp(Sim.Nonstatdecision, 'Yes') &&
strcmp(Sim.NonStatType, 'Adaptation') %&& (Radapt(j) < Sim.Radapt(j))

                    Gadapt(j) = Gadapt(j) -
(Gadapt(j)/(TauAdapt(j)/dt));
                    if Gadapt(j) <= 0
                        Gadapt(j) = 0;
                    end
                end
            end
        else
            V(i) = 0;
        end


        GadaptTemp(j,i) = Gadapt(j);
    end

    LIFinit.V(j) = V(i);                                %Carry over each
neuron's final voltage for next function call
    LIFinit.EndRefPeriod(j) = endRefPeriod-T;          %Adjust each
neuron's endRefPeriod to the local time for the next function call
    %Reset V
    V = V*0;

end
```

vonMisesTuningResp.m

```
function resp = vonMisesTuningResp(S, p)

% resp = vonMisesTuningResp(S, p);
%
% Calculates the response (spikes/s) of von Mises tuned neurons to a 1D
or 2D
% signal expressed in polar coordinates.
%
%-------INPUTS--------
% "S" is a {1xM} cell array of 1xNt vectors containing the M-
dimensional input signal
%       over Nt time steps.
%           [s{1}] is an optional input containing the magnitude of a
2D stimulus.
%                When present it scales the amplitude of the Cosine
%                response.
%           s{2} is the polar angle of the 2D stimulus.
% "p" is a {1x3} cell array of 1xN vectors containing the Cosine tuning
parameters for
%       N neurons.
%           p{1} contains the preferred stimulus angle
%           p{2} contains the kappa value - the scale in the
exponential
%           of the von Mises tuning
%           [p{3}] is an optional parameter that normalizes stimulus
magnitude.
%                It is used to incorporate linear tuning as a function
of radius.
%
%-------OUTPUTS--------
% "resp" is a NxNt matrix containing the responses (spikes/s) of N
neurons at Nt
%       time steps.
%

% Created 11 - 6 - 07 (Tushar Dharampal)
%
% Modification History:
%11 - 6 - 07 Initializing resp variable (Tushar Dharampal)

N = length(p{1});
Nt =length(S{1});
resp = zeros(N,Nt);
kappa = p{2}';
for j = 1:N
    % Jan 24 2008
    % Tushar Dharampal
    % Subtract from and scale the tuning function in order to conform
it to the
    % Alpha-Jbias format i.e be able to use the same Alpha and Jbias
equations
    % as before
    resp(j,:) =
S{1}./p{3}.*((exp(kappa(j)*ones(1,Nt).*cos(angle_mod(S{2},p{1}(j,:)'*on
```

```
es(1,Nt))))-(1./exp(kappa(j)))*ones(1,Nt))./((exp(kappa(j))-
(1./exp(kappa(j))))*ones(1,Nt)));
end
```

GetDecodingWeights.m

```
function [A,H,W,Q] = GetDecodingWeights(S, a_S)

% [phi] = GetDecodingWeights(S, a_S, noiseVar);
%
% Computes the optimal decoding weights for a fixed temporal filter.
%
%-------INPUTS--------
% "S" is a 1xNt vector containing the signal amplitudes at Nt time
points
% "a_S" is a NxNt matrix containing the convolution of the temporal
decoding
%      with the spike trains of N neurons. The result approximates the
instantaneous
%      firing rate of each neuron at each time point.
%
%-------OUTPUTS--------
% "phi" is a 1xN vector containing the optimal decoding weights used to
%   perform the signal decoding and reconstruction.
%
% Created 8-16-06 (Scott Beardsley)
%
% Modification History:
%
%Estimate decoding weights w/ noise
gamma = S*S'; %+ (noiseVar*ones(1,N).*eye(N, N));
upsilon = a_S*S';
phi = upsilon*inv(gamma);

X1 = S(:,1:size(S,2)-1);
X2 = S(:,2:size(S,2));

A = X2*X1' * inv(X1*X1');
H = phi;
Z = a_S;
X = S;
W = (X2 - A*X1)*(X2 - A*X1)'/size(X1,2);

Q = (Z - H*X)*(Z - H*X)'/size(X,2);
```

InitAdaptiveFilter.m

```
function [AdaptiveFilter adaptiveKalman] =
InitAdaptiveFilter(AdaptiveFilter)
% [AdaptiveFilter adaptiveKalman] = InitAdaptiveFilter(AdaptiveFilter)

% The 'InitAdaptiveFilter' function initializes the components of the
adaptive filter.
% For the adaptive Kalman filter there are three filters to be
initialized.
%
% INPUTS
% -------
% AdaptiveFilter = Struct variable with variables specific to the
adaptive filter
%
% OUTPUTS
% --------
% adaptiveKalman = Struct variable with Kalman filter specific
initializations

AdaptiveFilter.type = 'Kalman Filter';
Px =
(AdaptiveFilter.static.Hsu(:,1)\AdaptiveFilter.static.Qsu)/AdaptiveFilt
er.static.Hsu(:,1)'; %P = inv(H)*R*inv(H')
Py =
(AdaptiveFilter.static.Hsu(:,2)\AdaptiveFilter.static.Qsu)/AdaptiveFilt
er.static.Hsu(:,2)'; %P = inv(H)*R*inv(H')

% FILTER 1
adaptiveKalman.adaptfilt1x=[];
adaptiveKalman.adaptfilt1x.A = AdaptiveFilter.static.Asu(1,1);
adaptiveKalman.adaptfilt1x.B = 0;
adaptiveKalman.adaptfilt1x.H = AdaptiveFilter.static.Hsu(:,1);
adaptiveKalman.adaptfilt1x.Q = AdaptiveFilter.static.Wsu(1,1);
adaptiveKalman.adaptfilt1x.R = AdaptiveFilter.static.Qsu;
adaptiveKalman.adaptfilt1x.u = 0;
adaptiveKalman.adaptfilt1x.P = Px;

adaptiveKalman.adaptfilt1y=[];
adaptiveKalman.adaptfilt1y.A = AdaptiveFilter.static.Asu(2,2);
adaptiveKalman.adaptfilt1y.B = 0;
adaptiveKalman.adaptfilt1y.H = AdaptiveFilter.static.Hsu(:,2);
adaptiveKalman.adaptfilt1y.Q = AdaptiveFilter.static.Wsu(2,2);
adaptiveKalman.adaptfilt1y.R = AdaptiveFilter.static.Qsu;
adaptiveKalman.adaptfilt1y.u = 0;
adaptiveKalman.adaptfilt1y.P = Py;

% FILTER 2
adaptiveKalman.adaptfilt2x=[];
% adaptiveKalman.adaptfilt2x.P = 0.1;
adaptiveKalman.adaptfilt2x.A = eye(1);
adaptiveKalman.adaptfilt2x.B = 0;
adaptiveKalman.adaptfilt2x.Q = zeros(1);
adaptiveKalman.adaptfilt2x.R = AdaptiveFilter.static.Qsu(1,1);
```

```
adaptiveKalman.adaptfilt2x.u = 0;
adaptiveKalman.adaptfilt2x.x = AdaptiveFilter.static.Hsu(:,1)';

adaptiveKalman.adaptfilt2y=[];
% adaptiveKalman.adaptfilt2y.P = 0.1;
adaptiveKalman.adaptfilt2y.A = eye(1);
adaptiveKalman.adaptfilt2y.B = 0;
adaptiveKalman.adaptfilt2y.Q = zeros(1);
adaptiveKalman.adaptfilt2y.R = AdaptiveFilter.static.Qsu(1,1);
adaptiveKalman.adaptfilt2y.u = 0;
adaptiveKalman.adaptfilt2y.x = AdaptiveFilter.static.Hsu(:,2)';

% FILTER 3
adaptiveKalman.adaptfilt3x = [];
adaptiveKalman.adaptfilt3x.A = AdaptiveFilter.static.Asu(1,1);
adaptiveKalman.adaptfilt3x.B = 0;
adaptiveKalman.adaptfilt3x.Q = AdaptiveFilter.static.Wsu(1,1);
adaptiveKalman.adaptfilt3x.R = AdaptiveFilter.static.Qsu;
adaptiveKalman.adaptfilt3x.P = Px;
adaptiveKalman.adaptfilt3x.u = 0;
adaptiveKalman.adaptfilt3x.H = AdaptiveFilter.static.Hsu(:,1);

adaptiveKalman.adaptfilt3y = [];
adaptiveKalman.adaptfilt3y.A = AdaptiveFilter.static.Asu(2,2);
adaptiveKalman.adaptfilt3y.B = 0;
adaptiveKalman.adaptfilt3y.Q = AdaptiveFilter.static.Wsu(2,2);
adaptiveKalman.adaptfilt3y.R = AdaptiveFilter.static.Qsu;
adaptiveKalman.adaptfilt3y.P = Py;
adaptiveKalman.adaptfilt3y.u = 0;
adaptiveKalman.adaptfilt3y.H = AdaptiveFilter.static.Hsu(:,2);

AdaptiveFilter.xest=0;
AdaptiveFilter.yest=0;
AdaptiveFilter.zest = zeros(size(AdaptiveFilter.static.Hsu,1),1);
AdaptiveFilter.xtrue=0;
AdaptiveFilter.ytrue=0;

AdaptiveFilter.window_size_filter = 1; % Iterations
AdaptiveFilter.window_size_scale = 50; % Iterations
AdaptiveFilter.xscale = 0.2;
AdaptiveFilter.yscale = 0.2;
AdaptiveFilter.errorwindow = 10; % TIME WIDTH FOR RMS ERROR CALCULATION
IN SECONDS
AdaptiveFilter.true_avgerrorx = 0;
AdaptiveFilter.true_avgerrory = 0;
AdaptiveFilter.prev_avgerrorx = 0;
AdaptiveFilter.prev_avgerrory = 0;
```

Kalmansnapshot.m

```
function [sxplot, syplot] = Kalmansnapshot(TestType, Sim, Stim,
Nusable, H, AdaptiveFilter, P, R, flag, indchangedpopNusable,
indchangedpopReplace, indchangedpopLoss, cntt)

% The Kalmansnapshot function is used to measure the performance of the
algorithm as a snapshot during various points in the simulation.
% It gives a reconstruction of the desired stimulus as if it were the
current TEST stimulus (at this point in the simulation).
% {The 'Figure of 8' stimulus is chosen because the response is easily
assessed qualitatively}
% It does not alter the state / weights of the system in any way.
%
% INPUTS
% -------
% TestType = Stimulus used for the snapshot test
% Sim = Simulation parameters.
% Stim = Stimulus parameters.
% Nusable = The number of neurons that are used for the reconstruction.
% H = Current H matrix (hence, the snapshot).
% AdaptiveFilter = Adaptive filter parameters.
% changedpopcat = The indices for the neurons that are altered.
% flag = Variable that indicates the type of nonstationarity.
%
% OUTPUTS
% --------
% sxplot = Reconstruction along the X dimension.
% syplot = Reconstruction along the Y dimension.

TestLength = 5;
t = 0:Stim.FR.dt:TestLength;
N = Sim.nUnits;

switch (TestType)
    case 'Constant'
        theta = Stim.Test.theta.*ones(1,length(t));
        Sin_tst = Stim.Test.mag.*[cos(theta); sin(theta)];
    case 'Figure 8'
        theta = linspace(-pi/4, 3/4*pi, length(t));
        Sin_tst = [1.5*cos(2*theta); 1*cos(2*theta).*sin(2*theta)];
    case 'White Noise'
        for f = 1:Sim.nDim
            [Sin_tst(f,:),Amps(f,:)] =
genSignal(Stim.Test.FR.T,Stim.FR.dt,Stim.Test.rms,Stim.Test.bandwidth,S
tim.Training.randomSeed*pi*f);  %Increment random seed in deteministic
way across multiple dimensions when RandomSeed >0
            %pi multiple in randomSeed used to ensure different
amplitude coeff in generaiton of random training and test signals
            clear Amps
        end
end

LIFinit.V = zeros(1,N);
LIFinit.EndRefPeriod = zeros(1,N);
```

```
LIFinit.jitterSig = [];
LIFinit.Radapt = Sim.Radapt;

ndtperBin = Sim.FR.tRateInt/Sim.FR.dt;

% STATIC FILTER INITIALIZATIONS
sx = [];
sx.A = AdaptiveFilter.static.Asu(1,1);
sx.B = 0;
sx.H = H(:,1);
sx.Q = AdaptiveFilter.static.Wsu(1,1);
sx.R = R(:,1:Nusable);

sx.P = P;
sx.u = 0;
sxscale = 1;

sy = [];
sy.A = AdaptiveFilter.static.Asu(2,2);
sy.B = 0;
sy.H = H(:,2);
sy.Q = AdaptiveFilter.static.Wsu(2,2);
sy.R = R(:,Nusable+1:end);


sy.P = P;
sy.u = 0;
syscale = 1;

SUrateResp = zeros(N, TestLength/Sim.FR.tRateInt);
sSUCenters = zeros(2, TestLength/Sim.FR.tRateInt);

Radapt = Sim.Radapt;

for cnt=1:TestLength/Sim.FR.tRateInt
    % GENERATE THE FIRING RATES FOR THE TEST SIGNAL

    [SUrateResp(:,cnt), sSUCenters(:,cnt), LIFinit, Radapt] =
GetNeuronFiringRatesIterative(Sim, Stim, Sin_tst(:,((cnt-
1)*ndtperBin)+1:cnt*(ndtperBin)), LIFinit, N, 1, Radapt);

    % INTRODUCTION OF NONSTATIONARITY
    if flag == 1
        SUrateResp(Nusable:-1:indchangedpopLoss,cnt) = 0;
    elseif flag == 2
        if Sim.nchangedpop == Sim.neuronsEachTime
            SUrateResp(1:Sim.neuronsEachTime, cnt:end) =
SUrateResp(Sim.neuronsEachTime+1:end, cnt:end);
        else
            SUrateResp(1:indchangedpopNusable, cnt:end) =
SUrateResp(Nusable+1:indchangedpopReplace, cnt:end);
        end
    end
```

```
    % STATIC FILTER
    sx(end).z = SUrateResp(1:Nusable,cnt);
    if(cnt == 1) % PROVIDE INITIAL BEST ESTIMATES FOR THE KALMAN FILTER
        sx.x = sSUCenters(1,1);
        sy.x = sSUCenters(2,1);
    end
    [sx(end+1), K] = kalmanf(sx(end),sxscale);
    sy(end).z = SUrateResp(1:Nusable,cnt);
    [sy(end+1), K] = kalmanf(sy(end),syscale);

end

for cnt=1:TestLength/Sim.FR.tRateInt-1 % for extracting the array from
the struct
    sxplot(cnt)=sx(cnt+1).x;
    syplot(cnt)=sy(cnt+1).x;
end

% PLOTS

figure
hold on
grid on
plot
(sSUCenters(1,1:TestLength/Sim.FR.tRateInt),sSUCenters(2,1:TestLength/S
im.FR.tRateInt),'r', 'LineWidth', 2);
plot (sxplot(1:TestLength/Sim.FR.tRateInt-
1),syplot(1:TestLength/Sim.FR.tRateInt-1),'m--', 'LineWidth', 2);
axis([-2 2 -1 1])
set(gca, 'FontSize', 14), legend('Snapshot Test Signal', 'Filter
Reconstruction');
xlabel('X velocity V_x', 'FontSize', 14)
ylabel('Y velocity V_y', 'FontSize', 14)
if(flag == 0)
    title (['Snapshot of performance - ',
num2str((cntt*Sim.FR.tRateInt)), ' seconds'], 'FontSize', 16)
else
    title (['Snapshot of performance with nonstationarity - ',
num2str(round(cntt*Sim.FR.tRateInt)), ' seconds'], 'FontSize', 16)
end
drawnow;
```

kalmanf.m

```
function [s, K] = kalmanf(s,scale)
% Modified from KALMANF VERSION 1.0, JUNE 30, 2004 BY Michael C. Kleder
% http://www.mathworks.com/matlabcentral/fileexchange/5377-learning-
the-kalman-filter
%
% [s, K] = kalmanf(s,scale)
% ---INPUTS---
% s is a struct that holds the state variables
% scale is the factor influencing the progression of the Kalman gain
%
% ---OUTPUTS---
% s is a struct that holds the state variables
% K holds the Kalman gain between function calls

% set defaults for absent fields:
if ~isfield(s,'x'); s.x=nan*z; end
if ~isfield(s,'P'); s.P=nan; end
if ~isfield(s,'z'); error('Observation vector missing'); end
if ~isfield(s,'u'); s.u=0; end
if ~isfield(s,'A'); s.A=eye(length(x)); end
if ~isfield(s,'B'); s.B=0; end
if ~isfield(s,'Q'); s.Q=zeros(length(x)); end
if ~isfield(s,'R'); error('Observation covariance missing'); end
if ~isfield(s,'H'); s.H=eye(length(x)); end

if isnan(s.x)
    s.x = s.H\s.z;
    s.P = (s.H\s.R)/s.H';
end

% Discrete Kalman filter:

% Prediction for state vector and covariance:
s.x = s.A*s.x + s.B*s.u;
s.P = s.A * s.P * s.A' + s.Q;

% Compute Kalman gain factor:
K = s.P*s.H'*inv(s.H*s.P*s.H'+s.R);

% Correction based on observation:
s.x = s.x + scale*K*(s.z-s.H*s.x);%//a factor of 0.2 is introduced into
the gain -- Jan 02, 2007
s.P = s.P - scale*K*s.H*s.P;
```

adaptKalmanIterate.m

```
function [adaptiveKalman,AdaptiveFilter,statex,statey,Kall, K2xy] =
adaptKalmanIterate(adaptiveKalman,AdaptiveFilter,sSUCenters,SUrateResp,
cnt)

% An adaptive filter based on a cascaded Kalman filtering scheme
%
% [adaptiveKalman,AdaptiveFilter,statex,statey,Kall, K2xy] =
adaptKalmanIterate(adaptiveKalman,AdaptiveFilter,sSUCenters,SUrateResp,
cnt)
% ---INPUTS---
% adaptiveKalman is a struct that holds the state variables for each
Kalman
% filter.
% AdaptiveFilter is a Struct variable with variables specific to the
% adaptive filter.
% sSUCenters is a 2x1 vector that holds the averaged stimulus along two
% dimensions for each timestep.
% SUrateResp is a vector that holds the binned rates for each neuron
over the
% current timestep.
% cnt holds the value of the current bin timestep
%
% ---OUTPUTS---
% adaptiveKalman is a struct that holds the state variables for each
Kalman
% filter.
% AdaptiveFilter is a Struct variable with variables specific to the
% adaptive filter.
% statex holds the value of the decoded movement along the X-axis at
the current timestep
% statey holds the value of the decoded movement along the X-axis at
the current timestep
% Kall is a struct that holds the Kalman gains for the first and third
Kalman
% filters.
% K2xy is a struct that holds the Kalman gains for the second Kalman
% filter.

xscale = AdaptiveFilter.xscale;
yscale = AdaptiveFilter.yscale;

temp_x = adaptiveKalman.adaptfilt1x.x; % X AND Y VALUES FOR THE CURRENT
TIMESTEP TO BE FED INTO THE THIRD KALMAN
temp_y = adaptiveKalman.adaptfilt1y.x;

%################################
%For "window_size" timesteps...

adaptiveKalman.adaptfilt1x.z = SUrateResp;
[adaptiveKalman.adaptfilt1x, K1x] =
kalmanf(adaptiveKalman.adaptfilt1x,1);

adaptiveKalman.adaptfilt1y.z = SUrateResp;
```

```
[adaptiveKalman.adaptfilt1y, K1y] =
kalmanf(adaptiveKalman.adaptfilt1y,1);

AdaptiveFilter.xtrue = AdaptiveFilter.xtrue + sSUCenters(1,:);
AdaptiveFilter.ytrue = AdaptiveFilter.ytrue + sSUCenters(2,:);

AdaptiveFilter.xest=AdaptiveFilter.xest+adaptiveKalman.adaptfilt1x.x;
AdaptiveFilter.yest=AdaptiveFilter.yest+adaptiveKalman.adaptfilt1y.x;
AdaptiveFilter.zest=AdaptiveFilter.zest+adaptiveKalman.adaptfilt1x.z;

true_errorx = AdaptiveFilter.xtrue - AdaptiveFilter.xest;
true_errory = AdaptiveFilter.ytrue - AdaptiveFilter.yest;

%################################
%For calculating Q and H for the next time step
if (AdaptiveFilter.window_size_filter == 1 ||
mod(cnt,AdaptiveFilter.window_size_filter)==1)
    % FOR P
    if (cnt == 1 || (cnt-1)/AdaptiveFilter.window_size_filter == 1) %
INITIALIZE THE P VALUE THE VERY FIRST TIME
        adaptiveKalman.adaptfilt2x.P
=(adaptiveKalman.adaptfilt1x.x\adaptiveKalman.adaptfilt2x.R)/adaptiveKa
lman.adaptfilt1x.x'; %P = inv(H)*R*inv(H')
        adaptiveKalman.adaptfilt2y.P
=(adaptiveKalman.adaptfilt1y.x\adaptiveKalman.adaptfilt2y.R)/adaptiveKa
lman.adaptfilt1y.x'; %P = inv(H)*R*inv(H')
    end

    true_errorx = true_errorx/AdaptiveFilter.window_size_filter;
    true_errory = true_errory/AdaptiveFilter.window_size_filter;

    AdaptiveFilter.xest =
AdaptiveFilter.xest/AdaptiveFilter.window_size_filter;
    AdaptiveFilter.yest =
AdaptiveFilter.yest/AdaptiveFilter.window_size_filter;
    AdaptiveFilter.zest =
AdaptiveFilter.zest/AdaptiveFilter.window_size_filter;

    %         INCORPORATING THE TRUE ERROR INTO THE SIMULATION

    adaptiveKalman.adaptfilt2x.H = (AdaptiveFilter.xest +
true_errorx)';
    adaptiveKalman.adaptfilt2x.x = adaptiveKalman.adaptfilt1x.H';
    adaptiveKalman.adaptfilt2x.z = AdaptiveFilter.zest';

    [adaptiveKalman.adaptfilt2x, K2x] =
kalmanf(adaptiveKalman.adaptfilt2x,xscale);

    adaptiveKalman.adaptfilt2y.H = (AdaptiveFilter.yest +
true_errory)';
    adaptiveKalman.adaptfilt2y.x = adaptiveKalman.adaptfilt1y.H';
    adaptiveKalman.adaptfilt2y.z = AdaptiveFilter.zest';

    [adaptiveKalman.adaptfilt2y, K2y] =
kalmanf(adaptiveKalman.adaptfilt2y,yscale);
```

```
    AdaptiveFilter.xest=0;
    AdaptiveFilter.yest=0;
    AdaptiveFilter.zest=zeros(size(AdaptiveFilter.static.Hsu,1),1);
    AdaptiveFilter.xtrue=0;
    AdaptiveFilter.ytrue=0;
end

adaptiveKalman.adaptfilt3x.H = adaptiveKalman.adaptfilt2x.x';
adaptiveKalman.adaptfilt1x.H = adaptiveKalman.adaptfilt3x.H;

adaptiveKalman.adaptfilt3x.x = temp_x; % TO MAKE AN ESTIMATE FOR THE
SAME TIMESTEP USING THE NEWLY ADAPTED WEIGHTS
adaptiveKalman.adaptfilt3x.z = SUrateResp;

adaptiveKalman.adaptfilt3y.H = adaptiveKalman.adaptfilt2y.x';
adaptiveKalman.adaptfilt1y.H = adaptiveKalman.adaptfilt3y.H;
adaptiveKalman.adaptfilt3y.x = temp_y;
adaptiveKalman.adaptfilt3y.z = SUrateResp;

[adaptiveKalman.adaptfilt3x, K3x] =
kalmanf(adaptiveKalman.adaptfilt3x,1);

[adaptiveKalman.adaptfilt3y, K3y] =
kalmanf(adaptiveKalman.adaptfilt3y,1);

statex=adaptiveKalman.adaptfilt3x.x;
statey=adaptiveKalman.adaptfilt3y.x;

adaptiveKalman.adaptfilt1x.x=adaptiveKalman.adaptfilt3x.x;
adaptiveKalman.adaptfilt1y.x=adaptiveKalman.adaptfilt3y.x;

adaptiveKalman.adaptfilt1x.R=adaptiveKalman.adaptfilt3x.R;
adaptiveKalman.adaptfilt1y.R=adaptiveKalman.adaptfilt3y.R;

AdaptiveFilter.xscale = xscale;
AdaptiveFilter.yscale = yscale;

Kall = [K1x; K1y; K3x; K3y];
K2x = 1; K2y = 1; % Temporary place holder for K values
K2xy = [K2x; K2y];
```

Appendix B

To investigate the performance of the adaptive algorithm, a set of ten 400 second long sinusoid signals at frequencies every 0.1 Hz between 0.1 and 1 Hz was sampled at every 50 ms. A hundred randomized weights were assigned to each sinusoid and the composite signal obtained by the product of the weight matrix (100 x 10) with the sinusoid matrix (10 x 12000) was used to optimize the weights of the Kalman filter.

To simulate an effect similar to that observed in Chapter 6 for replacement of neurons, the order of the sinusoids was randomized and the resulting composite signal on multiplying the weights was used as the test signal input (measurement matrix z, see Chapter 3) to the Kalman filter. The adaptive filter showed a monotonic decrease in RMS errors when compared to a static Kalman filter that used the pre-optimized weight matrix as seen in the figure below.

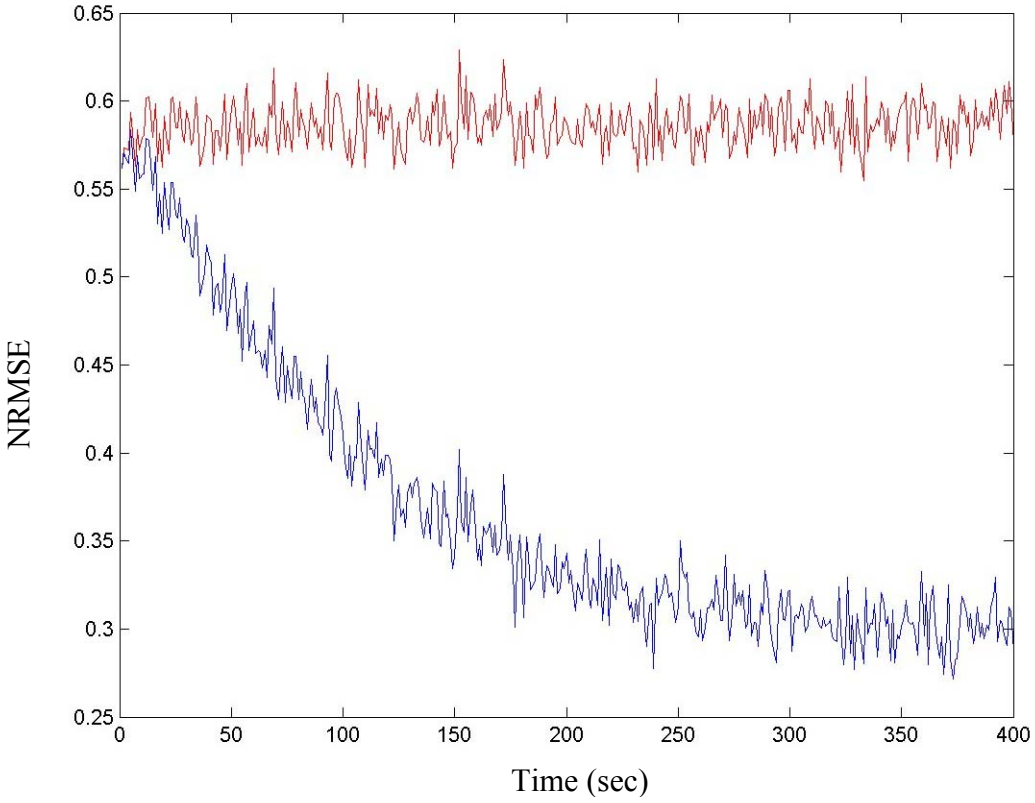


Figure A2.1: Normalized root mean square error (NRMSE) for a 400 second long composite of sinusoids. NRMSE is shown for the static Kalman (red) and adaptive Kalman (blue). Errors were computed over a 10 second non-overlapping window. The static Kalman filter was optimized to the initial order of the sinusoid waveforms. The order of the sinusoids was randomized at zero seconds in the plot shown above.