Particle Filtering Programmable Gate Array Architecture for Brain Machine Interfaces

A Dissertation
Submitted to
the Temple University Graduate Board

in Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

by
John M. Mountney
2011

Iyad Obeid co-advisor

Dennis Silage co-advisor

Marcus Sobel

Joseph Picone

Rahul Mangharam

ABSTRACT

Particle Filtering Programmable Gate Array Architecture for Brain Machine Interfaces

John M. Mountney

DOCTOR OF PHILOSOPHY

Temple University, 2011

Advisors: Iyad Obeid, PhD and Dennis Silage, PhD

Decoding algorithms for brain machine interfaces map neural firing times to the underlying biological output signal through dynamic tuning functions. In order to maintain an accurate estimate of the biological signal, the state of the tuning function parameters must be tracked simultaneously. The evolution of this system state is often estimated by an adaptive filter. Recent work demonstrates that the Bayesian auxiliary particle filter (BAPF) offers improved estimates of the system state and underlying output signal over existing techniques. Performance of the BAPF is evaluated under both ideal conditions and commonly encountered spike detection errors such as missed and false detections and missorted spikes.

However, this increase in neuronal signal decoding accuracy is at the expense of an increase in computational complexity. Real-time execution of

the BAPF algorithm for neural signals using a sequential processor becomes prohibitive as the number of particles and neurons in the observed ensemble becomes large. However, throughput is significantly increased by utilizing a parallel hardware architecture. This research describes a parallel processing architecture for implementing the BAPF in a field programmable gate array (FPGA) for real-time neural signal processing. While clock rates of FPGAs are inherently slower than those of conventional sequential processors, data rates are greatly enhanced when configured to simultaneously carry out computations in parallel.

ACKNOWLEDGEMENTS

I thank my advisors, Iyad Obeid and Dennis Silage, for their guidance and instruction throughout the course of my research. Their dedication to the advancement of education and technology is greatly appreciated. I also thank my committee members, Marc Sobel, Joseph Picone and Rahul Mangharam.

Without the constant support of my family this work would otherwise not be possible. My parents have been a constant encouragement and I will continue to do my best to make them proud. I would also like to thank the Temple University students and faculty for making my 10 years of education here rewarding and fun.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Encoding a biological signal or decoding a stimulus for an ensemble of neurons is the primary objective of the brain-machine interface (BMI)[1]. BMIs have had successful application in auditory and visual implants [2, 3] as well as controlling the movement of neuroprosthetics [4]. Regardless of the application, most BMIs are implemented by adaptive filtering algorithms that use the individual firing rates of neurons in the ensemble to estimate the underlying biological signal.

Whenever there is a requirement to interpret signals that result from operation in an environment of unknown statistics, adaptive signal processing provides a means of tracking the temporal evolution of system parameters. The use of adaptive filters offers signal processing capabilities that would otherwise not be possible and have been successfully applied in such diverse fields as digital communications, digital control, speech processing, radar and biomedical engineering [5].

Chapter 2 reviews linear adaptive algorithms such as the Weiner filter [6], least mean squares (LMS) algorithm [7] and the Kalman filter [8, 9] that have been employed to achieve pseudo electromyographic signal reconstruction. Nonlinear approaches to the estimation problem for BMIs are also investigated and include the extended Kalman filter [10], the unscented Kalman filter [11], point process adaptive filters [12, 13] and particle filters [14, 15].

It has been suggested that particle filters are better suited for neural signal processing than other estimation algorithms since they can be applied in non-Gaussian, non-stationary environments [16]. This work presents an application of the Bayesian auxiliary particle filter (BAPF) for estimating neural driving signals. It is demonstrated in Chapter 3 that the BAPF is capable of superior parameter estimation than other filtering techniques in terms of the error between the true and estimated signal. However, this increase in accuracy comes at the expense of computational complexity. For this reason, particle filters are not employed in real-time BMI technology.

In Chapter 4, a parallel hardware architecture is presented that significantly increases throughput over sequential processing. This hardware architecture is designed and verified using a field programmable gate array (FPGA). Even with clock rates slower than conventional processors, FPGAs can yield substantially higher throughput when configured to process signals in parallel. The reconfigurability of FPGAs also allows the device to be reprogrammed to support varying types of neurons and neural parameters as well as incorporating an array of particles.

Finally, Chapter 5 discusses the increased neural decoding accuracy of the BAPF over exiting methods. The real-time implementation of the BAPF for BMIs can be facilited by a parallel FPGA hardware architectures. Recomendations for future research are also addressed.

CHAPTER 2

Background

This chapter presents several linear and nonlinear decoding algorithms that have been applied recently in brain machine interface (BMI) engineering. The advantages and limitations of each algorithm are addressed. A comparison of some of these techniques is discussed in Chapter 3.

2.1 Wiener Filtering

The Weiner Filter (WF) is an optimal linear estimator in the mean squared error (MSE) sense that assumes a linear mapping between the system state or the model parameters and the input observations or the delayed spiking activity of individual neurons. Assuming knowledge of an invertible correlation matrix $\mathbf{R}$ of the input signal $\mathbf{u}(\mathrm{t})$ where

$$\mathbf{R} = \mathbf{E}[\mathbf{u}(t)\mathbf{u}(t)^T] \tag{2.1}$$

and the cross-correlation vector $\mathbf{p}$ between $\mathbf{u}$(t) and some desired response d(t)

$$\mathbf{p} = \mathbf{E}[\mathbf{u}(t)\,d(t)] \tag{2.2}$$

the optimal tap-weight vector $\mathbf{w} = [w_0 \cdots w_{M-1}]$ of the linear traversal filter can be expressed through the Wiener-Hopf Equations as

$$\mathbf{w} = \mathbf{R}^{-1}\mathbf{p} \tag{2.3}$$

The output y(t) of the filter is a weighted sum of the current and previous M inputs, which are stored in the registers of a tap-delay line of length M[5]. This output can be defined to be the stimulus or driving signal. The traversal filter block diagram is shown in Figure 2.1. It should be noted that the WF also assumes that the input signal is a stationary process, that is, the mean and variance do not change over time.

The WF has been successfully demonstrated to generate control of a robotic arm using the instantaneous firing rates of individual neurons in the primary, premotor and posterior parietal corticies of primates as the input to this linear filter [17]. Recordings from the primary and dorsal premotor corticies have been used as inputs to a WF to recreate center-out reaching tasks [18]. The

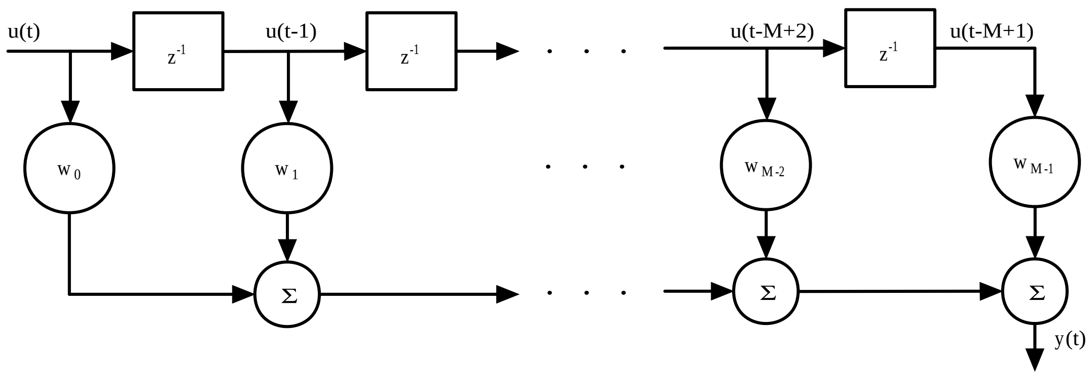

Figure 2.1: Block diagram of linear traversal filter

filter was also applied to reconstruct two dimensional cursor movements using electrode arrays implanted in paralyzed human participants [19].

When estimating the signal, it is often advantageous to observe a large number of neurons, which are believed to be correlated with the signal [20, 21]. This approach results in each neuron in the ensemble having its own WF with a unique weight vector that contributes to the overall output of the system. Depending on the number of delayed input samples M and the number of neurons in the ensemble K, inversion of the correlation matrix of dimension MK x MK during training periods becomes computationally intensive as M and K become large [22].

2.2 Least Mean Square

An alternative to the Weiner-Hopf equations is to use stochastic gradient-based algorithms for finding the optimal tap-weight vector $\mathbf{w}$. These algorithms converge to the optimum Weiner solution based on incoming data. The least-mean-square (LMS) algorithm developed by Widrow and Hoff [23] is one such algorithm. The LMS algorithm does not require any knowledge of the correlation matrix and avoids matrix inversion. It uses the estimation error $e(t)$ to update the filter coefficients in accordance with some learning rate ϵ. Estimates of the optimum tap-weight vector $\hat{\mathbf{w}}$ are obtained through three steps.

1. Compute the output

$$y(t) = \hat{\mathbf{w}}(t)^T \mathbf{u}(t) \tag{2.4}$$

2. Compute the estimation error

$$e(t) = d(t) - y(t) \tag{2.5}$$

3. Update the tap weights

$$\hat{\mathbf{w}}(t+1) = \hat{\mathbf{w}}(t) + \epsilon u(t) e(t) \tag{2.6}$$

The computational savings per iteration gives the LMS an advantage over the Weiner filter for real-time processing. However, if the correlation matrix of the input vector is ill conditioned (that is, the eigenvalue spread is large) the LMS algorithm produces excessive error. This translates into longer or interminable training sessions, which negates some of the desired properties of the LMS [5].

2.3 Kalman Filtering

A generalization of the Wiener solution for computing the optimal MSE, is the Kalman filter (KF). Using a state-space approach to the estimation problem, Kalman [24] was able to make improvements in that there is no assumption of a stationary environment. The basic KF can be applied to linear estimation problems. Here the system state vector $\mathbf{x}(t)$ to be estimated is modeled by Equation 2.7 as

$$\mathbf{x}(t+1) = \Phi(t)\mathbf{x}(t) + \mathbf{h}(t) \tag{2.7}$$

where Φ(t) is the known state transition matrix and $\mathbf{h}(t)$ is a zero mean Gaussian process with a diagonal covariance matrix $\mathbf{Q}_h(t)$. The measurement equation describing the observation vector $\mathbf{z}(t)$ is defined by Equation 2.8 to be

$$\mathbf{z}(t) = \mathbf{C}(t)\mathbf{x}(t) + \mathbf{v}(t) \tag{2.8}$$

with $\mathbf{C}(t)$ being a known measurement matrix and $\mathbf{v}(t)$ is also a zero-mean Gaussian process with diagonal covariance $\mathbf{Q}_v(t)$.The covariance matrix $\mathbf{P}(t)$

associated with the error $\mathbf{e}(t) = \mathbf{x}(t) - \hat{\mathbf{x}}(t)$ between the actual state and the estimated state $\hat{\mathbf{x}}(t)$ is expressed as

$$\mathbf{P}(t) = E[\mathbf{e}(t)\mathbf{e}(t)^T] = E[(\mathbf{x}(t) - \hat{\mathbf{x}}(t))(\mathbf{x}(t) - \hat{\mathbf{x}}(t))^T] \tag{2.9}$$

In order to use $\mathbf{z}(t)$ to achieve the best possible $\hat{\mathbf{x}}(t)$, Equation 2.10 is used [25, 26, 27]

$$\hat{\mathbf{x}}(t) = \hat{\mathbf{x}}(t-1) + \mathbf{K}(t)(\mathbf{z}(t) - \mathbf{C}(t)\hat{\mathbf{x}}(t-1)) \tag{2.10}$$

The Kalman gain $\mathbf{K}$(t) minimizes the MSE and was found in [24] to be

$$\mathbf{K}(t) = \mathbf{P}(t)\mathbf{H}(t)^T(\mathbf{H}(t)\mathbf{P}(t)\mathbf{H}(t)^T + \mathbf{Q}_v(t))^{-1} \tag{2.11}$$

The Kalman filtering process is illustrated by the block diagram of Figure 2.2

It has been shown that it is feasible to acquire 2D cursor control by applying the KF to recordings from the motor cortex [28, 29, 30]. Additional support for using the KF in cursor control can be found in [31], where the algorithm was used in closed-loop decoding experiments. The KF was also utilized to decode neural activity responsible for arm movement in the motor cortex [32].

By incorporating some previous history of the state estimates, the Nth order KF offers improvement over the standard KF in performing reaching tasks [33]. The switching Kalman filter (SKF) [34] uses multiple KFs that operate in parallel to estimate hand kinematics with less MSE than the KF.

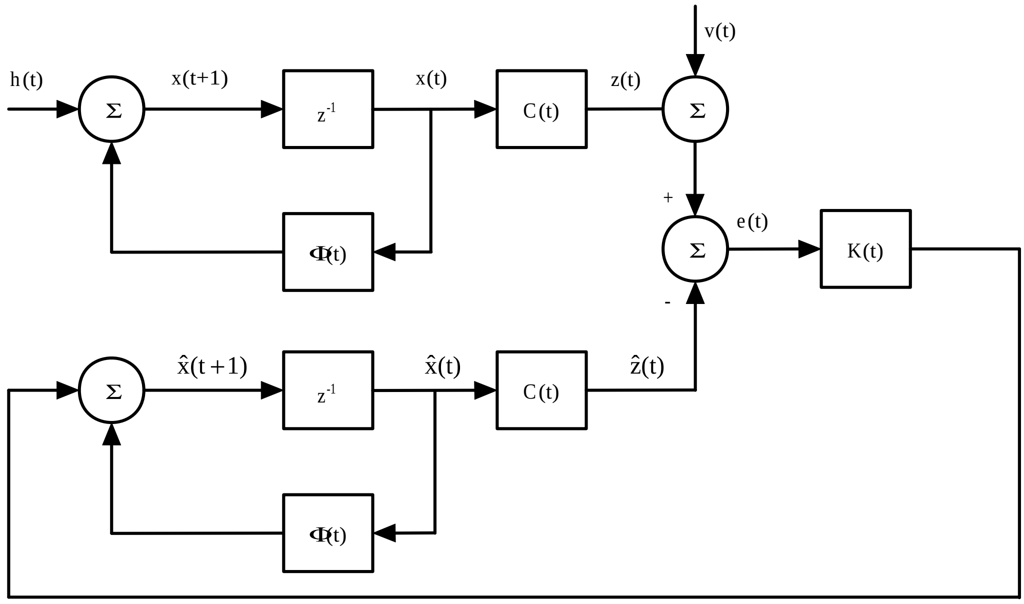

Figure 2.2: Signal flow graph of Kalman Filtering

The extended Kalman filter (EKF) can be applied to situations where the state transition and measurement matrices are nonlinear [35]. The EKF attempts to linearize the model by calculating the Jacobian of $\Phi(t)$ and $\mathbf{C}(t)$ around the estimated state. The EKF follows the same recursive structure described above except $\Phi(t)$ and $\mathbf{C}(t)$ are dependent on the current estimated state [36]. The first application of the EKF for neural signal processing was seen in [37], where control and tracking of spatiotemporal cortical activity are performed.

The unscented Kalman filter (UKF) is another nonlinear version of the KF [38]. Rather than trying to linearize the model through a first-order Taylor expansion as in the EKF, the UKF transforms a set of points and propagates them through the actual nonlinear function to eliminate the need to calculate any derivatives. This provides more accurate results while at the same time reducing some of the computationally intensive steps involved with the EKF.

An N^{th} order UKF was applied in [11] to accurately reconstruct reaching tasks in two rhesus macaques monkeys. Although the EKF and UKF offer a way to approximate nonlinear systems they are not optimal estimators [16]. Furthermore, they are susceptible to divergence. Error covariance is always under-approximated in stable filters. Since the Kalman gain is dependent on the error covariance matrix, it too is under-estimated. If the covariance is too small then $\mathbf{K}(t)$ tends to zero and no adaptation occurs [39]. Invertibility issues also arise when the number of unknown parameters to be estimated become large [40].

2.4 Neural Plasticity

Despite the initial success of both the linear and nonlinear estimators, the performance of these algorithms degrade over time due to the dynamic firing properties of individual neurons. Sensory experiences and control commands alter the strength of synapses between neurons. Based on how active they are during an experience, some synapses on a neuron grow stronger and others weaker. The pattern of synaptic strength changes represents a memory of the experience [41, 42]. Changes that occur in the organization of the brain and their spiking responses to various stimuli as a result of experience are referred to as neural plasticity [43].

Neural plasticity can be the result of environmental changes, learning, normal experience or brain injury. It has been shown that thinking, learning and acting actually change the brain's functional anatomy if not also its physical anatomy [44, 45]. However, when processing an ensemble of neural signals using the algorithms discussed in the previous sections, there is usually little or no effort to account for the dynamic properties of the individual neurons. This results in the system needing to be retrained often.

2.4.1 Dynamic Tuning Function

The firing rate of a neuron can typically be modeled using a tuning function. A neuron's tuning function describes how a neuron encodes information in response to a stimulus [46]. Each neuron has its own model parameters which comprise the state vector $\mathbf{x}(t)$ for that neuron. There are many different tuning function models used to describe the firing behavior specific types of neurons. The hippocampal place cell model is one example. The following equation can be used to define a Gaussian-shaped tuning function for a hippocampal place cell.

$$\lambda_(t) = exp\left\{\alpha_(t) - \frac{(s(t) - \mu(t))^2}{\xi(t)^2}\right\}, \tag{2.12}$$

Place cells fire when an animal is in a specific region $s(t)$ and are responsible for spatial mapping [47]. Equation 2.12 desribes one dimensional spatial mapping. The same model can be applied to two or three dimensions. Each neuron is *tuned* to have a maximum firing rate e^{α} at the center of its receptive field μ and responds to a range ξ of locations. Figure 2.3 shows the tuning function of a place cell modeled by Equation 2.12 with a state of $\mathbf{x}(t_0) = [\alpha = 3.5 \ \mu = 10 \ \xi = 10]^{\mathrm{T}}$. This means that when the animal is located at the receptive field center, $s(t) = 10$ cm, this particular neuron will fire at a maximum rate of $e^{\alpha} \approx 33$ Hz. As the animal deviates from the receptive field center in either direction, the firing rate for this neuron starts to decrease. When the animal is a sufficient distance from the receptive field center, one can expect that the neuron will stop firing depending on the width of the receptive field ξ.

However, plasticity results in neurons having dynamic state vectors. This means that a neuron will exhibit different firing properties as it evolves. This process is illustrated in Figure 2.4, where it can be seen how the same neuron responds very differently to the same stimulus over time.

The amount of variation the neurons exhibit dictates how often the system needs to be retrained or recalibrated. In order to maintain an accurate encoding/decoding process from a dynamic neuron, the evolution neural tuning parameters must be approximated as part of the system state vector. The point process adaptive filters and particle filtering algorithms

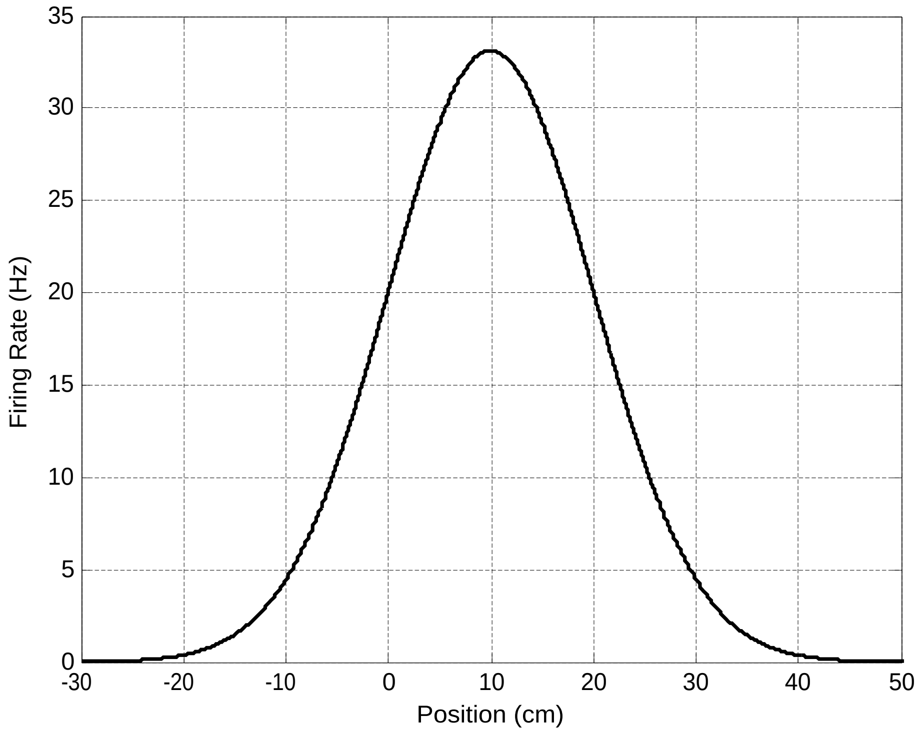

Figure 2.3: Tuning function for a single hippocampal place cell.

presented next describe how to track the time-varying parameters of a plastic neuron.

2.5 Steepest Descent Point Process Adaptive Filtering

The first signal processing algorithm is based on a steepest descent adaptive filtering methodology. It considers the observed neural spike trains as point

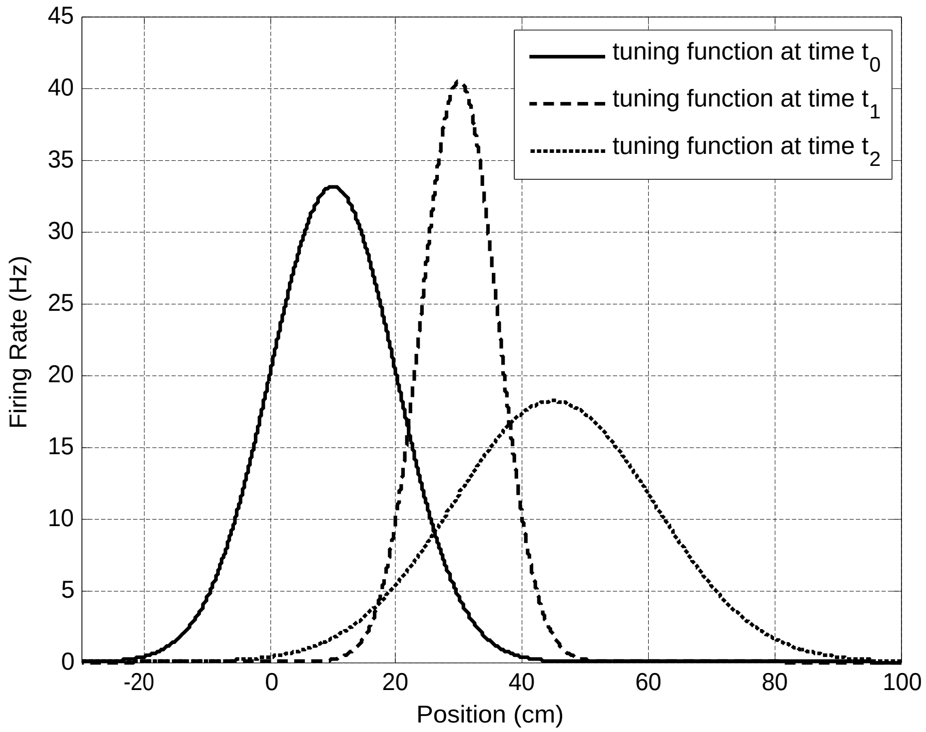

Figure 2.4: A simulated dynamic tuning function at three different instances of time.

processes that have a Poisson arrival rate. Using the instantaneous firing rate of a neuron, it is possible to estimate the dynamic tuning function parameters [12].

Over an observation interval $(0, \mathrm{T}]$, $N(t)$ is the counting process of the total number of spikes fired by a given neuron up to time t, where $0 < t \leq \mathrm{T}$. A random point process representing neural firing times can be characterized by its conditional intensity function $\lambda(t, s(t), \mathbf{x}(t), \mathrm{H}(t))$, where $s(t)$ is a vector representing the biological signal to which the neural system is tuned, $\mathbf{x}(t)$ is a vector representing the state of tuning function parameters for this neuron and $\mathrm{H}(t)$ denotes the history of the spiking process and the biological signal up to time t [48]. Given the spiking history and the model parameters, the instantaneous firing rate of a neuron can be described in terms of its spiking probability by Equation 2.13

$$\lambda(t|s(t), \mathbf{x}(t), \mathrm{H}(t)) = \overset{lim}{\Delta t \to 0} \frac{P(N(t+\Delta t) - N(t) = 1|s(t), \mathbf{x}(t), \mathrm{H}(t))}{\Delta t} \tag{2.13}$$

The probability of a neuron spiking over $[t, t + \Delta t)$ can be assumed to be $\lambda(t|s(t), \mathbf{x}(t), \mathrm{H}(t))\Delta t$, which generalizes the inhomogeneous Poisson process [49]. A discrete version of these functions can be obtained by dividing T into K individual time steps of size Δt. Each time step is then indexed by k with $k = 0, \ldots, K - 1$. If Δt is made small enough (on the order of milliseconds), then receiving more than one spike per sample interval generally will not occur and therefore, $\Delta N_k = N_k - N_{k-1} = 1$ if a spike occurs during the observation interval or 0 otherwise [50].

Using the conditional intensity function of Equation 2.13, an expression for the probability of observing a spike over Δt is

$$P(\Delta N_k) = [\lambda(t_k|\mathbf{x}_k, \mathrm{H}_k)\Delta t]^{\Delta N_k}[1 - \lambda(t_k|\mathbf{x}_k, \mathrm{H}_k)\Delta t]^{1-\Delta N_k} + o(\Delta t_k) \tag{2.14}$$

From [50], it was shown that for small enough values of Δt the conditional probability reduces to

$$P(\Delta N = 1|\mathbf{x}_k, \mathrm{H}_k) = exp(\Delta N_k log(\lambda(t|\mathbf{x}_k, \mathrm{H}_k)\Delta t) - \lambda(t_k|\mathbf{x}_k, \mathrm{H}_k)\Delta t) \quad (2.15)$$

Since the probability density of Equation 2.15 depends on the unknown parameter vector $\mathbf{x}_k$, the log of Equation 2.15 as a function of $\mathbf{x}_k$ given N_k is the sample path log likelihood and results in the instantaneous log likelihood defined as

$$l_k(\mathbf{x}_k) = log(\lambda(t|\mathbf{x}_k, \mathrm{H}_k))\Delta t - \lambda(t|\mathbf{x}_k, \mathrm{H}_k) \quad (2.16)$$

Equation 2.16 measures the instantaneous accrual of information from the neural spike train about the parameter vector $\mathbf{x}$ and was therefore used as the cost function in deriving a steepest descent point process adaptive filter [12].

The derivation of an adaptive filter using instantaneous steepest descent to estimate a parameter vector $\mathbf{x}$ takes the form

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_{k-1} - \epsilon \left[\frac{\partial J_x(\mathbf{x})}{\partial \mathbf{x}} \right]_{\mathbf{x}=\hat{\mathbf{x}}_{k-1}} \quad (2.17)$$

where $\hat{\mathbf{x}}_k$ is the parameter estimate at the k^{th} iteration of the algorithm computed from the previous estimate $\hat{\mathbf{x}}_{k-1}$, the step-size parameter ϵ and the cost function $J_k(x)$, typically chosen to be a quadratic function of $\mathbf{x}$ is the instantaneous log likelihood of a Gaussian process. Similarly, to derive an adaptive algorithm to estimate the time varying parameter vector of a point process, the cost function in Equation 2.17 is replaced by the instantaneous likelihood in Equation 2.16 to yield

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_{k-1} - \epsilon \left[\frac{\partial l_x(\mathbf{x})}{\partial \mathbf{x}} \right]_{\mathbf{x}=\hat{\mathbf{x}}_{k-1}} \tag{2.18}$$

A recursive update equation for implementing a steepest descent point process adaptive filter using the conditional intensity function as a method to define an error between the previously estimated and current states of the system is then given as

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_{k-1} - \epsilon \left[\frac{\partial log\lambda}{\partial \mathbf{x}} (\Delta N_k - \lambda \Delta t_k) \right]_{\mathbf{x}=\hat{\mathbf{x}}_{k-1}} \tag{2.19}$$

Although this filter does not include the encoded signal as part of the state space vector, it does provide a powerful technique to observe the influence of neural plasticity on the system model.

2.6 Stochastic State Point Process Adaptive Filtering

Based on the concept of steepest descent point process adaptive filtering, the signal processing algorithm was extended to also include a learning rate that is proportional to the error and is therefore dynamic [13]. Here a recursive expression for the posterior mean $\mathbf{x}_k$ and variance $\mathbf{W}_k$ of the system state is used to track dynamic neurons as well as estimate the driving signal. This process is well defined and the recursive update can be obtained using the following steps.

1. Given a state transition matrix $\mathbf{F}$, predict $\mathbf{x}_k$ based on $\mathbf{x}_{k-1}$:

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} + \eta_k \tag{2.20}$$

   where η_k is a white Gaussian process with covariance $\mathbf{Q}_k$

2. Update the variance:

$$\mathbf{W}_{k|k-1} = \mathbf{F}\mathbf{W}_{k-1}\mathbf{F}^T + \mathbf{Q}_k \tag{2.21}$$

3. Update the posterior variance in terms of the probability of firing λ defined in Equation 2.15:

$$\begin{aligned}(\mathbf{W}_{k|k})^{-1} \quad = \quad & (\mathbf{W}_{k|k-1})^{-1} + \left[\left(\frac{\partial log\lambda}{\partial \mathbf{x}_k}\right)[\lambda\Delta t]\left(\frac{\partial log\lambda}{\partial \mathbf{x}_k}\right)^T\right. \\ & \left. -(\Delta N_k - \lambda\Delta t)\left(\frac{\partial^2 log\lambda}{\partial \mathbf{x}_k \partial \mathbf{x}_k^T}\right)\right]_{x_{k-1}}\end{aligned} \tag{2.22}$$

4. Update the mean (state):

$$(\mathbf{x}_{k|k}) = (\mathbf{x}_{k|k-1}) + \mathbf{W}_{k|k}\left[\left(\frac{\partial log\lambda}{\partial \mathbf{x}_k}\right)^T (\Delta N_k - \lambda\Delta t)\right]_{x_{k-1}} \tag{2.23}$$

This algorithm has been shown to perform well in a known statistical environment [13]. Specifically, this methodology assumes the state vector evolves as a zero-mean white Gaussian process. This assumes that the system parameters evolve independently of each other which results in a diagonal covariance matrix **Q**. When the covariance matrix is not diagonal and parameters exhibit correlation, the filter becomes unstable and quickly diverges. Additionally, it was also found that when periods of neural inactivity are encountered, the filter would lose track of the correct estimates and was not able to 'lock on' when the neuron started to fire again [51]. An additional drawback of this algorithm is that it requires two matrix inversions in order to update the posterior mean.

2.7 Particle Filtering

Particle filters can be used to estimate the current state of both linear and nonlinear systems using numerical simulation methods that approximate the often difficult to solve integrals of the recursive Bayesian estimation problem [52]. If the integrals cannot be solved analytically, Monte Carlo integration can be used to provide discrete support to represent the posterior probability $p(N(t)|\mathbf{x}(t))$ as a set of randomly chosen weighted samples, or particles. These particles are drawn from a proposal density that is chosen to approximate the posterior probability. As the number of particles becomes large, the approximation converges to the optimal Bayesian estimate.

The estimates obtained by using particle filters are not selective, that is they do not favor local optima, as is the case with many least squares-based algorithms. As a consequence, they may provide estimates which are both accurate and robust [53].

The unknown state vector $\mathbf{x}(t)$ is estimated by a set of P particles. Each particle is comprised of a state estimate $\mathbf{x}^r(t)$ and associated weight $w^r(t)$, with $r = 0, \ldots, P-1$. Particle weights are assigned based on how likely the current observations $N(t)$ relate to the estimated state $\mathbf{x}^r(t)$. A filter estimate can be formed as the weighted sum of all particle estimates. The following procedure describes a single iteration of a particle filter signal processing algorithm.

The particle values are first simulated by drawing hyperparameters Θ^r from a proposal density π that closely approximates the current posterior distribution of the dynamic state. Updated particle estimates are then obtained by combining the previous particle estimate with the hyperparameters as

$$\mathbf{x}^r(t) = \mathbf{x}^r(t-1) + \Theta^r(t) \tag{2.24}$$

Each particle weight is then assigned a value based on how likely they are to represent the current observations $N(t)$ by the following equation

$$w^r(t) \propto w^r(t-1)\frac{p(N(t)|\mathbf{x}^r(t))p(\mathbf{x}^r(t)\mathbf{x}^r(t-1),\Theta(t))}{\pi(\mathbf{x}^r(t)|\mathbf{x}^r(t-1),N(t),\Theta(t))} \tag{2.25}$$

Equation 2.25 states that the weight of a particle at time t is proportional to its former weight at time $t-1$ and its new likelihood and inversely proportional to the probability given to it by the proposal density π. The particle weights are then normalized so that they sum to unity. Next, the particles are *resampled*. This process retains particle values with higher probabilities and discard those with low probability. Low weighted particles are then reassigned values from high weighted particles.

Various sampling techniques have been employed such as importance sampling (IS), sampling importance sampling (SIS) and sampling importance resampling (SIR) [52] in an attempt to avoid the effects of degeneracy. Degeneracy occurs when all but a few particle weights are arbitrarily close to zero. This results in most of the particles providing no viable information and the filter output becomes based on just a few particle estimates. Once the particles have been resampled, the state estimate $\mathbf{x}(t)$ can be computed as weighted sum of the particle values as

$$\hat{\mathbf{x}}(t) = \sum_{r=1}^{P} w^r(t)\mathbf{x}^r(t) \tag{2.26}$$

Particle filters have been shown to be a powerful algorithm for estimating parameters in neural firing state models. Decoding a neuron's response to a wind stimulus was successfully simulated in [54]. Here a simple exponential model was used to represent the firing rate of a single neuron in the sensory system of a cricket. This was further improved in [55], where the tuning function parameters are also estimated as part of the state vector to account for neural plasticity.

The particle filter was found to outperform the Kalman filter estimate in

[56]. Here, 185 neuron channels are used as the observations in estimating 2D hand kinematics in a primate reaching task. This work did not however take into account that the tuning functions are dynamic and no effort was made to track the system parameters.

2.8 Bayesian Auxiliary Particle Filtering

A major deficiency in particle filtering is the effect that outliers or unexpected observations have on filter performance [57]. Auxiliary particle filter signal processing algorithms address this problem by performing resampling at time $t-1$ using the current observation at time t before updating the particles. More specifically, the Bayesian auxiliary particle filter (BAPF) introduced by Liu and West [58] uses a two-stage weight update procedure. The first stage weights $g^r(t)$ are computed as an intermediate step used for the resampling process. The second stage weights $w^r(t)$ are used to compute the state estimate as a weighted sum. The following procedure describes a single iteration of the BAPF signal processing algorithm.

A first set of hyperparameters Θ_1^r is drawn from a proposal density π_1. Just as in Equation 2.24 the hyperparameters are combined with the previous state estimate according to

$$\hat{\mathbf{x}}^r(t) = \mathbf{x}^r(t-1) + \Theta_1^r(t) \tag{2.27}$$

Particle first stage weights are assigned a value based on their previous second stage weights and current likelihood as

$$g^r(t) \propto w^r(t-1)p(N(t)|\hat{\mathbf{x}}^r(t)) \tag{2.28}$$

Particle weights are then normalized and resampled according to $g^r(t)$. Next, a new set of hyperparameters Θ_2^r are drawn from a second proposal density π_2 and combined with the resampled particle estimates

$$\mathbf{x}^r(t) = \hat{\mathbf{x}}^r(t) + \Theta_2^r \tag{2.29}$$

Particles are then assigned second stage weights as a ratio of the particle's resampled likelihood and original likelihood.

$$w^r(t) \propto \frac{p(N(t)|\hat{\mathbf{x}}^r(t))}{p(N(t)|\mathbf{x}^r(t))} \tag{2.30}$$

A final estimate is computed as a weighted sum using the second stage weights.

$$\chi(t) = \sum_{r=1}^{P} w^r(t)\mathbf{x}^r(t) \tag{2.31}$$

Successful application of the BAPF was demonstrated in [51] where the parameters of a Gaussian shaped tuning function for a single neuron are estimated using the driving signal and firing times as the observations. It was compared to the stochastic state point process adaptive filtering algorithm described and was shown to result in a reduced MSE. Chapter 3 demonstrates how the BAPF can be extended to include the driving signal as part of the estimated state vector and exemplifies the superior estimation capabilities of the BAPF over other estimators.

CHAPTER 3

Simulation and Verification

This Chapter demonstrates the superior decoding ability of the Bayesian auxiliary particle filter (BAPF). The BAPF is first used to predict animal position while simultaneously compensating for neural plasticity by observing simulated neuronal activity. Here the BAPF tracks two parameters, 1) the driving signal $s(t)$ common to the observed neurons and 2) the receptive field center $\mu_j(t)$, in each of K hippocampal place cell neurons whose firing rates are described by Equation 2.12 from Section 2.4.1. The technique is assessed by simulating neurons that are tuned to the position of a virtual mouse following a random walk trajectory along a track. Reconstructed mouse positions are compared to the true mouse positions in Section 3.4 and results are quantified using both the mean squared error metric and confidence intervals.

An investigation of the BAPF to reconstruct neural signals in cases where the simulated spike train had been corrupted by varying degrees and types of spike detection/sorting errors are also presented. Performance of the algorithm is assessed and compared against the linear Wiener filter and the sampling importance resampling (SIR) particle filter. Here it is shown that the BAPF is a more accurate and robust decoding algorithm than both the Wiener and SIR particle filter in terms of mean square error (MSE).

In Section 3.5, the BAPF is employed to reconstruct elbow angle from neural firing times acquired through micro electrode recordings. The experimental data was recorded from the M1 area of primary motor cortex in an adult rhesus monkey. The BAPF decoding performance is compared to the Kalman filter described in Section 2.3 using MSE as the metric.

3.1 Neural Firing Model

Although the methods presented here are applicable to any number of neural tuning function models, the Gaussian hippocampal place cell model is used as a prototypical example. The term $\lambda_j(t)$ represents the instantaneous firing rate of the j^{th} neuron:

$$\lambda_j(t) = exp\left\{\alpha_j(t) - \frac{(s(t) - \mu_j(t))^2}{\xi_j(t)^2}\right\},\ j = 1, \ldots, K \tag{3.1}$$

In this model, the firing rate of each neuron is controlled by the driving signal $s(t)$ and three parameters, $\alpha_j(t)$, $\mu_j(t)$ and $\xi_j(t)$.

The parametric quantity $exp\{\alpha_j(t)\}$ characterizes the maximum firing rate of neuron j and can only be achieved when the driving signal is at the center of the j^{th} receptive field $\mu_j(t)$. The parameter $\xi_j(t)$ characterizes the width of the neuronal receptive field, with larger values of ξ signifying neurons that are responsive to a broader range of driving signal values. In order to make the neural tracking process more robust, the proposed Bayesian auxiliary particle filter here simplifies the model described in Equation 3.1 by assuming that the parameters α and ξ are constant over time and the same for all K neurons. Equation 3.1 therefore simplifies to:

$$\lambda_j(t) = exp\left\{\alpha - \frac{(s(t) - \mu_j(t))^2}{\xi^2}\right\},\ j = 1, \ldots, K \tag{3.2}$$

Note that these simplifications apply only to the Bayesian auxiliary particle filter.

This assumption is based on prior work concerned with Bayesian auxiliary particle filter tracking [51]. Specifically, it was noted that if the maximum firing rate for a typical hippocampal place cell does not exceed 50 spikes per second [59], then $\alpha \leq ln(50)$.

Since α has such a small range of viable values, it is hypothesized here that assigning it to a pre-selected a priori constant would introduce only modest error. By eliminating the need to track α, gross errors in estimating $s(t)$ are avoided since even small over-estimates of α can result in biologically impossible firing rates.

Similarly, the parameter ξ measuring the receptive field width is modeled as a constant because precise knowledge of its true value was not necessary for accurate tracking of $s(t)$, even though a wide range of values is biologically plausible [60]. The unknown state parameters $s(t)$ and $\mu_j(t)$ are estimated using only the observed spike firings $N_j(t)$.

Neural spikes are assumed to arrive according to an inhomogeneous Poisson process, meaning that for a given time interval Δt, the expected number of spikes in the j^{th} neuron is $\lambda_j \Delta t$. Given Poisson firing rates and the current state estimate of the r^{th} particle $\hat{\mathbf{x}}^r(t)$, the likelihood equations become

$$p(N(t)|\hat{\mathbf{x}}^r(t)) = \frac{(\lambda \Delta t)^{\Delta N(t)} e^{-\lambda \Delta t}}{\Delta N(t)!} \tag{3.3}$$

For the purposes of this work, the time intervals are assumed to be sufficiently small so that there can be at most one spike arrival per interval (i.e. $N_j(t) =$ 0 or 1). The conditional probability for neuron j becomes

$$p(N_j(t) = 1) = \lambda_j \Delta t \; exp\left\{-\lambda_j(t)\Delta t\right\} \tag{3.4}$$

$$p(N_j(t) = 0) = exp\left\{-\lambda_j(t)\Delta t\right\} \tag{3.5}$$

3.2 Time Bins

The neural firing variable $N_j(t)$ is assumed to be observed at discrete time intervals of Δt seconds, i.e. t = $\Delta t,\ 2\Delta t,\ \ldots,\ T\Delta t$. To further facilitate the execution of the Bayesian auxiliary particle filter, time bins are incorporated $B_1, \ldots, B_M$, each comprised of b consecutive time steps such that each bin is of duration $b\Delta t$ and that the total number of bins is $M = T/b$.

The state parameters $s(t)$ and $\mu_j(t)$ are assumed to be constant over each bin, implying that the time variable can be redefined as th index bin number instead of the time step. Specifically, if $t = 1, \ldots, T$ is replaced with $t = 1, \ldots, M$ without loss of generality. The notation $N_j(t)$ will hereafter refer to the number of firings in bin t with $t = 1, \ldots, M$.

3.3 Methods

In order to verify the accuracy with which the BAPF can decode a biological signal, a series of simulations are executed in which a virtual mouse moved back and forth along a track of length 300 cm. The time-varying parameters of an ensemble of hippocampal place cells and the mouse position are estimated using the BAPF. The only observed parameters are the neural firing times. The mouse position $s(t)$ is simulated as a random walk with the change in position from one time step to the next given by a normally distributed zero-mean random variable. Each trial simulates 30 seconds of neural behavior with a sampling period of $\Delta t = 2$ ms. The BAPF is compared to the linear Weiner filter and standard sampling importance resampling particle filter described in Section 3.3.3.

3.3.1 Neural Signal Simulation

Each simulation begins by defining a random walk trajectory and the three parameters ($\alpha_j(t)$, $\mu_j(t)$, and $\xi_j(t)$) for each of the K hippocampal place cells. For each neuron, $\alpha_j(t)$ is varied at random such that the maximum firing rate varies linearly over the 30 second simulation up to a maximum of 50 spikes/second. Likewise, the spatial receptive field for each neuron $\xi_j(t)$ is also chosen at random, varying linearly over the range $10 \leq \xi_j(t) \leq 20$.

The receptive field centers $\mu_j(t)$ of each of the neurons are defined to be linear functions of time with initial values uniformly distributed on the range $-50 \leq \mu_j(0) \leq 350$ cm to simulate observable neurons that fire in response to a wider range of values outside the 300 cm track. These receptive field centers had rates of change selected at random over the range -0.5 to $+0.5$ cm/s to incorporate the affect of neural plasticity and a dynamic tuning function. Using these simulated neuron parameter values, the instantaneous firing rate $\lambda_j(t)$ is calculated for each neuron. Finally, spike trains are generated using a Poisson random process with $\lambda_j(t)\Delta t$ expected firings per time step. A maximum of one firing is allowed per time step. Since $\Delta t = 2$ msec, it is not necessary to enforce a minimum refractory period between spikes.

3.3.2 Filter Parameters

As stated in Section 3.1, several assumptions are made in the configuration of the BAPF. These assumptions served the dual purpose of reducing computational complexity while also making the filter more robust. The BAPF assumed that the neural parameters α_j and $\xi_j(t)$ are the same constant for all K neurons with $\alpha = 3.5$, which keeps the maximum estimated firing rate less than 50, and $\xi = 12$, which is an average receptive field width of the simulated neurons. This assumption takes advantage of the fact that precise knowledge of these two parameters is not essential for accurate estimation of the driving signal $s(t)$.

It is also assumed that the BAPF could be initialized using estimates of the true values of $\mu_j(0)$ and $s(0)$ within $\pm 5\%$ error. This assumption is based on an earlier work in which it is determined that the BAPF can capture place cell parameters to within ± 5 cm during a brief training period in which both the driving signal $s(t)$ and the neural firings $N_j(t)$ are observed [51].

As described in Section 3.2, data is grouped into non-overlapping bins of $b = 25$ samples (duration 50ms). The receptive field centers $\mu_j(t)$ and the mouse position $s(t)$ are assumed to be constant (i.e. stationary) over the duration of each bin. In all cases, $P = 100$ particles are used to estimate the state parameters. It is assumed that the mouse position $s(t)$ varied substantially faster than the receptive field centers $\mu_j(t)$. Consequently, the standard deviations σ characterizing mouse position are made an order of magnitude larger than their counterparts.

The following describes the specific implementation of the BAPF for estimating neural driving signals according to the algorithm outlined in Section 2.8. These computational steps are described in Sections 4.2.3 through 4.2.10. Note: $\mathcal{N}(0, \sigma)$ refers to a normal random variable with zero mean and standard deviation σ.

Step 1: Random Sampling

$$\Theta_1^r \sim \mathcal{N}(0, \sigma_1) \; for \; r = 0, \ldots, P-1 \tag{3.6}$$

In order to be able to accurately estimate biological signals whose amplitudes may vary significantly with time, it is important that the standard deviations of the proposal densities be large enough so that the random samples capture the dynamics of the signal. In this experiment, it was found that $\sigma_1 = 1$ for the $s(t)$ hyperparameter and $\sigma_1 = 0.1$ for the $\mu_j(t)$ hyperparameters were sufficient values.

Step 2: Add Randomness to Particle Estimates

$$\hat{\mathbf{x}}^r(t) = \mathbf{x}^r(t-1) + \Theta_1^r \tag{3.7}$$

Step 3: Compute the first stage weights

$$g^r(t) \propto w^r(t-1)p(N(t)|\hat{\mathbf{x}}^r(t)) \tag{3.8}$$

When Poisson firing rates are assumed, the posterior density becomes

$$p(N(t)|x^r(t)) = \frac{(\lambda \Delta t)^{\Delta N(t)} e^{-\lambda \Delta t}}{\Delta N(t)!} \tag{3.9}$$

Step 4: Resample Particles According to the First Stage Weights

'Each particle's estimate is replaced by a new estimate drawn randomly from the set of all P particles. The probability of drawing particle r is given by the normalized weights.

Step 5: Random Sampling

$$\Theta^r \sim \mathcal{N}(0, \sigma_2), \tag{3.10}$$

After resampling all particles, it is assumed that the higher weighted (more correct) particles have been replicated. It would therefore, not be favorable to draw the second set of hyperparameters from the same distributions as the first set. This would result in particle estimates deviating to far from the current resampled estimates. Rather here, the standard deviations of the proposal densities are reduced so that they provide a finer resolution of current resampled estimates. In this instance $\sigma_2 = 0.1$ for the $s(t)$ hyperparameter and $\sigma_1 = 0.01$ for the $\mu_j(t)$ hyperparameters.

Step 6: Add Randomness to Resampled Particles

$$\mathbf{x}^r(t) = \hat{\mathbf{x}}^r(t) + \Theta_2^r \tag{3.11}$$

Step 7: Compute the Second Stage Weights

$$w^r(t) \propto \frac{p(N(t)|\mathbf{x}^r(t))}{p(N(t)|\hat{\mathbf{x}}^r(t)} \tag{3.12}$$

Computing the second stage weights determines how well a particular particle was resampled. If the likelihood of the resampled state is less than the likelihood of the initial state, a smaller value for $w^r(t)$ will be produced than if the likelihood of the resampled state was larger than the initial likelihood. Therefore, if a particle's estimate is replaced with a less likely estimate during the resampling process, its contribution to the final weighted estimate will be reduced.

Step 8: Compute a Weighted Estimate

$$\chi(t) = \sum_{r=0}^{P-1} w^r(t)\mathbf{x}^r(t) \tag{3.13}$$

3.3.3 Sampling Importance Resampling Particle Filter

The Sampling Importance Resampling (SIR) particle filter [54, 55] is used as a benchmark for the BAPF neural decoder. SIR filters are sequentially configured at each stage using weights which are proportional to a product of the former weights and the current prior and likelihoods. These newly calculated weights are used to simulate particles intended to estimate the signal at the new stage. SIR filters do not employ hyperparameters or any of the auxiliary features characteristic of the BAPF signal processing algorithm. Studies indicate that SIR filters frequently fail to be robust or adapt to signal changes [57]. This is a consequence of the way in which their weights are calculated. The results here show that BAPF filters consistently out-perform their SIR counterparts.

3.3.4 Wiener Filtering

The performance of the auxiliary particle filters is also compared against benchmark Wiener filters which have been used in real-time BMI applications [20]. Wiener filtering is applied to exactly the same artificial data sets that are analyzed with the auxiliary particle filter. The Wiener filter estimate of mouse position, $\hat{u}(t)$, is given by:

$$\hat{u}(t) = \gamma + \sum_{j=1}^{K} \sum_{\tau=-d}^{0} N_j(t-\tau)\ \rho_{j,\tau} \tag{3.14}$$

Here, $N_j(t-\tau)$ refers to the number of spikes of the j^{th} neuron during the τ^{th} prior bin. Spike bin rates are weighted by $\rho_{j,\tau}$.

These weights and the offset γ are determined using an optimal linear least squares fit between $N_j(t-\tau)$ and $s(t)$ that is calculated using a training subset of the data. As with the auxiliary particle filter, the Wiener filter is implemented using 50 msec time bins. The number of lagged bins is $d = 20$, hence the Wiener filter operates over the preceding one second of data. Each 30 second simulated data set is divided into two 15 second intervals. The first interval is used to train the Wiener filter weights. The second interval is used to test those weights in predicting $s(t)$. Owing to its observed poor performance on this particular data set, the Wiener filter results are only presented as a benchmark reference in Figure 3.1.

3.3.5 Robustness Testing

The robustness of the BAPF is tested by using noisy spike observations for tracking mouse position. Three common neural signal processing errors are tested: missed spike detections, falsely detected spikes and incorrectly sorted spikes. Errors of this nature have been shown to markedly degrade the theoretical information content of the encoded neural signal [61]. In all three cases, spike times are generated as described in Section 3.3.1.

For trials involving missed spike detections, a fraction of the detected spikes are summarily deleted. Likewise, for trials involving falsely detected spikes, a fraction of the spike-free time steps are assigned spikes. For trials involving spike sorting errors, neurons are grouped together as sets of two. Then, for the fraction of the times when a spike occurred in one neuron but not the other in the set, the spike is transferred from the given neuron to the other in the set. In all tests of robustness, $K = 50$ neurons are used.

3.4 Evaluation of the Bayesian Auxiliary Particle Filter

The Bayesian auxiliary particle filter is evaluated under five experimental conditions: two with ideal noiseless spike trains and three with various types of spike noise. Figure 3.1 shows signal reconstruction mean square error (MSE) using ideal (i.e. noiseless) spike trains as a function of the number of neurons K included in the simulation.

Each point in Figure 3.1 represents an average over 100 simulated data sets; in each data set, all simulated parameters (i.e. $\alpha_j(t)$, $\mu_j(t)$, $\xi_j(t)$, $s(t)$, and $N_j(t)$) are completely re-randomized as specified in Section 3.3.1. Averaging is performed over 100 completely different data sets for each test case. The differences between the true mouse trajectories and their reconstructions are quantified using the MSE metric.

Figure 3.2 demonstrates the superior estimation capabilities of the BAPF over the SIR particle filter. In this 30 second simulated experiment, 50 dynamic neurons are observed. The animal position and receptive field centers are estimated by both filters using 100 particles. Due to the perceived accuracy of

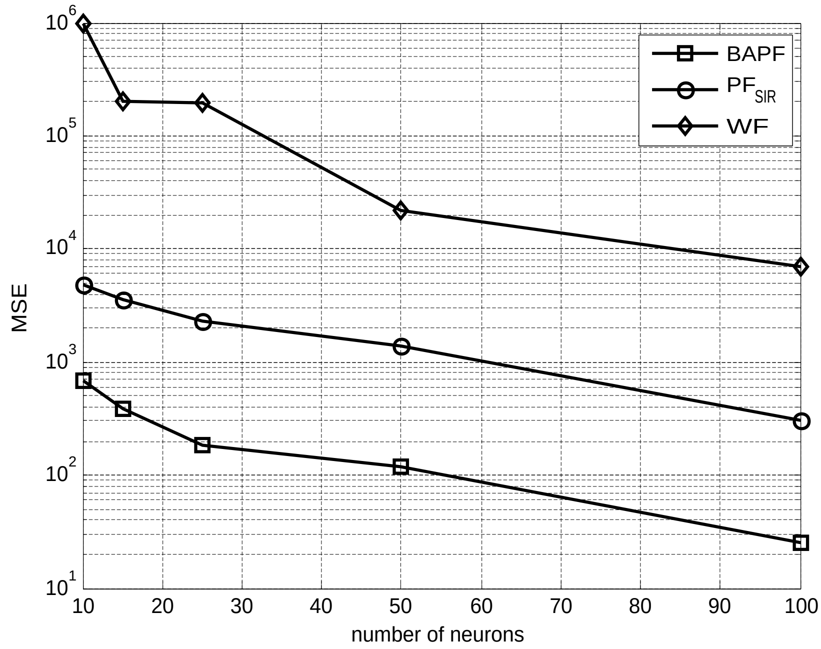

Figure 3.1: Comparison of MSE vs number of neurons for the SIR particle filter and BAPF

the BAPF, an expanded 4 second representation in Figure 3.2 is required to exaggerate the difference on an expanded scale between the BAPF estimate and the true position.

Figure 3.3 compares neural reconstruction by the BAPF and SIR particle filters for a single typical mouse trajectory using ideal noiseless spikes. The results are expressed using 95% confidence intervals. Note that the BAPF confidence interval (white) is nearly identical to the actual mouse trajectory (black) and is hard to discern in Figure 3.3.

The confidence interval for the BAPF is substantially narrower than its SIR particle filter counterpart, indicating superior tracking of both mouse position

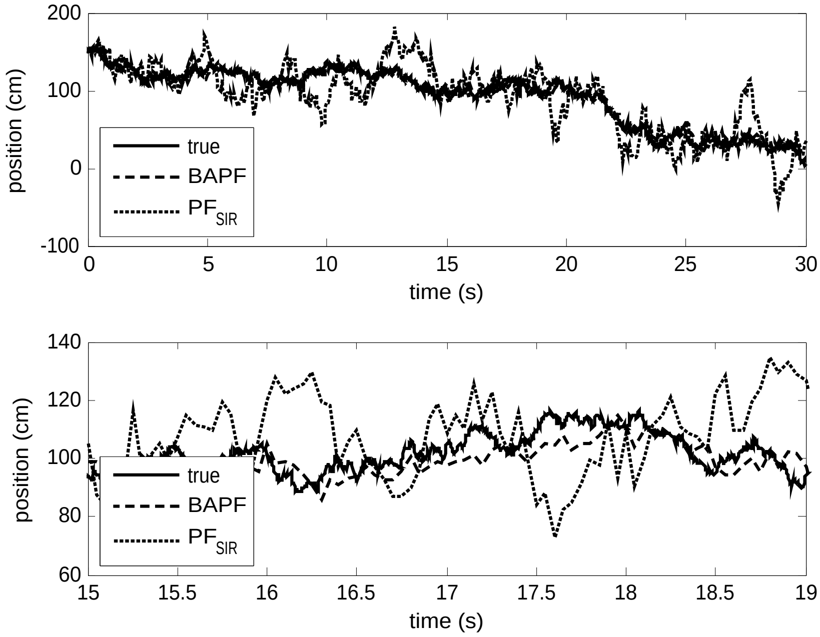

Figure 3.2: Top:Position estimates for the BAPF and SIR particle filter. Bottom: 4 second window of position estimates.

and the dynamic neural firing parameters. In contrast to Figure 3.1, in which each data point represents the filter performance averaged over 100 data sets, the results in Figure 3.3 are constructed from 100 simulations of a single data set. Thus, the results in Figure 3.3 illustrate that, unlike the SIR particle filter, the BAPF is consistently able to reconstruct the mouse trajectory.

To demonstrate the how the number of particles used to estimate $s(t)$ can affect the estimation error, the number of particles was varied from 10 to 100 while the number of neurons was held constant at 50. The MSE is plotted versus the number of particles in Figure 3.4. Although the accuracy of neural decoding improves as the number of particles increases, it is important

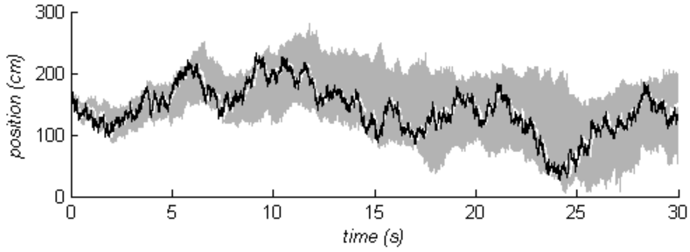

Figure 3.3: Mouse trajectory reconstruction (expressed using confidence intervals) for a typical random walk using K = 50 neurons. [Black] actual mouse trajectory; [White] BAPF confidence interval; [Gray] SIR particle filter confidence interval. The BAPF confidence interval is nearly indistinguishable from the actual mouse trajectory.

to notice here that the acquired increase in accuracy approaches a limit as the number of particles becomes large. It is therefore important to set an acceptable level of MSE before selecting the number of particles to be used in the estimation process since those additional particles may not provide a significant increases in decoding accuracy.

In Figure 3.5, a percentage of the simulated spikes have been deleted. Commonly known as missed detections or false negatives, this scenario arises when action potential waveforms are too small to trigger the detection threshold. The results demonstrate that the BAPF outperformed the SIR particle filter by an order of magnitude for all the missed detection percentages.

Figure 3.6 illustrates BAPF and SIR particle filter performance when false positive spikes are injected into the spike train at various rates. False positives arise when signal noise is misinterpreted by the neural processor as a spike. As

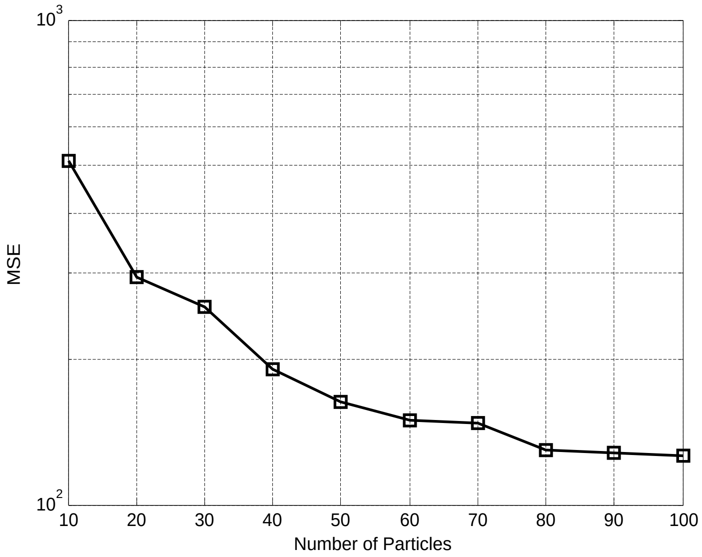

Figure 3.4: MSE vs Number of Particles:

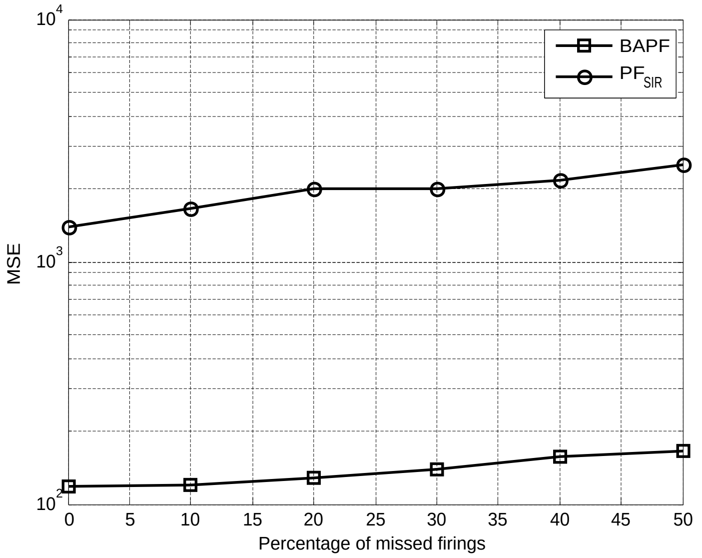

Figure 3.5: MSE vs percentage of missed detections for a 50 neuron ensemble.

with the missed detection trials, the BAPF significantly outperformed the SIR particle filter at all false alarm rates tested, although the difference is greater for smaller rates.

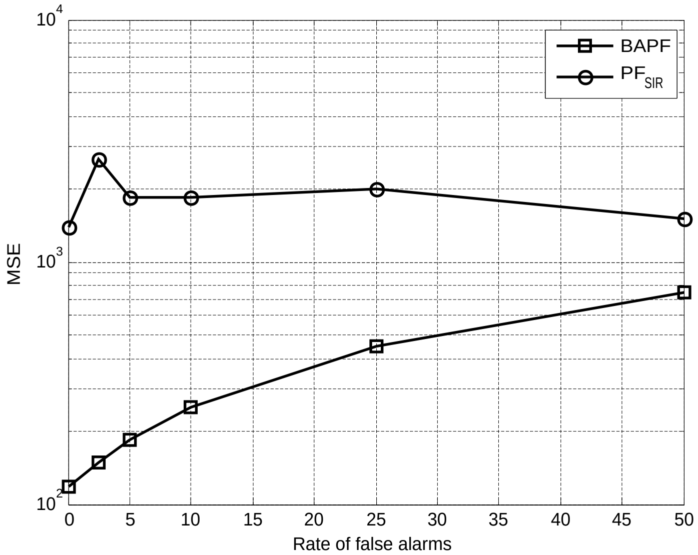

Figure 3.6: MSE vs false alarm rate (spikes/sec) for a 50 neuron ensemble.

Figure 3.7 compares filter performance as a function of spike sorting error. Spike sorting error occurs when a spike originating from one neuron is attributed to some other neuron. In this analysis, sorting errors are generated between pairs of neurons. It is found that BAPF performance is significantly better than that of the SIR particle filter up to a spike sorting error rate of 50%.

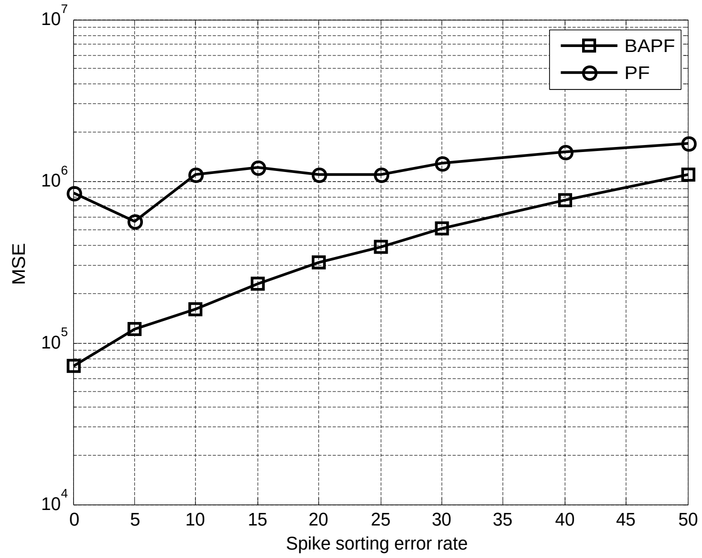

Figure 3.7: MSE vs percentage of sorting error for a 50 neuron ensemble.

3.5 Neural Data Analysis

This Section analyzes the ability of the Bayesian Auxiliary Particle Filter (BAPF) to decode neural activity recorded through micro electrode arrays. The BAPF signal processing algorithm is used to predict the angle of a subject's elbow during a center-out reaching task. The results are compared to the predictions obtained using a Kalman filter (KF). The data acquired for this analysis was provided by Dr. Jose Carmena at the University of California, Berkley.

In this experiment [62], an adult male rhesus monkey was trained to perform a center-out reaching task. In center-out reaching tasks, a subject positions their hand or holds a cursor at a central target. Then on command the subject reaches or moves the cursor to a specified target located on the circumference of a circle around the central target. Once the target location is acquired, the subject is trained to hold that position for a period of time before returning their hand back to the central target.

In this experiment, 8 target locations were placed around a central target on a circle with diameter of a 8 cm. The target locations were circles with diameters of 1.5 cm placed equidistant around the perimeter as seen below in Figure 3.8. When prompted, the subject is required to hold his hand at one of specified target locations for 500-1000 ms.

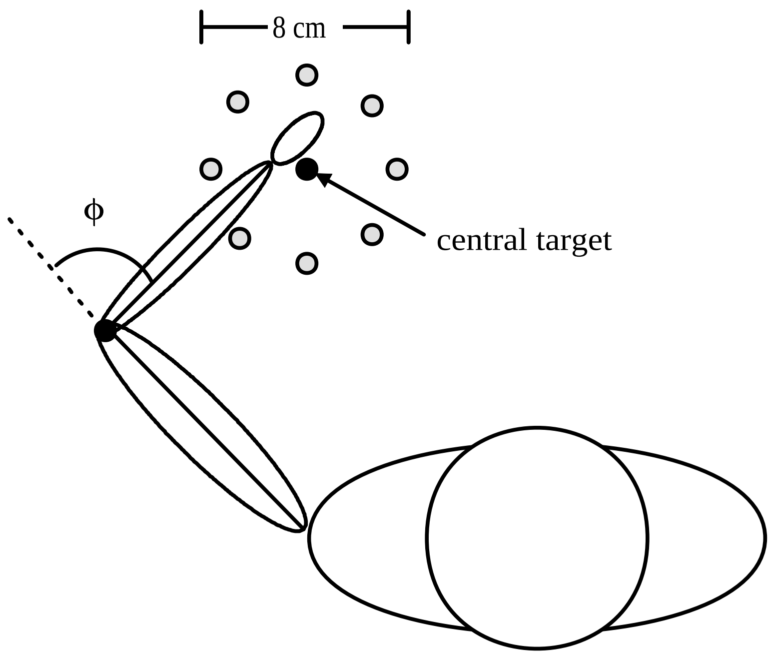

Figure 3.8: Conceptual top view of the experimental workspace for the center-out reaching task.

An array of 64 micro electrodes is arranged in an 8x8 configuration and implanted in the arm area of the primary motor cortex (M1) in the left hemisphere of the subject. During recording sessions, neural activity is sampled at 1 kHz. The monkey performs the center-out reaching task while elbow angle, velocity and acceleration were sampled at 36 kHz. Spike detection and spike sorting were performed using the Multichannel Acquisition Processor system by Plexon [63].

Research has shown that neurons which are associated with directional movement fire maximally in response to a particular angle [64, 65]. This angle is known as the preferred direction ϕ_p. The tuning functions of these neurons are typically modeled by the following cosine tuning curve

$$\lambda(t) = \begin{bmatrix} \beta_3(t) & \beta_2(t) & \beta_3(t) \end{bmatrix} \begin{bmatrix} 1 \\ \cos\phi(t) \\ \sin\phi(t) \end{bmatrix} \tag{3.15}$$

In this model, β_1, β_2 and β_3 are time varying parameters. The amplitude γ and preferred direction of the sinusoid are defined as

$$\gamma(t) = \sqrt{\beta_2^2(t) + \beta_3^2(t)} \tag{3.16}$$

$$\phi_p(t) = \tan^{-1}\frac{\beta_3(t)}{\beta_2(t)} \tag{3.17}$$

Equation 3.15 can be rewritten as

$$\lambda(t) = \gamma(t) \cos\left(\phi(t) - \phi_p(t)\right) + \beta_1(t) \tag{3.18}$$

An example of a cosine tuning function is seen in Figure 3.9, with $\beta = \gamma = 20$ and is directional tuned to $\phi_p = \pi/4$.

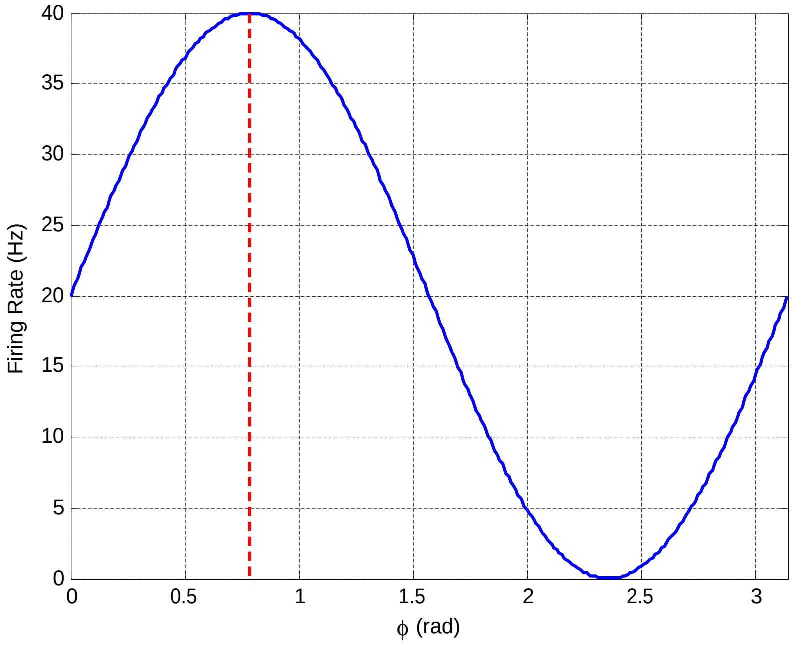

Figure 3.9: The solid line represents the instantaneous neural firing rate in response to an angle ϕ. The dotted line represents where the maximum firing rate is obtained in response to ϕ_p

In this research the BAPF is used to simultaneously predict the true elbow angle $\phi(t)$ common to the observed neurons and the preferred direction $\phi_p(t)$ of each of the $K = 79$ observed motor neurons whose firing rates are directionally modulated according to Equation 3.18. A training period of 300 seconds with known ϕ is first used to obtain initial estimates of ϕ_p, γ, and β for each neuron before predicting ϕ.

During the initial training session, the BAPF was executed K times using $P = 500$ particles to estimate each neuron's parameters separately. The estimated state vector for each particle during training sessions is $\hat{\mathbf{x}} = [\phi_p \;\; \gamma \;\; \beta]^{\mathrm{T}}$. With a sampling period of $\Delta t = 1$ msec, the neuronal spike trains are processed using a block of size $B = 100$. This training session was also used to obtain $\Phi(t)$, $\mathbf{z}(t)$, $\mathbf{P}(t)$ and $\mathbf{K}(t)$ in Equations 2.7 through 2.11 of Section 2.3 for the KF.

All neural parameters are assumed to evolve as Gaussian processes. As in Section 3.3.2, the filter parameters must be chosen such that the dynamics of the system are captured with the least amount of variability. The random sampling of Equation 3.6 is performed with $\sigma_{\phi 1} = 0.01$, $\sigma_{\gamma 1} = 0.02$ and $\sigma_{\beta 1} = 0.05$. The random sampling described in Equation 3.10 is performed with $\sigma_{\phi 2} = 0.01$, $\sigma_{\gamma 2} = 0.005$ and $\sigma_{\beta 2} = 0.01$. The final estimate of the parameters of each neuron from the training session are next used as the initial conditions of the BAPF when decoding begins.

When predicting ϕ during decoding sessions, the BAPF also estimates ϕ_p for each observed neuron while γ and β are held constant at their respective initial values. The estimated state vector for each particle during decoding sessions becomes $\hat{\mathbf{x}} = [\phi, \, \phi_{p1}, \, \cdots \, \phi_{pK}]^{\mathrm{T}}$. During the decoding sessions the number of particles was varied with $P = 50,\ 100,\ 200,\ 500,\ 1000,\ 2000$ and 5000. The sampling period remains the same with $\Delta t = 1$ msec and the samples are processed using block size $B = 100$.The random sampling of Equation 3.6 is performed with $\sigma_{\phi 1} = 0.15$, $\sigma_{\phi_{p1}} = 0.001$. The random sampling of Equation 3.10 is performed using $\sigma_{\phi 2} = 0.05$, $\sigma_{\phi_{p2}} = 0.001$.

After the initial training session, the recorded neural firings were decoded for a period of 30 seconds. Using 79 observed neurons, the monkey's elbow angle was predicted by observing neural firing times. Table 3.1 compares the MSE between predicted ϕ and true ϕ of the KF and various BAPF configurations. Figure 3.11 demonstrates the feasibility of using the BAPF to predict elbow angle by decoding actual cortical measurements. Figure 3.10 shows the elbow angle predicted via the KF.

Table 3.1: MSE between between the true position and predicted angle obtained from the Kalman Filter and varying particle size BAPF.

Filter	MSE	Filter	MSE
Kalman	0.2551	BAPF_{500}	0.2424
BAPF_{50}	0.2641	BAPF_{1000}	0.2003
BAPF_{100}	0.2554	BAPF_{2000}	0.2001
BAPF_{200}	0.2440	BAPF_{5000}	0.1965

The KF performed better than the BAPF until the number of particles was

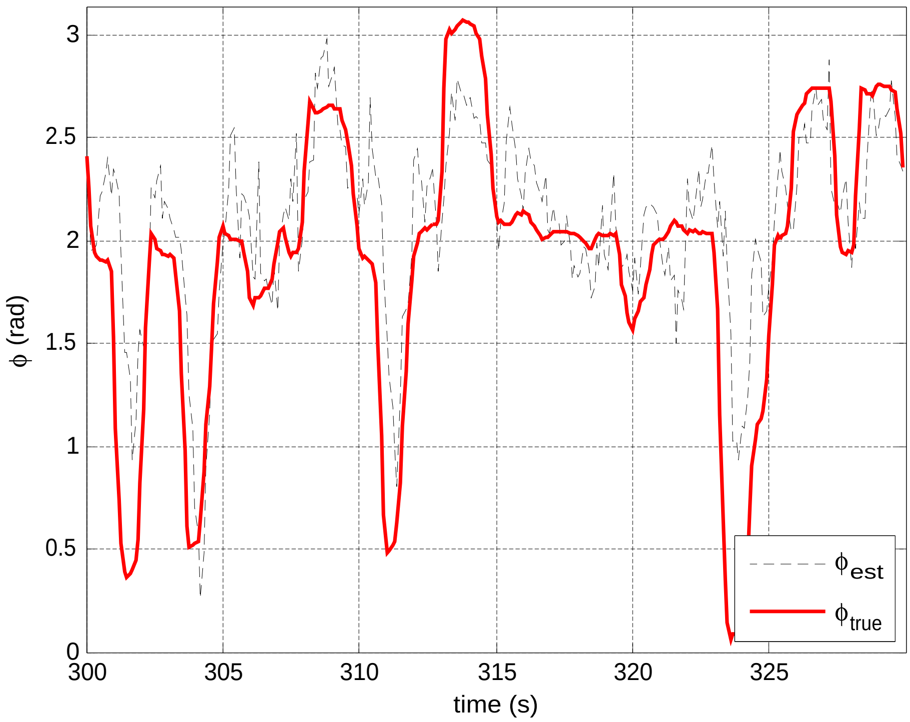

Figure 3.10: The solid line represents the true elbow angle ϕ. The dotted line represents the predicted elbow angle using the BAPF. In this decoding experiment, the number of particles is P=5000.

increased to 200. However, the KF was not able to accurately decode the entire neural ensemble. If there are observed neurons that have relatively low firing rates, the covariance matrix $\mathbf{P}(t)$ of Equation 2.9 becomes sparse and can not be inverted. This causes the filter to quickly diverge. In the KF experiment, it was necessary to remove 23 of the 79 observed neurons from the ensemble. On the contrary, this was not the case with the BAPF. If neurons have low neural activity, the BAPF is able to compensate for this during training sessions by estimating the maximum firing rate accordingly.

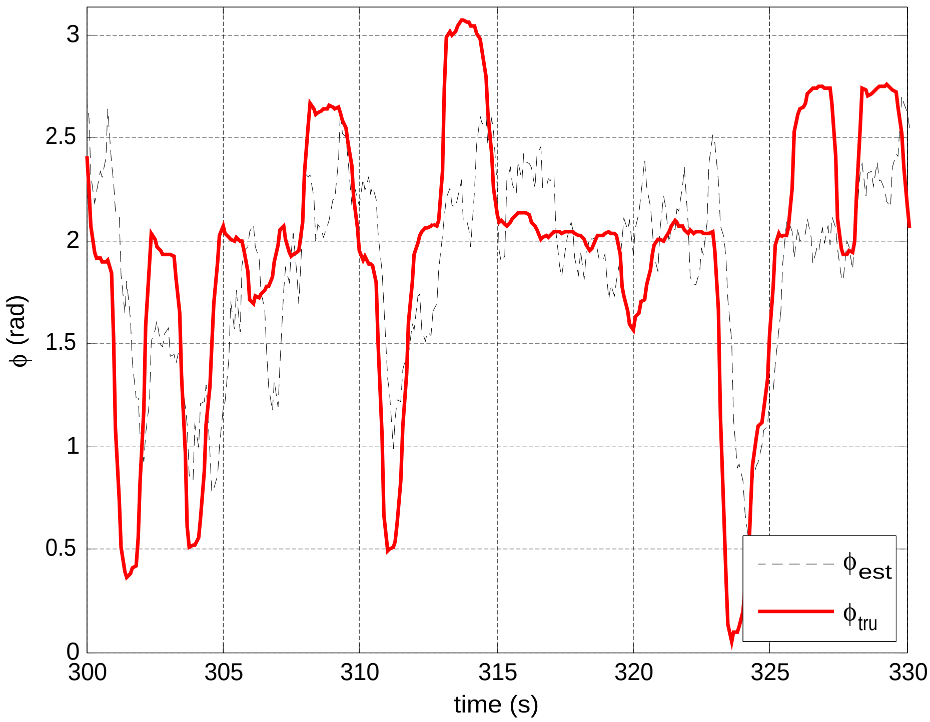

Figure 3.11: The solid line represents the true elbow angle ϕ. The dotted line represents the predicted elbow angle using the KF.

This Chapter has discussed and analyzed some existing methods used to decode neural spike trains. These include both linear and non-linear approaches [1]. Although these approaches represent important strides towards understanding the underlying neural processes, their drawbacks limit their applicability and stability in BMIs. For example, non-adaptive methods must be periodically retrained. Kalman filter-based methods (both linear and non-linear) are not capable of estimating multimodal distributions and erroneously assume that all noise is Gaussian [16]. Furthermore, the class of

Kalman filters require knowledge of the measurement and observation matrices.

The underlying algorithmic basics of the BAPF addresses many of the limitations of current neural decoding methods. For example, BAPFs have an inherent memory of the neural system as it evolves over time. This memory is effectively encoded in the particles through the assumption of Markov chain priors [66].

The posterior distribution of the evolving neural parameters given past firing is approximated using particles and their associated weights. Neural parameter estimates are thus constructed using a compromise between this approximated posterior distribution and the current likelihood of neural firing via the Bayesian paradigm [57]. When the neural firing likelihood does not accurately characterize current firing, the compromise favors the approximated posterior distribution in order to correct the estimate in subsequent time steps.

One practical limitation of the BAPF is that it is computationally intensive. Owing to the large number of tracked parameters, each of which is modeled by a large number of particles, this method is not well suited to real-time implementation using a conventional sequential processing unit. For example, the simulations presented in Section 3.4 tracked up to 100 parameters for 100 neurons, each modeled with 100 particles. These simulations required 125 seconds to execute 30 seconds of simulated data in the MATLAB environment on an Intel Core2 Duo. As the number of particles and number of neural channels increases, the computation time increases exponentially which prohibits real-time implementation. Particle filtering algorithms using general purpose processors for BMI applications are not practical. It may be for this reason that particle filters are not currently used for real-time BMI systems [11].

The serial computational nature of conventional processors prevents real-time use of the BAPF algorithm when the state vector incorporates a large number of dynamic parameters. This suggests that a parallel processing architecture in configurable hardware of the BAPF algorithm would recommend itself for use in real-time application of BMIs. A parallel hardware implementation accommodates large neural ensembles as well as an increase in the number of particles used by the filter to provide more accurate neural decoding. It was shown in [67] that it is possible to design custom dedicated computational hardware units that can significantly increase throughput of the system. Such a parallel configurable hardware filtering architecture is described in Chapter 4.

CHAPTER 4

Hardware Implementation

A salient focus of this research is to translate the Bayesian auxiliary particle filter (BAPF) into reconfigurable hardware implemented as a field programmable gate array (FPGA). This Chapter describes the parallel processing units that are utilized to facilitate the implementation of the BAPF by exploiting properties of the BAPF signal processing algorithm which make it conducive to parallel processing. A highly scalable hardware architecture is developed for its use in real-time neural signal processing.

4.1 Field Programmable Gate Array Architectures

Field programmable gate arrays (FPGAs) are customizable logic devices that are comprised of configurable logic blocks (CLBs), programmable interconnections and input/output (I/O) cells. Each CLB may include: look-up tables (LUTs), multiplexers, flip-flops, basic logic elements (AND, OR and NOT), block random access memory (BRAM) and shift registers [68, 69]. The number of CLBs contained within a single FPGA can vary from hundreds to tens of thousands. In addition to the CLBs, current fine-grained devices incorporate dedicated fixed point multipliers for efficient implementation of arithmetic functions.

FPGAs offer advantages over other processing devices such as a digital signal processor (DSP) or application specific integrated circuit (ASIC). DSPs are essentially highly specialized general purpose processors tailored to the needs of signal processing [70]. Although DSPs offer efficient execution of computationally intensive algorithms such as fast Fourier transform pairs, they operate using high level instruction sets and do not lend themselves to parallel processing architectures which can adversely affect throughput performance. Conversely, with the FPGA, multiple DSP computations can be executed simultaneously in parallel through dedicated architectures.

While the high speed and low production costs of ASICs make them appealing for real-time signal processing applications, the initial cost of designing an optimized architecture is expensive and time-to-market can take years [71]. Another deficiency of the ASIC is that its circuitry is static once it is produced. That is, the functionality of the device can longer be altered. However, using an FPGA, the synthesis of a design can be implemented instantly and any changes that need to made can be reconfigured into the design.

FPGAs provide an ideal platform for DSP implementation, combining the reprogrammability, architectural flexibility and system-level integration of general purpose processors with the performance offered by customizable hardware. Even with current clock speeds well below conventional processors, reconfigurable logic can yield substantially superior throughputs when made massively parallel [72]. Therefore, the BAPF algorithm can be implemented in hardware for real-time neural signal processing using an FPGA.

4.1.1 Xilinx Virtex FPGA

The course-grained architecture of the Xilinx Virtex FPGA (2002) shown in Figure 4.1 uses a general routing matrix (GRM) to configure the interconnects between each of its CLBs. The GRM is an array of static memory cells which store values that control the configurable logic elements and interconnects between the CLBs. These configuration values can be reloaded to change the functionality of the device.

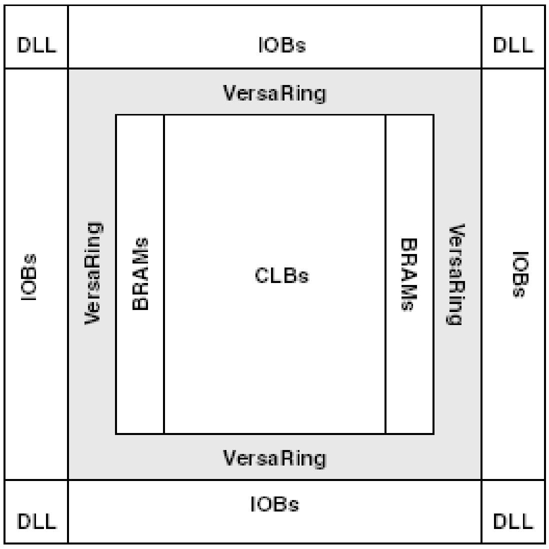

Figure 4.1: Xilinx Virtex FPGA architecture - the GRM controls the connectivity of each of the CLBs to BRAM and I/O pins

The CLBs communicate with external devices through the I/O cells surrounding them. Each CLB of Virtex family devices consists of two slices. A simplified view of a 2-slice CLB is shown in Figure 4.2. Each of these slices consists of two logic cells (LCs). The LCs themselves contain: four-input LUTs, a flip-flop and carry and control logic. Not shown in the figure are the 1-bit adders and AND gates for fast and efficient multiplication. Each LC also has feedthrough paths for signal routing that do not consume any logic. The maximum clock rate for Virtex devices is 200 MHz.

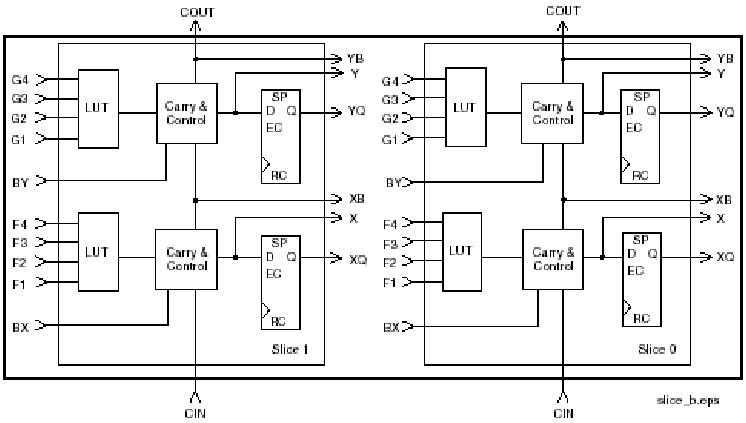

Figure 4.2: Simplified view of a Xilinx Virtex slice

4.1.2 Xilinx Virtex-5 FPGA

By interconnecting CLBs to create modules such as adders and multipliers, FPGA performance is reduced [69]. To overcome this, fine-grained FPGA architectures incorporate frequently used modules as hard core constructions. This is seen in the later generation Xilinx Virtex 4 (2004) and Virtex 5 (2006) families, where some of the slices contain dedicated DSPs. In particular, the Virtex-5 devices have 18-bit x 25-bit embedded multipliers called a DSP48E, whose architecture is shown in Figure 4.3.

Additional features of Virtex-5 devices include: phase-locked loops to serve as a frequency synthesizer to produce a wide range of clock frequencies, 36 Kb dual-port distributed RAM is available to reduce signal routing and clock rates of up to 550 MHz are possible with some designs. The functionality of each slice was further enhanced by expanding the architecture to include four

LUTs, four flip-flops, 16:1 multiplexers and 32-bit shift registers. The Virtex-5 slice is shown in Figure 4.4. The Virtex-5 was chosen as the target device for synthesizing the parallel BAPF architecture.

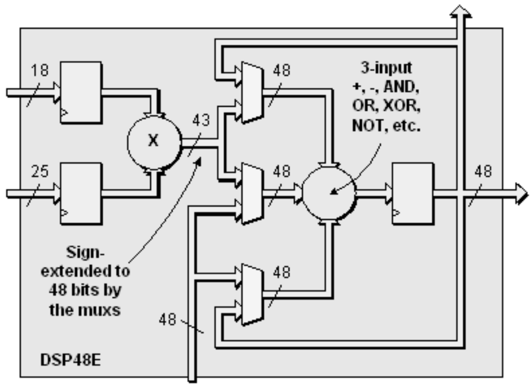

Figure 4.3: Detailed view of DSP48E

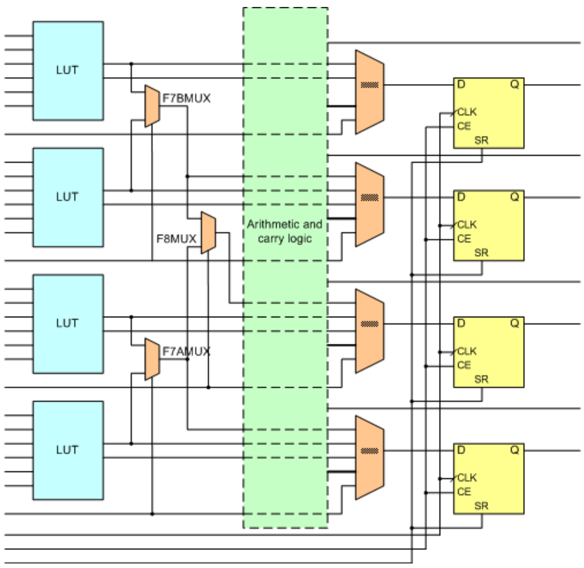

Figure 4.4: Detailed view of Virtex-5 slice

4.2 Parallel BAPF Implementation

One of the significant points of this research was to develop a neural decoder that is both modular and reconfigurable in hardware. More specifically, a parallel hardware architecture for the BAPF was developed in order to increase throughput over conventional sequential processing for real-time neural signal processing. The modular design presented in this Chapter allows for rapid prototyping of particle filters for a variety of applications and observed neural ensembles. The reconfigurable design is verified through a scaled version of the particle filter described in Section 3.3.2.

Figure 4.5 is a top-level hardware configuration for a particle filtering architecture. There are four major components: 1) the controller, 2) the random number generator (RNG), 3) the particle processors and 4) the resampler. Each of these components can be easily replaced with another module to accommodate the desired application.

The controller is modeled as a finite state machine (FSM). The FSM provides select lines (sel) to multiplexers that determine signal routing through the datapath. The FSM also regulates the status of enable (en) and reset (rst) signals of data registers. Control signal values are determined by the state of the FSM. The state changes as a function of the number of observed neurons K, the number of particles P and the block size B. States 0 and 1 are used as a reset and an initialization state.

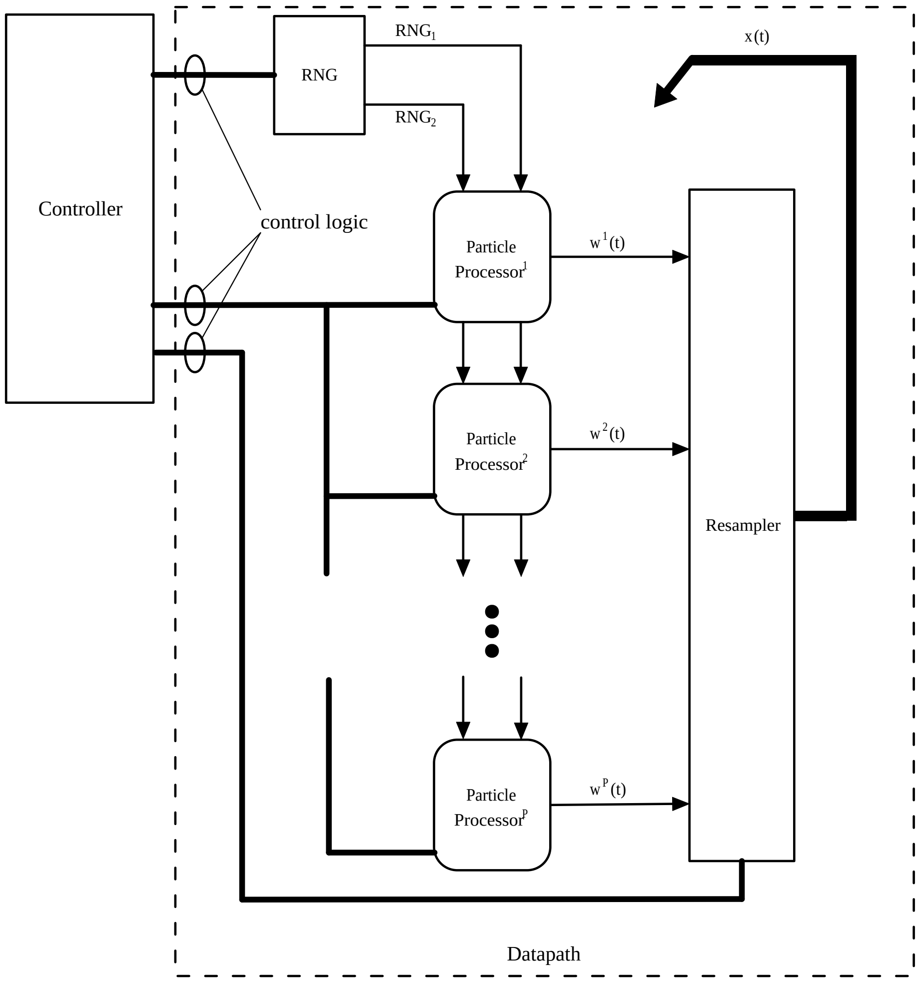

Figure 4.5: Top level controller-datapath diagram of the Bayesian auxiliary particle filer

The RNG is a stand-alone dedicated module. It operates independent of all other units in the system. It is a pipelined processor that produces a pair of random samples every clock cycle in order to sustain high throughput computations. The RNG is a fundamental component of any particle filter.

Each of the P particles described in Section 2.8 are used to estimate the driving signal $s(t)$ and arranged in a parallel fashion. The modular design allows many particle processors to be replicated easily in hardware. They share the same control logic, which allows all particle processors to perform the same computations simultaneously.

The resampling stage discussed in Section 4.2.6 has been configured to relieve most of computation time associated with the resampling process. The number of clock cycles needed to perform the resampling process with this innovative design is significantly reduced over existing methods.

The processing system is multirate. Using multiple clock domains across the particle filter, the separate computational units can operate at different clock frequencies. This significantly relieves the timing constraints of the datapath. Since this is a massively parallel processor, the required throughput can be achieved using relatively low clock rates.

Most of the computations in the datapath are performed according to the IEEE 754 single precision standard [73]. Fixed-point arithmetic is used where possible in order to reduce the required resources needed to design the filter architecture. One such occurrence is in the RNG. This module makes use of iterative fixed-point algorithms that provide high throughput data rates.

4.2.1 IEEE 754 Floating Point Package

The IEEE-754 single precision floating point (FP) standard is employed for performing many of the computations throughout the BAPF. The FP system provides a variable resolution for the range of numbers being represented. All particle filter computations are FP operations unless otherwise noted. In this format, a 32-bit number is divided into fields as seen in Figure 4.6 [74].

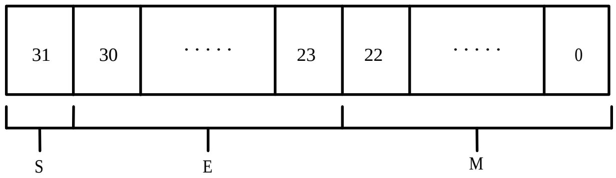

Figure 4.6: IEEE 32-bit floating point format representation

The most significant bit is the sign bit S, where 0 is positive and a 1 is negative. Eight bits are assigned as the exponent E and the remaining twenty three bits are the mantissa M. The exponent E is coded in a biased form as $E - 127$. Using this scheme a value n is represented as:

$$n = (-1)^S 2^{E-127}(1 + M) \tag{4.1}$$

where,

$$M = \sum_{k=1}^{23} \frac{b_k}{2^k} \tag{4.2}$$

and b_k is the value of bit position k in the mantissa.

4.2.2 Particle Datapath

An expanded view of one particle processor from Figure 4.5 can be seen below in Figure 4.7. Here it is shown that each particle processor is comprised of three submodules. The particle processor is partitioned into these three submodules to facilitate the reconfigurability of the FPGA.

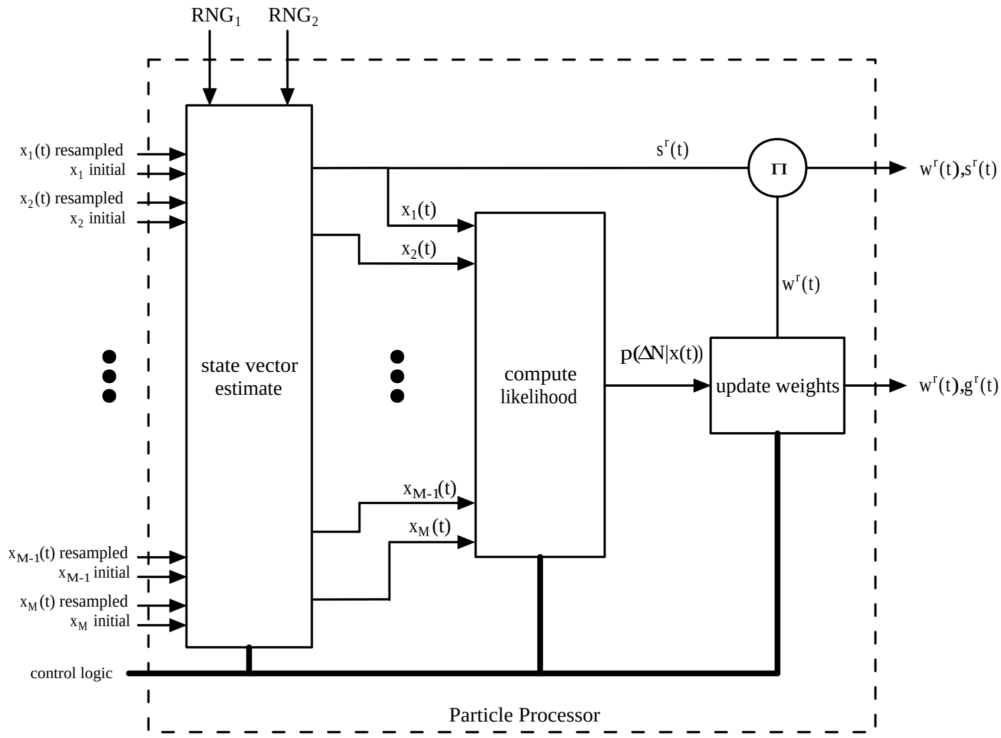

Figure 4.7: Sub-modules within a single particle 1)store its state vector, 2)compute its likelihood equation and 3) update its first and second stage weights.

The block labeled *state vector estimate* contains logic that stores an estimate of every element in the state vector. The number of neurons in the observed ensemble and the number of parameters being estimated per neuron will determine the length of the state vector M. As the size of M changes with the filter application, so can the configurable FPGA hardware that supports the state vector. A new *state vector estimate* module can easily be inserted to support the desired operation.

Likewise, the conditional probabilities that define the likelihood equations will also vary with the particle filter application. For example, the simulated hippocampal place cell firings of Section 3.1 are defined to arrive according to an inhomogeneous Poisson process with an expected arrival rate of $\lambda(t)$ defined by Equation 3.1. However, if a different tuning model is required for a different class of neurons, such as those defined by Equation 3.18, then the *compute likelihood* submodule can be replaced with one to compute $\lambda(t)$ specific to their firing behavior.

Conversely, the functionality of the *update weights* module never has to be replaced. This submodule only contains computational resources that store (registers and multiplexers) and normalize (floating-point divider) the first and second stage weights. The following sections describe in detail the submodules in Figure 4.7 as well as those shown in the top-level design of Figure 4.5.

In order to demonstrate this modular concept, a scaled version of the filter discussed in Section 3.3.2 was implemented to predict position from an ensemble of simulated hippocampal place cell firing times. Animal position $s(t)$ is estimated along with the receptive field centers $\mu(t)$ of each observed neuron. Although this configuration implements neurons with Gaussian-shaped tuning curves and Poisson firing, the concepts can readily be adapted to a variety of neural ensemble configurations, such as the cosine tuning curve described in Chapter 3.5.

4.2.3 Step 1: Random Sampling

Step 1 of Section 3.3.2 is to simulate the hyperparameters according Equation 3.6 as:

$$\Theta_1^r \sim \mathcal{N}(0, \sigma_1) \text{ for } r = 0, \ldots, P-1$$

Recall that each of the P particles contains an estimate of the receptive field center $\mu_j(t)$ for all K neurons in the ensemble. There is also an estimate of the driving signal $s(t)$. Therefore, $P(K+1)$ normally distributed random samples with standard deviation σ_1 are needed for Step 1.

Generation of random variables with distributions such as the normal, Poisson, chi-squared, exponential and many others can be obtained by transforming a uniformly distributed random variable on the interval $(0, 1]$ [75, 76]. Several techniques for generating a normal distribution from a uniform distribution are investigated in the following subsections.

4.2.3.1 Linear Feedback Shift Registers with Skip-ahead Logic

Generating a uniform random number on the interval $(0, 1]$ in hardware is typically done using a linear feedback shift register (LFSR)[77]. An LFSR is an ordinary shift register made up of m flip-flops and mod-2 adders (XOR gates) that are interconnected to form a feedback circuit. The register is initially seeded with any value other than 0. On each clock cycle, all bits are shifted one position with the output bit an input to a combinational feedback circuit whose output is then input to the shift register [78].

Since there are a finite number of possible states, the sequence will eventually repeat itself and is therefore not truly random. It is for this reason that LFSRs are often referred to as pseudo-noise (PN) generators. In order to extract as long a PN sequence as possible, the combinational feedback taps are chosen according to a generator polynomial $g(x)$ which results in a $2^m - 1$ maximal length sequence [79], where m is the number of bits in the register. If the coefficient of bit position n for $g(x)$ is 0, then that bit is omitted from the feedback logic. The contents of the shift register can be used to represent a number between 0 and 1 defined as

$$r = \sum_{n=0}^{m-1} x^n \frac{1}{2^n} \tag{4.3}$$

Figure 4.8 shows an example of a PN generator which uses a generator polynomial of $g(x) = x^4 + x^3 + x^1 + 1$, where x^n represents bit position n.

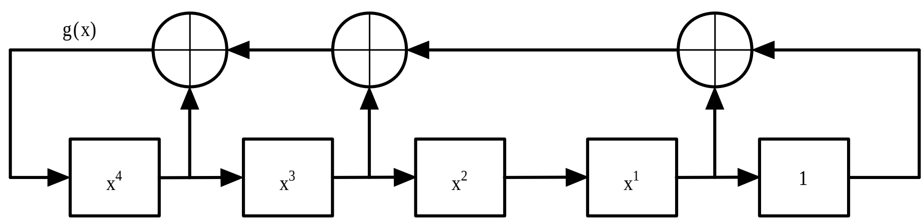

Figure 4.8: Feedback Circuit of an LFSR

The output of a PN generator is assumed to be spectrally white. That is, all frequencies should have approximately the same value across the power spectrum and are uncorrelated. However, there exists some degree of correlation between consecutive outputs of the PN generator. This is because from one state to the next, the contents of the register are shifted only one bit position and the input bit may or may not change. This results in a lowpass effect, where high frequency spectral components decrease resulting in non-white noise [80].

In order to avoid attenuation of the high frequency components and decorrelation of the uniform distribution, skip-ahead logic can be employed [81]. This process advances the contents of the register by k states rather than just one. Doing this will shift the register contents by k bits and allow k bits to be altered. Defining the behavior of the LFSR in matrix notation, $\mathbf{x}$(t) is the register state vector at time t and $\mathbf{G}$ is the state transition matrix representing the generator polynomial where:

$$\mathbf{G} = \begin{bmatrix} g_{m-1} & g_{m-2} & \cdots & g_1 & g_0 \\ 1 & 0 & & & 0 \\ 0 & 1 & & & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \qquad \text{and} \qquad \mathbf{x} = \begin{bmatrix} x^{m-1} \\ x^{m-2} \\ \vdots \\ x^1 \\ x^0 \end{bmatrix}$$

with the state of the register at time t+1 being defined by

$$\mathbf{x}(t+1) = \mathbf{G}\mathbf{x}(t) \tag{4.4}$$

For example, given a generator polynomial $g(x) = x^3 + 1$ for $m = 4$ the transition matrix to advance the state to $\mathbf{x}(t+1)$ is

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.5}$$

Which results in

$$\mathbf{x}(t+1) = \begin{bmatrix} x^3(t+1) = x^3(t) \oplus x^0 t \\ x^2(t+1) = x^3(t) \\ x^1(t+1) = x^2(t) \\ x^0(t+1) = x^1(t) \end{bmatrix} \tag{4.6}$$

In order to advance the state by k stages, one simply exponentiates the state transition matrix to the k^{th} power. Using the skip-ahead technique for $k = 3$, the transition matrix of Equation 4.5 becomes

$$\mathbf{G}^3 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{4.7}$$

The resulting state vector is now expressed as

$$\mathbf{x}(t+1) = \begin{bmatrix} x^3(t+1) = x^3(t) \oplus x^2t \oplus x^1t \oplus x^0t \\ x^2(t+1) = x^3(t) \oplus x^1t \oplus x^0t \\ x^1(t+1) = x^3(t) \oplus x^0t \\ x^0(t+1) = x^3(t) \end{bmatrix} \tag{4.8}$$

An efficient hardware implementation of an LFSR using skip-ahead logic is presented in [82], where efforts are made to reduce the amount of extra logic that results from using this technique.

Note that when using this method, the value of k should not be a prime factor of $2^m - 1$ otherwise a maximal length sequence can not be obtained. As is the case in this example, $k = 3$ which is a factor of $2^m - 1 = 15$ and the resulting sequence repeats after $5 \neq 2^m - 1$ cycles. This phenomena can be avoided by using a Mersenne sequence in which the length of the sequence is a prime number [83].

A 16-bit LFSR using skip-ahead logic is implemented as an initial step for designing a parallel BAPF architecture. The generator polynomial is chosen as $g(x) = x^{15} + x^{14} + x^{13} + x^4$ which results in a maximal length sequence of 65535. The number of advanced states is $k = 13$.

4.2.3.2 Cumulative Distribution Function Transformation

The cumulative distribution function (CDF) describes the probability that an observed random variable X is less than or equal to x [84].

$$x \mapsto F_X(x) = p(X \leq x) \tag{4.9}$$

The CDF of X can be obtained by integrating its probability density function (pdf) $f_X(x)$ from $-\infty$ to x, which describes the likelihood of X to occur at a given value x [85].

$$F_X(x) = \int_{-\infty}^{x} f_X(x)dx \tag{4.10}$$

The mapping of Equation 4.9 can be used to produce a value on the real number line from a corresponding probability on the interval (0,1] generated from Equation 4.3. The pdf of a normal random variable with mean μ and variance σ^2 is defined as

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} exp\left\{\frac{(x-\mu)^2}{2\sigma^2}\right\} \tag{4.11}$$

Unfortunately, there is no closed form solution to the integral of Equation 4.10 for normally distributed random variates [86]. Numerical integration techniques are therefore often used to obtain a solution to this problem. One such solution is to use the error function $erf(x)$ [87]. The error function is defined for positive values of x as

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-u^2} du \tag{4.12}$$

The CDF of X for standard normals can be determined using Equation 4.12 as

$$F_X(x) = \frac{1}{2}\left[1 + erf\left(\frac{x}{\sqrt{2}}\right)\right] \tag{4.13}$$

Using the relation $erf(-x) = -erf(x)$, one can find the CDF for negative values of x.

Approximations to Equation 4.12 can be obtained using a Taylor series expansion, Chebyshev expansion or continued fractions [88]. Exhaustive tables which list the values of $F_X(x)$ for values of x with high resolution are widely available [89]. These tables are often implemented as look-up tables (LUT) in hardware and direct synthesis of the approximations is not required. Values are simply read from memory resulting in higher throughput. However, the LUTs required to store the $erf(x)$ values can become costly in terms of memory usage.

4.2.3.3 Composite Look-up-table

As discussed in Section 4.2.3.2 approximations to Equation 4.12 are often implemented using a LUT, which consume large amounts of memory resources. One way to reduce memory requirements is to use a composite LUT [90]. This methodology utilizes two LUTs to estimate the inverse cumulative distribution function (ICDF).

Here a high resolution LUT and a low resolution LUT, which is a decimated version of a high resolution LUT, are used to represent different portions of the ICDF curve. The ICDF curve for normal distributions exhibits an approximately linear relationship for values on the interval $(0.1, 0.9)$ and is highly nonlinear for values outside this domain. The decimated low resolution LUT is used to represent the majority of the values that lie between 0.1 and 0.9 using a stair case approximation and the high resolution LUT is used to represent the remaining values. This allows for large memory savings in the linear regions while introducing only modest errors and the nonlinear regions can be approximated more accurately [90].

4.2.3.4 Box-Muller Transformation

An alternative to using a LUT is to use a computational algorithm that transforms the uniform distribution into a normal distribution such as the method suggested by Box and Muller [91]. The Box-Muller transformation generates two standard normal distributions $\mathcal{N}_1$ and $\mathcal{N}_2$ from two uniformly distributed numbers U_1 and U_2. A standard normal refers to a normal distribution with zero mean and unit variance. The two uniform distributions are transformed into normal random variables by the Box-Muller method using the following steps:

$$let \;\; \omega = 2\pi U_1 \tag{4.14}$$

$$let \;\; R = \sqrt{-2ln(U_2)} \tag{4.15}$$

$$\mathcal{N}_1(0,1) = R\cos(\omega) \tag{4.16}$$

$$\mathcal{N}_2(0,1) = R\sin(\omega) \tag{4.17}$$

There are a number of ways to perform computations involving exponentials, logarithms and trigonometric functions in hardware. However, there are trade-offs between hardware utilization and throughput. The Coordinate Rotation Digital Computer (CORDIC) algorithm is an iterative process for computing the rotation of a vector in a Cartesian coordinate system and evaluating the length and angle of the vector [92]. The CORDIC method was later expanded for multiplication, division, logarithm, exponential and hyperbolic functions. The resulting vector z_n of the rotation of a vector $[x_0, y_0]$ by an angle θ in Cartesian coordinates can be computed by the following matrix operation:

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \tag{4.18}$$

Using the identity: $\cos\theta = 1/\sqrt{1+\tan^2\theta}$ and factoring out $\cos\theta$ Equation 4.18 can be modified as follows:

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \frac{1}{\sqrt{1+\tan^2\theta}} \begin{bmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \tag{4.19}$$

If $\tan\phi = 2^{-n}$, then

$$K_n R\sin(\theta \pm \phi) = R\sin\theta \pm 2^{-n} R\cos\theta \tag{4.20}$$

$$K_n R \sin(\theta \pm \phi) = R \cos\theta \mp 2^{-n} R \sin\theta \qquad (4.21)$$

where, $K_n = \sqrt{1 + 2^{-2n}}$

These equations can be used to implement the rotation of a vector R through either a positive or negative angle equal in magnitude to $\tan(2^{-n})$. Therefore, a sequence of rotations with increasing powers of n can be used to rotate R through any desired angle [93].

In the CORDIC method, the rotation by an angle θ is implemented as an iterative process, consisting of finer rotations during which the initial vector is rotated by pre-determined step angles [94]. As the number of iterations increases, so does the accuracy of the algorithm. This algorithm uses very little resources with only shift and add operations. However, the number of iterations required to achieve the desired accuracy may become burdensome.

The CORDIC algorithm is used to implement the trigonometric, logarithmic and square root functions in Equations 4.15 through 4.17 of the Box-Muller algorithm as shown in Figure 4.9. Starting with two 16-bit LFSRs described in Section 4.2.3.1, each seeded with a different initial value, two independent uniform random numbers between 0 and 1 are generated. Since there is no dependence between R and ω for computing the output sequences, they are computed in parallel.

U_1 is passed through a constant multiplier where it is scaled by 2π to execute Equation 4.14. U_1 is then delayed for 61 clock cycles to synchronize it with U_2 for the final stage. While U_1 is being delayed, U_2 is being processed by the CORDIC log unit for 28 clock cycles to achieve the desired accuracy. The output of this unit is then multiplied by -2, which is carried out by a left-shift of one bit and an inversion of the sign bit in order to avoid hardware resource consumption and to decrease computational latency.

This intermediate quantity is then passed to the CORDIC square root unit for 44 clock cycles to complete the implementation of Equation 4.15. While the final value of R is being computed, the $\sin\omega$ and $\cos\omega$ values are simultaneously being processed by another CORDIC processor which takes 11 clock cycles. The 61 cycle delay in the ω datapath is is due to the computation of $\sin\omega$ and $\cos\omega$ requiring only 11 clock cycles total and computation of R

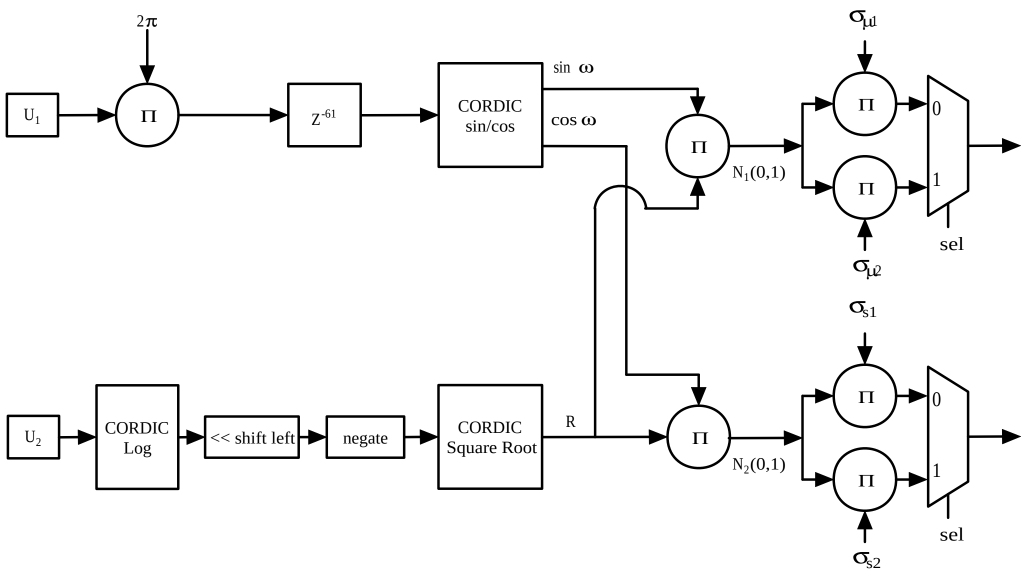

Figure 4.9: Box-Muller using CORDIC processors

for 72 clock cycles. Two multiplications can now be executed in parallel to produce the two sequences defined by Equations 4.16 and 4.17, which require another 3 clock cycles.

The total number of clock cycles needed to compute a pair of random values is 75. However, this is just an initial delay until the computational pipeline is filled. Once the pipeline is full, the minimum latency to produce a pair of random values becomes only one clock cycle.

Additionally, the length of $\mathcal{N}_1$ and $\mathcal{N}_2$ can be conveniently increased by using a re-seeding process of one of the LFSRs. After $2^n - 1$ cycles, the LFSR that generates U_1 is given a new initial value and the LFSR that generates U_2 is allowed to repeat. The sequence of values that represents U_2 will be the same. However, the order of the sequence of the next $2^n - 1$ values that represents U_1 will be different. This will produce a different pair of values that represent R and ω. This process can be repeated $2^n - 1$ times to generate all possible combinations of U_1 and U_2. An efficient way to implement this methodology is to use a counter that has the same number of bits as the LFSRs. After $2^n - 1$ cycles, the counter is incremented by one and that value is used to re-seed the U_1 LFSR. This process continues until the counter cycles through all possible values with the exception of zero.

4.2.3.5 Marsaglia Polar Method

The Marsaglia Polar Method is another computational technique for generating standard normal random variables from a uniformly distributed random variable [95]. It follows a process similar to the Box-Muller method but eliminates the need to compute $\sin(\omega)$ and $\cos(\omega)$ by directly generating those values from a uniformly distributed variable U on the interval ± 1 [96]. If U is on the interval $(0, 1]$, then $V = 2U - 1$ is on the interval ± 1. Using Equations 4.22 and 4.23 the Marsaglia polar method generates two standard normals [97].

$$\mathcal{N}_1 = V_1 \sqrt{\frac{-2logW}{W}} \tag{4.22}$$

$$\mathcal{N}_2 = V_2 \sqrt{\frac{-2logW}{W}} \tag{4.23}$$

This algorithm requires that $W = V_1^2 + V_2^2 < 1$. If it is not, then it is rejected and a new V_1 and V_2 are selected. This would require extra logic and a parallel implementation as in Section 4.2.3.4 can not be used since pipelining of the data is interrupted.

For a single iteration of the BAPF algorithm, there are two instances where a random sample is drawn for every element of the state vector fro each particle. When the number of observed neurons and number of particles become large, this random sampling hinders real-time performance.

Therefore, a high throughput random number generator is necessary. After analyzing the various techniques for generating normal random variables, it was best decided that the Box-Muller method be used for high throughput and low resource utilization. The parallel pipelined hardware architecture of Figure 4.9 produces two random samples every clock cycle. This will help facilitate the BAPF algorithm for real-time signal processing.

4.2.4 Step 2: Add Randomness to Particle Estimates

According to Equation 3.7 of Section 3.3.2, random samples are drawn by adding zero-mean random numbers to previous estimates as

$$\hat{\mathbf{x}}^r(t) = \mathbf{x}^r(t-1) + \Theta_1^r$$

Storing the random samples along a tapped delay line allows $\mathbf{x}^r(t)$ to be formed as the sum of a delayed sample and the previous estimate $\hat{\mathbf{x}}^r(t-1)$. Figure 4.10, which is an expanded view of the *state vector estimate* block of Figure 4.7, illustrates how this is accomplished through a feedback loop by setting sel=0 and en=1 for a single clock cycle. The total length of the tapped delay line for the $s(t)$ estimates is P. The total length of the tapped delay line for the $\mu(t)$ estimates is KxP over all particles. This can result in a long delay line. However, if the RNG is allowed to run at a rate much higher than the system sampling rate, all random samples can be refreshed before the next computation. This step, adding randomness to particle estimates, requires only a single clock cycle to execute.

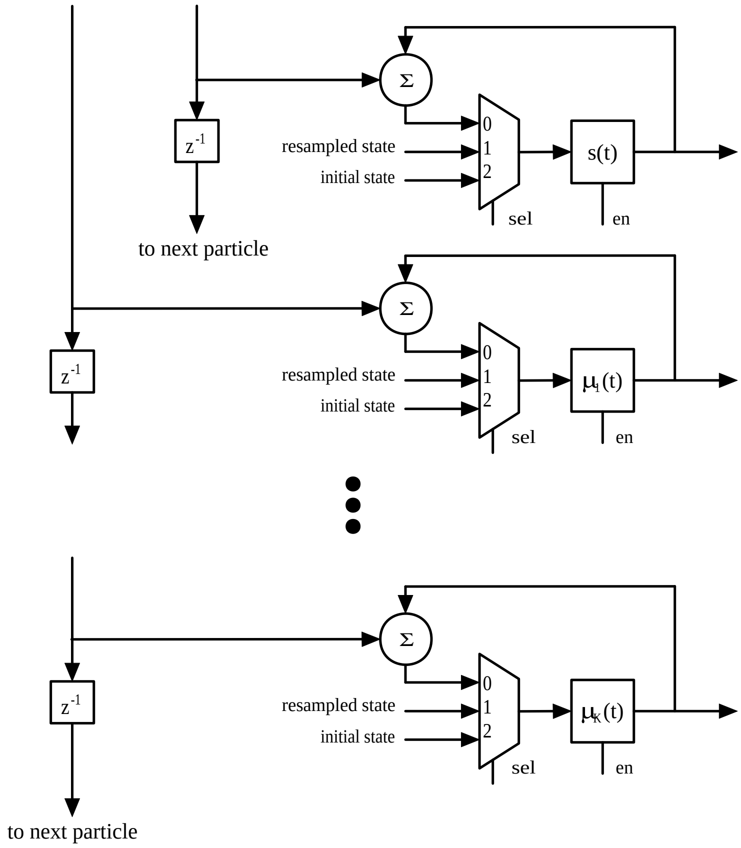

Figure 4.10: Expanded view of the state vector submodule. Select and enable signals control the flow of data into the registers storing the state vector.

Recall that the standard deviation of the random samples in Equation 3.6 is different for the driving signal estimates than it is for the receptive field centers. This requires that samples be drawn from two separate distributions. To obtain a normally distributed random variable of some arbitrary standard deviation, the samples drawn from a standard normal $\mathcal{N}(0, 1)$ can simply be multiplied by the desired standard deviation. This is conveniently executed by setting *sel* in Figure 4.9 to produce two orthogonal sequences of the appropriate standard deviations.

4.2.5 Step 3: Compute the first stage weights $g^r(t)$

According to Equation 2.28, each first stage weight $g^r(t)$ is computed as a product of the second stage weight from the previous iteration $w^r(t-1)$ and the conditional likelihood function as

$$g^r(t) \propto w^r(t-1)p(\Delta N(t)|\hat{\mathbf{x}}^r(t)) \tag{4.24}$$

When Poisson firing rates are assumed, the posterior density for particle r becomes

$$p(\Delta N_j(t)|x^r(t)) = \prod_{j=1}^{K} \frac{(\lambda_j \Delta t)^{\Delta N_j(t)} e^{-\lambda_j \Delta t}}{\Delta N_j(t)!} \tag{4.25}$$

which simplifies to

$$p(\Delta N_j(t)|x^r(t)) = \prod_{j=1}^{K} (\lambda_j \Delta t)^{\Delta N_j(t)} e^{-\lambda_j \Delta t} \tag{4.26}$$

since $\Delta N(t)$ is either 0 or 1. For blocks of data as discussed in Section 3.2, assuming a Poisson firing model the conditional probability for neuron j is computed as a product over a block duration as

$$p(\Delta N_j(t)|x^r(t)) = \prod_{b=1}^{B} p(N_j(t)^b|x^r(t)) \tag{4.27}$$

where B is the block size and b indexes the sample number within a given block.

Recall from Section 3.3.2 that the term $\lambda_j(t)$ is a Gaussian-shaped function with two of the terms, $\alpha = 3.5$ and $\xi = 12$, being constant. Therefore, the expected firing rate for neuron j of the r^{th} particle becomes

$$\lambda_j^r = exp\left\{3.5 - \frac{(s^r(t) - \mu_j^r(t))^2}{144}\right\} \tag{4.28}$$

After the updated value of $x^r(t)$ from Section 4.2.4 is stored into the state registers, it can be used to compute Equation 4.26 which is executed over multiple stages. First, the argument of Equation 4.28 is easily computed using the structure shown in Figure 4.11.

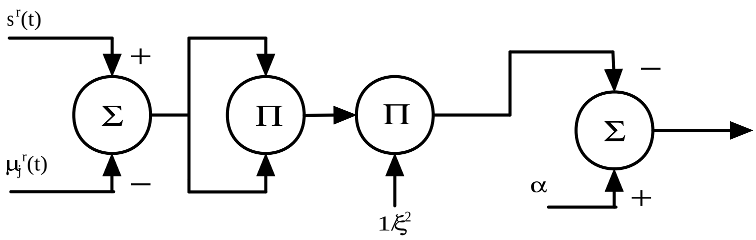

Figure 4.11: Architecture for computing the exponential argument of Equation 4.28

The quantity $\mu_j^r(t) - s^r(t)$ is the output of the subtracter. The multiplier output is $\left(\mu_j^r(t) - s^r(t)\right)^2$. Two more floating point operations are then carried out to obtain the argument of the exponential over a total of 4 clock cycles. The output of Figure 4.11 is then passed as the input to a module for computing the exponential term.

4.2.5.1 Floating Point Exponential

The exponential function $f(x) = e^x$ is often used in scientific and engineering applications. It is defined as the following Taylor series

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots \tag{4.29}$$

Since it is impossible to compute an infinite series, numerous techniques have been employed to approximate a solution to the exponential function. One way is to perform an n^{th} order polynomial approximation by truncating Equation 4.29 to the first n terms [98]. Using this method requires *a priori* knowledge of the number of terms needed in order to acquire a desired degree of accuracy [99]. The required number of terms varies according to the magnitude of x and can occasionally result in a large number of multiplications which severely hinders throughput. Another common method is to use a look-up-table (LUT). Traditionally, LUTs store all the possible solutions to some arbitrary function in read only memory (ROM) [100]. The size of the memory varies according to bit representation and desired accuracy.

If high resolution is needed, the size of the memory required to store the solution becomes quite large. For example, if IEEE 754 single precision is used, there is a possible 2^{32} inputs to the LUT which are each 32 bits wide. That is over 17 GB of memory to compute the solution. Reducing the size of the LUT can be achieved by separating the argument of the exponential into its integer and fractional parts v and w respectively to get

$$f(x) = e^x = e^{(v+w)} = e^v e^w \tag{4.30}$$

This requires storing only LUT values of e^w for $0 < w < 1$ and e^v where $v = \lfloor x \rfloor$. Additional savings can be made by storing only positive values of x since $e^{-x} = 1/e^x$. This method performs $v + 2$ multiplications which can become burdensome when $|v|$ is large.

Recently, methods have been introduced that combine LUTs and Taylor series expansions to offer trade-offs between computation time and resource utilization for FPGA implementation [101, 102]. Even though these methods reduce ROM storage over existing methods, the overall designs still consume 20% of the available resources on the Xilinx Virtex 4 XC4VLX200 FPGA for a single computational exponential unit. This means that only 5 units could exist in parallel on this current FPGA. The architectures described in [101] and [102] have pipeline latencies of 74 and 35 respectively.

In order to reduce resource utilization and increase throughput for the parallel BAPF architecture, a new floating point exponential core is designed as part of this research. By exploiting some relationships between floating point number representation and base 2 exponents, this novel architecture computes e^x without memory, Taylor series expansion or iteration.

Attention is focused on computing e^x for $x \geq 0$, since $e^{-x} = 1/e^x$ can be accounted for by a division. The system begins using the following property

$$e^x = 2^u, \text{ where } u = x/ln(2) \tag{4.31}$$

The value of u is obtained through a floating point division. The sign bit is then forced to zero in order to assure $u \geq 0$ while the original sign bit is passed along a delay line to be used later in the process. The quantity u is then separated into its integer v and fractional w parts resulting in

$$2^u = 2^{v+w} = 2^v 2^w, \text{ where } v = \lfloor u \rfloor \ and \ w = u - v \tag{4.32}$$

This results in two new terms, 2^v and 2^w, that need to be computed. A single precision floating point implementation of the *floor* function $\lfloor u \rfloor$ in Equation 4.32 is be executed as follows:

Since $u \geq 0$, the solution to $v = \lfloor u \rfloor$ is always going to be the integer portion of u. The number of bits residing in the mantissa which are used to represent the integer portion of an IEEE 754 number is given by the unbiased exponent $Q = E - 127$, where E is the 8-bit value described in Section 4.2.1. Therefore, all bits that are not needed to represent the integer part can be set to zero to obtain $\lfloor u \rfloor$.

The total number of non-zero bits that can possibly be used to represent an integer is 24 (23 bits of the mantissa + 1 implied bit for normalized numbers). These 23 bits of the mantissa are assigned simultaneously by first comparing Q to integers 1 through 23 in parallel The outputs of the 23 comparators are then used as the select lines of 23 multiplexers that determine each of the 23 bits of the mantissa of v as seen in Figure 4.12.

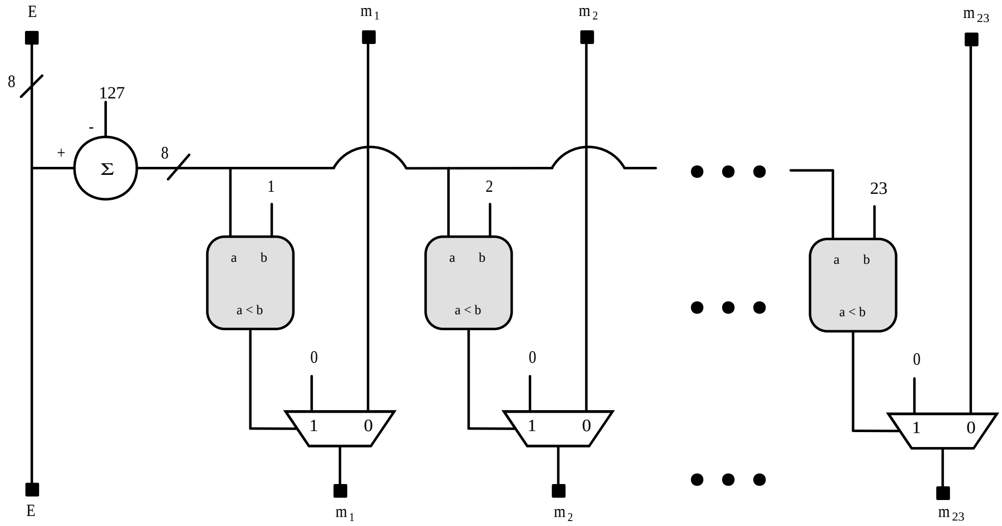

Figure 4.12: Parallel circuit for computing $\lfloor u \rfloor$

The output of a comparator is a 1 when Q is less than integer$_i$ and 0 otherwise. When the comparator selects input 1 on the corresponding multiplexer, bit position i of the mantissa becomes 0. When the comparator selects input 0, bit position i of the mantissa remains unchanged. The quantity v is then the result of concatenating a 0 as the sign bit, E as the exponent and the 23 multiplexer outputs as the mantissa to form a floating point representation of the integer. Computing $\lfloor u \rfloor$ requires only a single clock cycle. Subtracting v from u then yields w.

Now, the computation of 2^v and 2^w is executed. Calculating 2^v for any integer v for floating point numbers is easily computed. Note: the value 2^v

for any integer v in floating point notation is obtained by setting E equal to $127 + v$ and all 23 bits of the mantissa to zero.

Conversely, there is not a straightforward method for computing 2^w for $0 < w < 1$. In Figure 4.13(a), a plot of $f(w) = 2^w$ and $g(w) = w + 1$ for $0 \leq w \leq 1$ is shown. The difference between the two functions $d(w) = g(w) - f(w)$ can be observed in Figure 4.13(b).

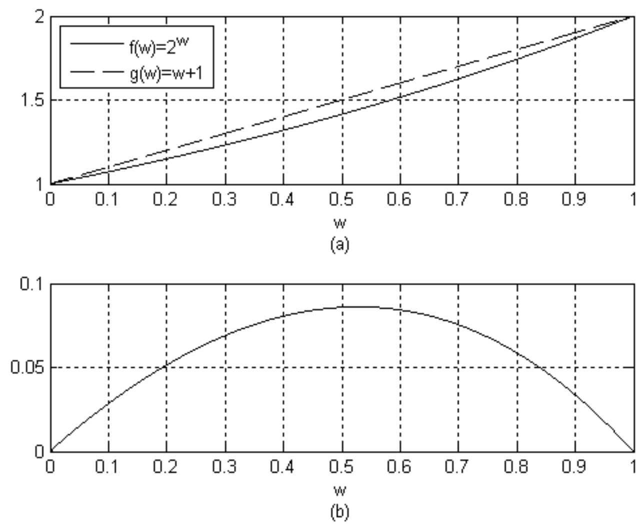

Figure 4.13: (top) $f(w) = 2^w$, $g(w) = w + 1$, (bottom) $d(w) = g(w) - f(w)$

Using a 2^{nd} order least squares approximation, an estimate $\widehat{d}(w)$ of $d(w)$ is obtained on the interval $0 \leq w \leq 1$. This provides a set of coefficients (a, b, c) that best fits the quadratic difference as

$$\widehat{d}(w) = aw^2 + bw + c \tag{4.33}$$

By subtracting Equation 4.33 from the straight line $g(w)$, the term 2^w for

$0 \leq w \leq 1$ is computed by Equation 4.34 with an error of 0.001%. Computing 2^w in this manner requires 3 clock cycles and is illustrated by Figure 4.14. It is now possible to multiply 2^v and 2^w to get 2^u, which is the desired result of Equation 4.31.

$$2^w = -aw^2 + (1 - b)w - c + 1 \tag{4.34}$$

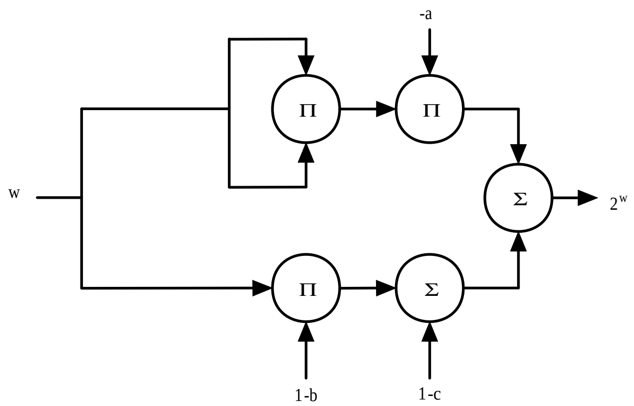

Figure 4.14: Computing 2^w

This fast floating point exponential core has a pipeline latency of 8 clock cycles. The top-level architectural view is shown in Figure 4.15. As mentioned earlier, e^x for $x < 0$ is the reciprocal of $e^{|x|}$. Dividing 1 by $e^{|x|}$ is performed as a final calculation. This is why the sign bit is delayed for 8 clock cycles. It is used as the select line to choose between $e^{|x|}$ for $x \geq 0$ or $1/e^{|x|}$ for $x < 0$. The quantity $\lambda_j^r(t)$ of Equation 4.28 is computed after a total of 12 clock cycles.

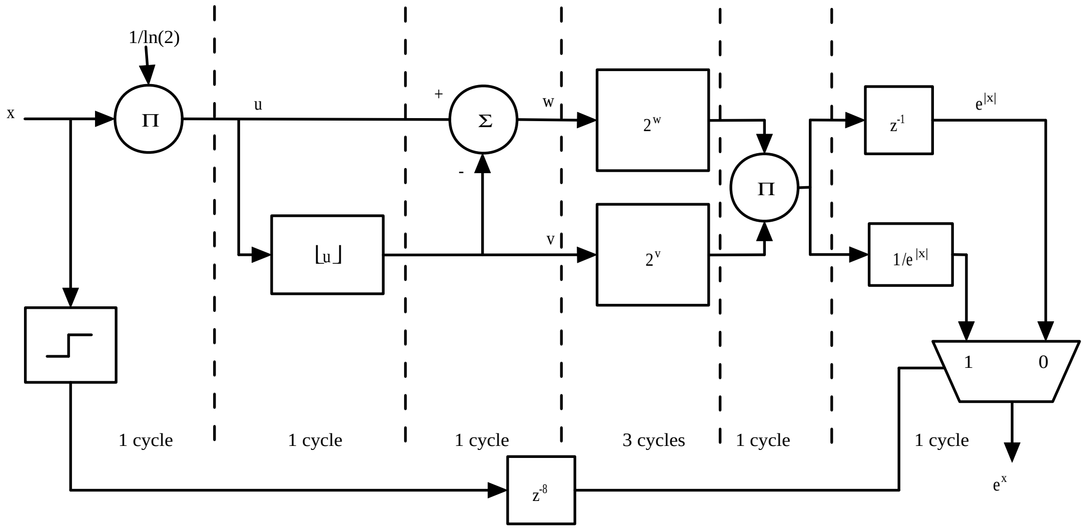

Figure 4.15: Top-level design for the fast exponential core

4.2.5.2 Computing the Conditional Likelihood

Equation 4.27 defines the likelihood of observing ΔN firings given the current estimated state over a block of samples as the product of the individual likelihoods within that block. Since ΔN is assumed to be either 0 or 1, Equation 4.27 can be rewritten as

$$p(\Delta N(t)_j | x^r(t)) = (\lambda_j \Delta t)^{\Delta N_j(t)} e^{-\lambda_j \Delta t B} \tag{4.35}$$

This requires computing of the terms $(\lambda_j \Delta t)^{\Delta N_j(t)}$ and $exp(-\lambda_j \Delta t B)$. Figure 4.16 is used to illustrate the computation process of these terms and their final product.

The term $\lambda_j(t)$ is the output of an *exp* module described in the previous

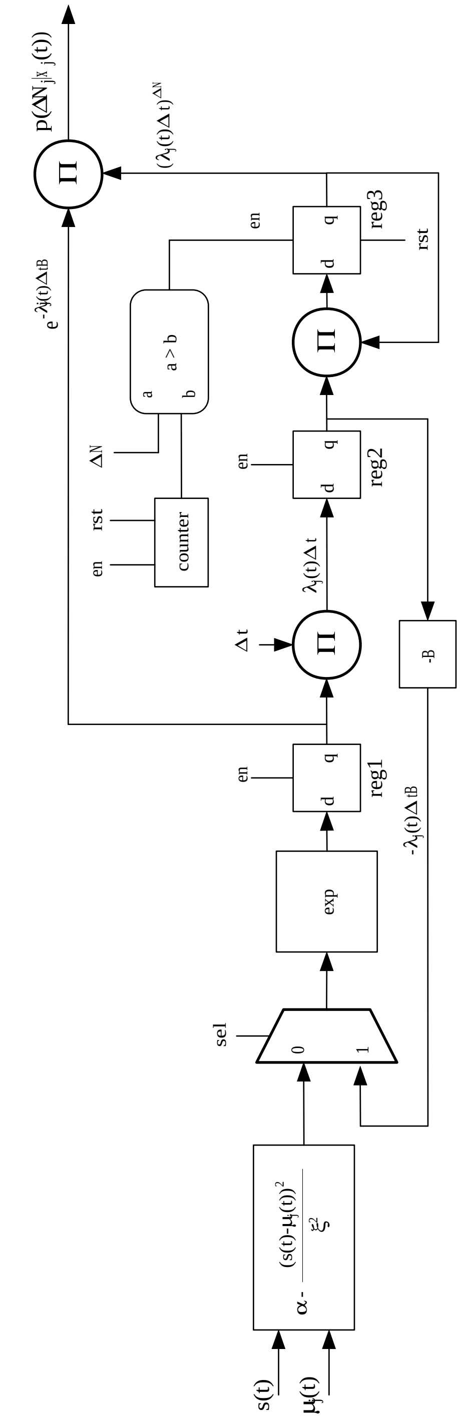

Figure 4.16: Architecture to compute $p(N_j(t)|x^r(t))$ of a single hippocampal place cell with a Gaussian-shaped tuning function

Section. This value is stored into *reg1*. Then, multiplied by Δt and stored into *reg2* to acquire $\lambda\Delta t$.

Since blocks of data are processed as described in Section 3.2, it is possible that the number of firings ΔN can take on values from 0 to B, the block size. In order to multiply $\lambda_j^r(t)$ by itself ΔN times, a multiplier that accepts $\lambda_j^r(t)\Delta t$ and feedback from a third register as inputs can be used over multiple clock cycles to form the product.

The value of *reg3* is initially set to 1 upon reset. The enable signal is governed by the output of a comparator that determines if ΔN is greater than a counter value. If the number of firings is greater than the count, the enable signal remains high and the feedback register continues to store successive products from the multiplier.

While $(\lambda_j \Delta t)^{\Delta N}$ is being computed, a feedback loop multiplies $\lambda_j(t)\Delta t$ by $-B$ before passing it back through a multiplexer to compute $exp(-\lambda_j^r(t)\Delta tB)$ through hardware re-use of the *exp* module. After $2B$ clock cycles, the product of $\left(\lambda_j^r(t)\Delta(t)\right)^{\Delta N}$ and $exp(-\lambda_j^r(t)\Delta tB)$ is formed to obtain the conditional probability for one neuron over a block duration.

The conditional probabilities of all K neurons are computed simultaneously in parallel as shown in Figure 4.17. These individual probabilities are then multiplied together to form the likelihood for a single particle processor. The required clock cycles to multiply all of the conditional probabilities together can be reduced to $log_2(K)$ by using parallel multipliers. The final product is then stored into the *likelihood* register.

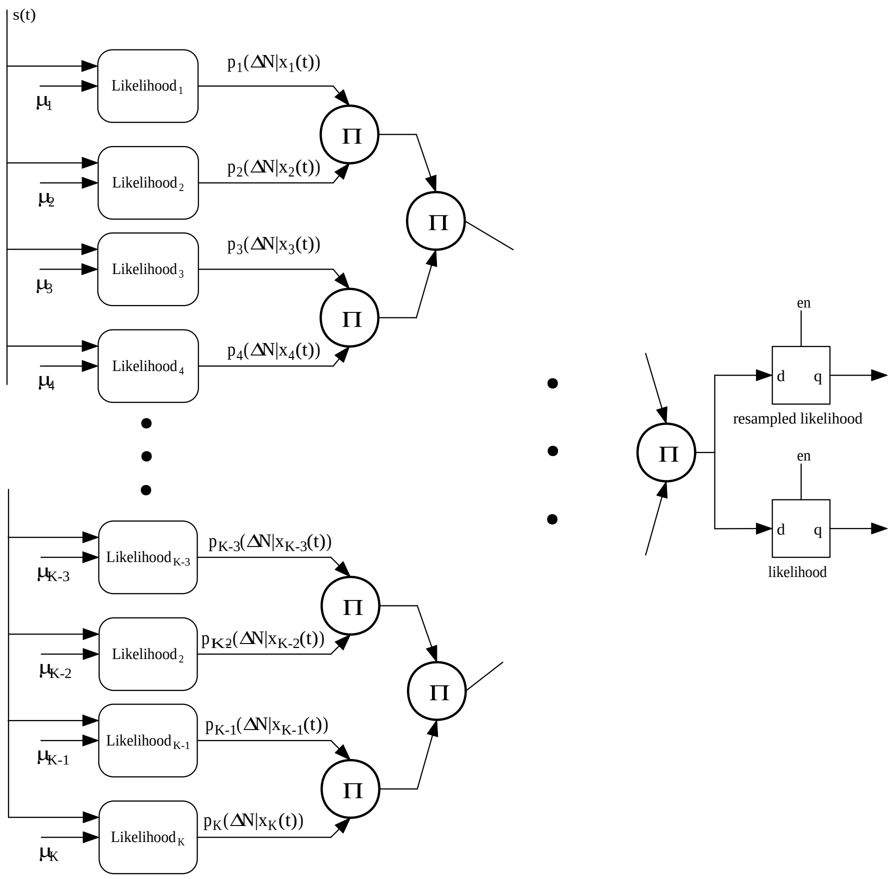

Figure 4.17: Computing a particle's likelihood as the product of parallel neuronal conditional probabilities

This term is then multiplied by the second stage weight from the previous iteration $w^r(t-1)$ to obtain an intermediate value for the first stage weight $g^r(t)$. The first stage weights for all P particles are then summed together. The intermediate values for all $g^r(t)$ are divided by this summation in order to normalize the first stage weights so they sum to unity. These values are then stored as the final first stage weights for their respective particles.

Step 3: Computing the First Stage Weight requires a total of $2B + \lceil log_2(K) \rceil + \lceil log_2(P) \rceil + 20$ clock cycles to execute. Table 4.1 summarizes the required cycles to implement each stage of this Section.

Table 4.1: Number of clock cycles required to compute and store intermediate terms to acquire the first stage weights.

function	clock cycles
$y = \alpha - \frac{(s(t)-\mu(t))^2}{\sigma^2}$	4
$\lambda = e^y$	8
store λ	1
$\lambda \Delta t$	1
store $\lambda \Delta t$	1
$(\lambda \Delta t)^{\Delta N}$	$2B$
$p(\Delta N \vert \mathbf{x}(t)) = e^{-\lambda \Delta t B} (\lambda \Delta t)^{\Delta N}$	1
$g(t) \propto w(t-1) \prod_{j=1}^{K} p(\Delta N_j \vert \mathbf{x}(t))$	$\lceil log_2(K) \rceil$
store $g(t)$	1
normalizing sum	$\lceil log_2(P) \rceil$
store normalizing sum	1
normalize	1
store normalized $g(t)$	1
total	$2B + \lceil log_2(K) \rceil + \lceil log_2(P) \rceil + 20$

4.2.6 Step 4: Resample Particles According to the First Stage Weights

A particle's weight describes how likely that particle's state estimate explains the current observations (neural firing times). The resampling process is used to retain and replicate particles with high weights and discard particles with low weights. Resampling avoids the affects of degeneracy described in Section 2.7 where only a few particles become highly weighted and the rest of the particles render no useful information.

In the resampling process, a new set of P particles is drawn from the current particle set which are approximately distributed by Equation 4.24 [52]. Since the weights have been normalized, the probability of selecting particle r as a replacement particle is $g^r(t)$. This process is know as *sampling importance resampling* (SIR).

When a large number of particles are used to implement particle filters, the resampling process becomes laborious on sequential processors. Many sequential resampling algorithms employ additional temporary memory and require $P\text{x}K + L$ clock cycles to resample all K neural parameters for all P particles using a processor with a pipeline latency of L stages [103, 104, 105]. In order to significantly decrease computation time, a parallel resampler was developed as part of this dissertation research which is capable of resampling all particles simultaneously.

The resampling process can be executed by first constructing a cumulative sum cs of the normalized first stage weights. Successive points in the summation $\{cs_1,\ cs_2,\ \ldots,\ cs_P\}$ are then used as the bin edges of a histogram which sorts P uniform random samples $\mathrm{U}\left\langle 0,1\right]$. The number of uniform samples that fall into bin^r determines the number of times particle^r is replicated. This process replicates particles with high weights and discards particles with low weights proportionally according to their respective probabilities $g^r(t)$.

To demonstrate this process, let the sample space consist of $P = 5$ particles with normalized weights $\widehat{\mathbf{w}} = [0.125\ 0.375\ 0.250\ 0.125\ 0.125]$. The cumulative sum of these weights along with 5 samples drawn from $\mathrm{U}\langle 0, 1]$ are shown in Figure 4.18. Notice that 3 of the uniform samples fall into bin 2 (the highest weighted probability) and 2 other samples fall into bins 4 and 5 weighted as 0.125. This produces a new set of P resampled particles

$$\begin{aligned} \text{intial set} &= [\text{particle}_1\ \text{particle}_2\ \text{particle}_3\ \text{particle}_4\ \text{particle}_5] \\ \text{resampled set} &= [\text{particle}_2\ \text{particle}_4\ \text{particle}_5\ \text{particle}_2\text{particle}_2] \end{aligned}$$

A parallel hardware architecture for implementing this resampling algorithm is illustrated by Figures 4.19 through 4.22. A top-level diagram of the resampler is seen in Figure 4.19. There are 3 main components to the resampler: 1) the uniform random number generator, 2) a module that computes cs and 3) a module for selecting appropriate bins. The random number generator comes from the output of one of the linear feedback shift registers defined in Section 4.2.3.1. The cumulative sum is computed over P clock cycles by the architecture shown in Figure 4.20 and passed to each of

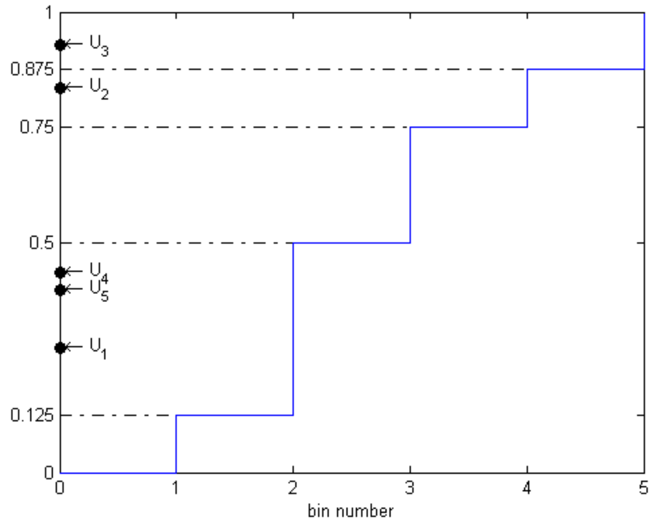

Figure 4.18: Illustration of weighted sampling with replacement.

the bin selectors which are arranged in parallel.

Selecting which bin a random value belongs to is accomplished by comparing a uniform number to each of the successive cumulative sums. If cs_m is less than the uniform random number, the comparator output is 1 otherwise it is 0. Summing the total number of 1's provides the appropriate bin number to which the random value belongs. This summation is computed by arranging fixed point adders in the same configuration as the floating point multipliers in Figure 4.17 which requires an additional $log_2(P)$ clock cycles. Each particle has its own bin selector and unique uniform random sample. The uniform random samples are stored when the enable signal is asserted. Figure 4.21 illustrates the selection process.

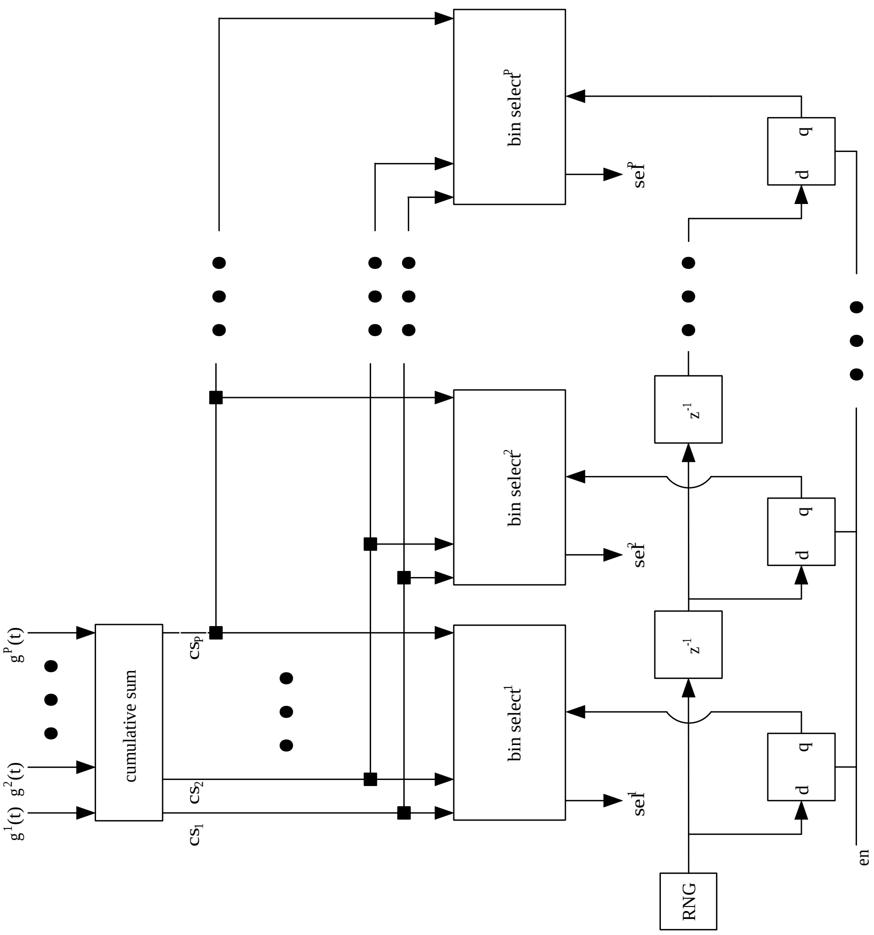

Figure 4.19: Top-level diagram of the resampler

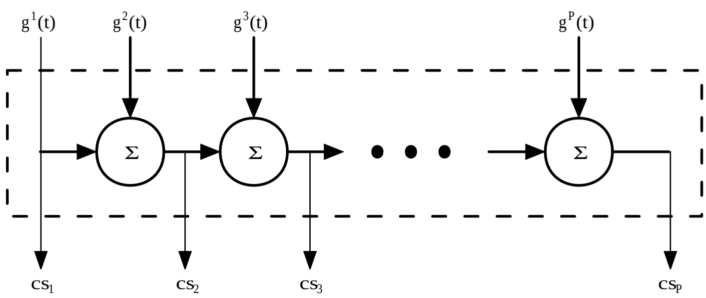

Figure 4.20: Cascaded adders to compute the cumulative sum of the normalized particle weights

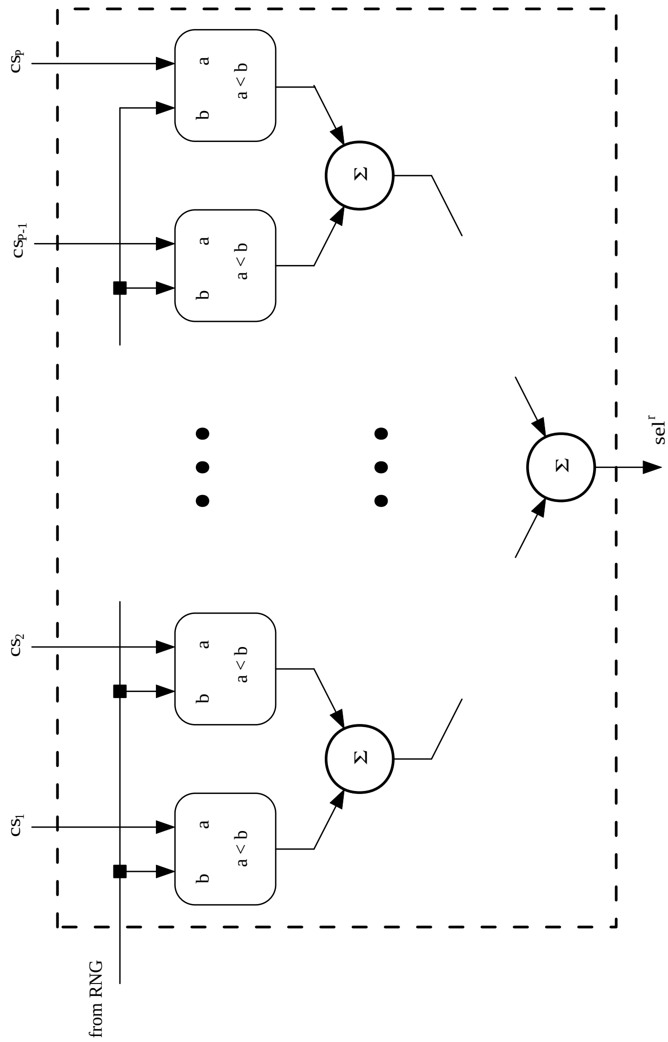

Figure 4.21: Bin Select - determine the appropriate bin for a uniform random variable by comparing it to successive cumulative sums *cs*.

In order to store resampled particles in place of the existing particles, the select lines of the multiplexers in Figure 4.7 are first set to equal 1. Before the enable signals of the state registers are set high, the proper resampled state vector elements must be presented as inputs 1 on the multiplexers in Figure 4.7. Though use of a secondary set of multiplexers, each particle is provided immediate access every other particle's state estimate in the filter. This is accomplished by using the bin selector output sel^r as the select line for the $K+1$ secondary multiplexers within the r^{th} particle.

Each of the P particles that comprise the particle filter makes use of the structure shown in Figure 4.22. By arranging P of these units in parallel the feedback loop from the resampler in Figure 4.5 is closed and allows all particles to be resampled simultaneously in $P + log_2(P) + 1$ clock cycles. This resampler is independent of the number of observed neurons, which offers a significant increase in throughput compared to the methods described in [103, 104, 105] that require $P\text{x}K + L$ clock cycles.

4.2.7 Step 5: Random Sampling

Step 5 of Section 3.3.2 is to simulate a second set of hyperparameters according to:

$$\Theta_2^r \sim \mathcal{N}(0, \sigma_2) \ for \ r = 1, \ldots, P$$

The hyperparameters are generated just as in Section 4.2.3. By setting $sel = 1$ in Figure 4.9 the RNG will output samples drawn from $\mathcal{N}(0, \sigma_2)$. Since the RNG operates at a different clock rate than the rest of the system, setting sel high after step 2 allows the tap delay line of random samples to be replaced with the new set of hyperparameters before step 5. In doing so, the resampled state vectors can immediately be updated without waiting for the

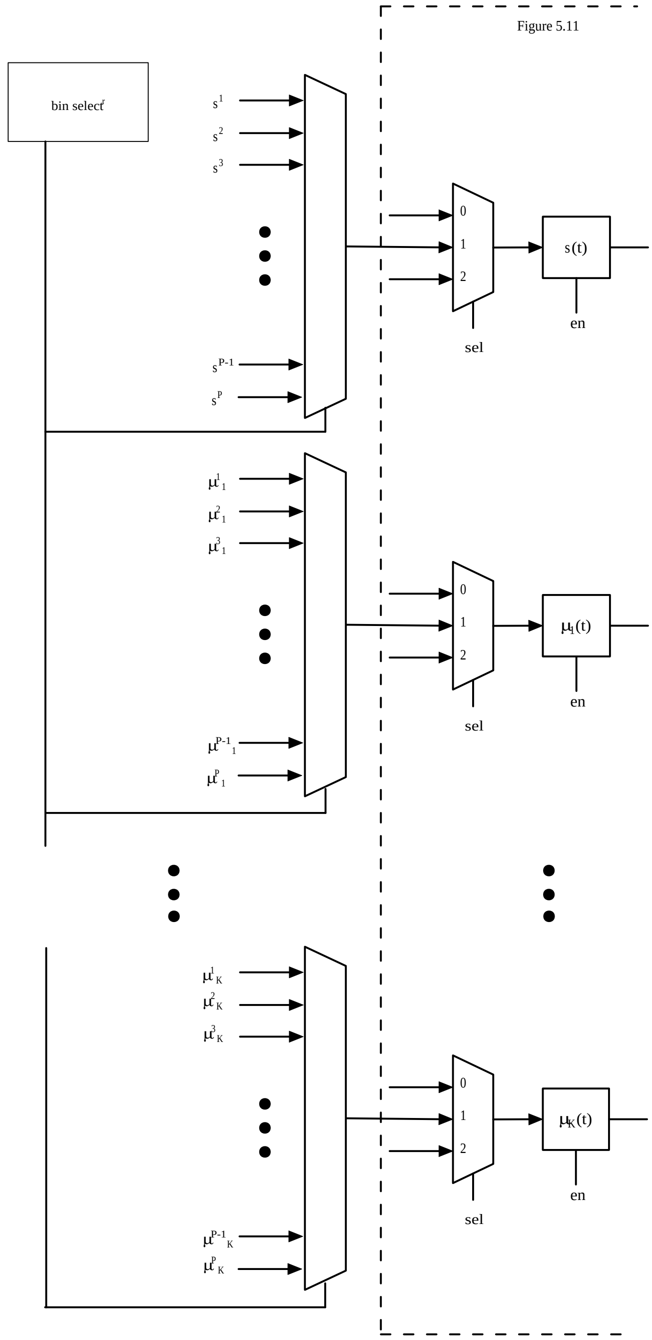

Figure 4.22: Secondary multiplexers to provide access to appropriate resampled particle.

delay line to be flushed out. Here the RNG runs at 100 MHz, which produces $2 \cdot 10^8$ samples/second. In Section 4.2.12, it is demonstrated that this is a sustainable rate to replenish all random samples.

4.2.8 Step 6: Add Randomness to the Resampled Particles

According to Equation 3.11, the resampled particles are then updated using the second set of hyperparameters Θ_2^r as

$$\mathbf{x}^r(t) = \hat{\mathbf{x}}^r(t-1) + \Theta_2^r$$

As in Section 4.2.4, the randomness is added to the state vectors by setting $sel = 0$ in Figure 4.7 and setting the enable signal high for one clock cycle. Step 6 requires a single clock cycle to execute.

4.2.9 Step 7: Compute the Second Stage Weights $w^r(t)$

According to Equation 3.12, the second stage weight $w^r(t)$ of a particle is defined to be the ratio of its resampled likelihood function over its original likelihood function

$$w^r(t) \propto \frac{p(N(t)|\hat{\mathbf{x}}^r(t))}{p(N(t)|\mathbf{x}^r(t)}$$

The resampled likelihoods are computed in parallel just as in Section 4.2.5. A particle stores its resampled likelihood in the *resampled likelihood* register as seen in Figure 4.17 by setting the appropriate *en* signal high for one cycle.

Once the resampled likelihoods are available, they are divided by their respective original likelihoods that still reside in the *likelihood* registers. This provides intermediate values for the second stage weights. The second stage weights for all P particles are then summed together. The intermediate values for all $w^r(t)$ are divided by this summation in order to normalize the second stage weights so they sum to unity. These values are then stored as the final second stage weights.

4.2.10 Step 8: Compute a Weighted Estimate

The second stage weights are used in the final step of the algorithm to provide an estimate of the state vector as a weighted sum of all particle estimates defined by Equation 3.13 as

$$\chi(t) = \sum_{r=1}^{P} w^r(t)\mathbf{x}^r(t)$$

Since $\mathbf{x}^r(t) = [s^r(t)\ \mu_1^r(t)\ \ldots\ \mu_K^r(t)]$ and the parameter of interest here is the driving signal $s(t)$, this computation can be reduced by only summing weighted estimates of the animal position rather than the entire state vector as

$$\hat{s}(t) = \sum_{r=1}^{P} w^r(t)s^r(t) \tag{4.36}$$

Weighted estimates are provided by each particle through the multiplier shown in Figure 4.7, which are then summed together and stored in a final output register. The P multiplications are executed in parallel over one clock cycle and summed over $\lceil log_2(P) \rceil$ clock cycles. One more cycle is required to store the estimate into the output register. This completes one iteration of the BAPF signal processing algorithm. At this point, the controller returns to state 2 to process the next block of neural data. The total number of clock cycles required to implement the BAPF using the parallel hardware architecture described in Sections 4.2.3 through 4.2.9 is given by Table 4.2.

Table 4.2: Number of clock cycles required to execute one iteration of the BAPF algorithm over a block period using the parallel hardware processing architecture.

	clock cycles
Step 1	0
Step 2	1
Step 3	$2B + \lceil log_2(K) \rceil + \lceil log_2(P) \rceil + 20$
Step 4	$P + \lceil log_2(P) \rceil$
Step5	0
Step 6	1
Step 7	$2B + \lceil log_2(K) \rceil + \lceil log_2(P) \rceil + 21$
Step 8	$\lceil log_2(P) \rceil + 2$
Total	$P + 4 \lceil log_2(P) \rceil + 2 \lceil log_2(K) \rceil + 4B + 45$

4.2.11 Hardware Verification

Assembly of the hardware architecture is facilitated by the Xilinx System Generator™ for DSP environment. This EDA tool provides a platform for accelerated design of high performance DSP systems using Xilinx FPGAs. System Generator is a Simulink add-on blockset that provides automatic HDL generation directly from the MATLAB/Simulink® environment. The software offers direct synthesis of designs into bitstream files for programming FPGAs. Additionally, designs can be co-simulated with hardware in the loop. This allows designs to run in hardware while comparing the output to the software simulations.

Simulation of a hardware prototype designed using the System Generator software requires a significant amount of memory. Simulating a hardware model of the filter discussed in Section 3.4 is not feasible. Therefore, a scaled version of the BAPF signal processing algorithm was constructed as a proof of concept. Simulating this design requires approximately 2.5 GB of physical memory. The operating system on which System Generator runs can only recognize up to 3 GB.

Here, $K = 10$ neurons and $P = 4$ particles were used to predict 30 seconds of animal position $s(t)$ as well as estimate all K time-varying receptive field centers $\mu_j(t)$ from a simulated data set to verify the hardware design. The block size is $B = 25$ samples with a sampling period of $\Delta t = 2$ ms. The standard deviations for Step 1 were set to $\sigma_1 = 6$ for the $s(t)$ estimates and $\sigma_1 = 0.125$ for the $\mu_j(t)$ estimates. The standard deviations for Step 5 were set to $\sigma_2 = 1$ for the $s(t)$ estimates and $\sigma_2 = 0.0625$ for the $\mu_j(t)$ estimates.

To verify the performance of the single precision hardware architecture, the average mean square error (MSE) between the true position and 10 estimated positions obtained from the hardware design was compared to the average MSE from 10 software simulations. These results are presented in Table 4.3. Figure 4.23 demonstrates the ability of the hardware configuration to accurately predict the true position for a single simulation.

Table 4.3: Average MSE between between the true position and predicted positions obtained from both the hardware and software outputs.

	Hardware	Software
MSE	109.7	107.6

4.2.12 Throughput Comparison

The execution time required to obtain an estimate from a single iteration of the BAPF algorithm using the parallel hardware design running at 500 MHz was compared to the execution time required by a sequential Intel Core2 Duo running at 2.5 GHz. The hardware processing times were determined through Table 4.2. The software throughput was determined by computing the average execution time over 20 trials for each configuration.

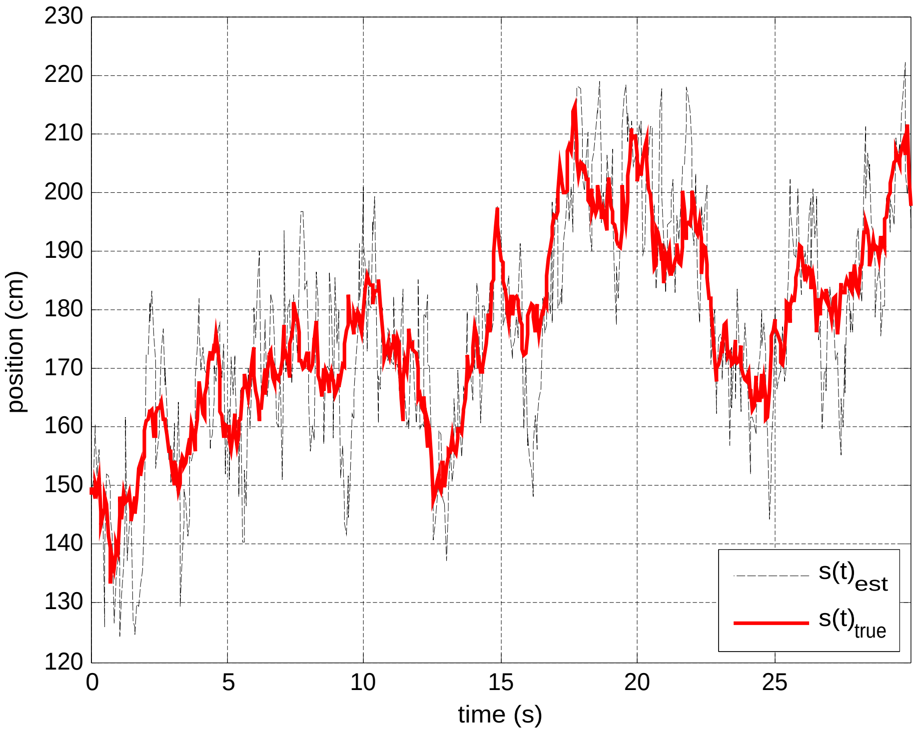

Figure 4.23: A single experiment of position tracking using the parallel hardware design

In each experimental configuration, one of three parameters was varied. In the first experiment, the number of particles was set to $P = 500$, the number of neurons was set to $K = 75$ and the block size B was varied from 20 to 200. In the second experiment, $P = 500$, $B = 100$ and the number of neurons K was varied from 25 to 250. In the final experiment, $K = 75$, $B = 100$ and P varied from 50 to 5000. In all cases $\Delta t = 0.001$ sec. The computation times from each experiment are shown in Figures 4.24 through 4.26.

It is apparent that the FPGA hardware implementation has significantly

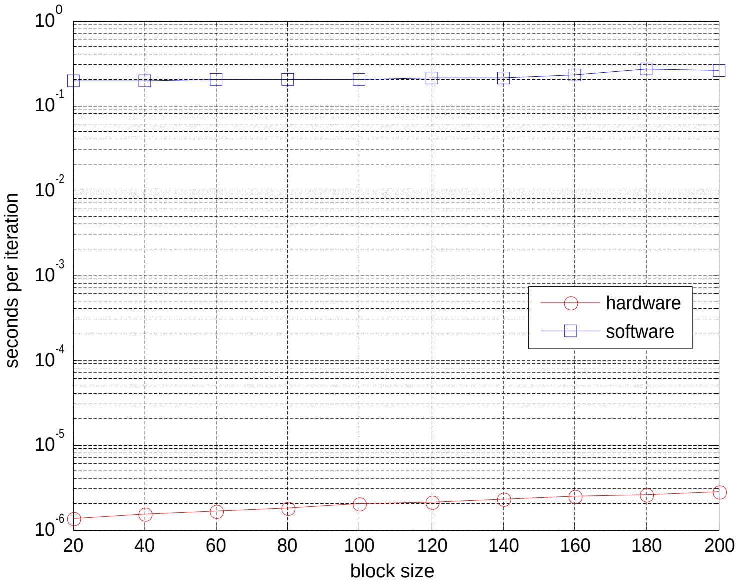

Figure 4.24: Hardware-Software throughput comparison as a function of block size. The number of neurons is $K = 75$ and the number of particles is $P = 500$ for all 10 experiments.

reduced execution time compared to that of the sequential microprocessor system, especially when the number of particles is increased. The decrease in execution time is primarily due to the parallel computations and the intrinsic pipelined architecture of the FPGA. By avoiding the use of instruction sets and memory storage, this hardware implementation has demonstrated how custom processing modules can improve throughput exponentially over a general purpose processing implementation.

To utilize the BAPF for real-time estimation of neural driving signals, computing an estimate for an observed block duration must occur before the next block of samples arrives. This demands that the time needed to acquire

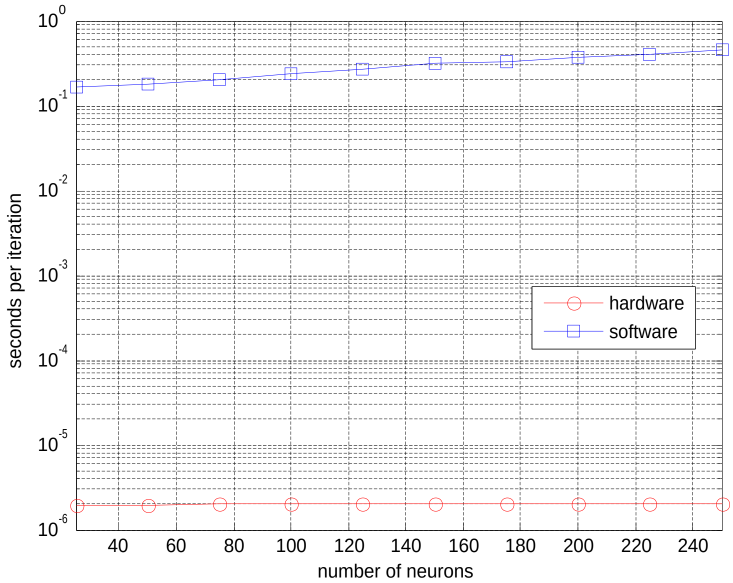

Figure 4.25: Hardware-Software throughput comparison as a function of the number of observed neurons. The number of particles is $P = 500$ and the block size is $B = 100$ for all 10 experiments.

this estimate be less than $B\Delta t$. In these experiments, the sampling rate of the neurons was $\Delta t = 0.001$ seconds. The affective range of block sizes was found to be $50 < B < 100$ resulting in block periods of $0.05 < \mathrm{T}_B < 0.1$ seconds. From Figure 4.26, it is evident that this constraint can only be met when $P < 200$. However, it was demonstrated in Section 3.5 that the number particles needed to accurately decode neural firing times and predict elbow angle ϕ be $P \gg 200$.

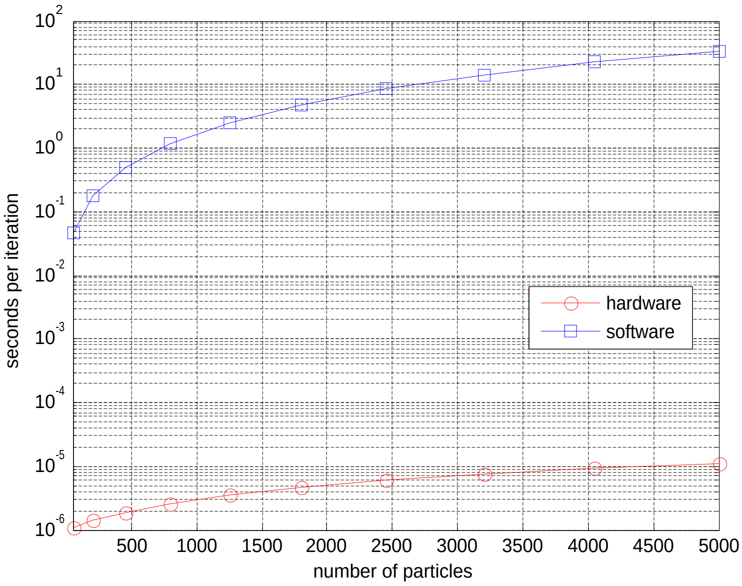

Figure 4.26: Hardware-Software throughput comparison as a function of the number of particles. The number of neurons is $K = 75$ and the block size is $B = 100$ for all 10 experiments.

4.2.13 FPGA Synthesis

The Xilinx XC5VSX50T FPGA is the target device for synthesizing the BAPF architecture of Section 4.2.11 because of its current availability. It is found that this specific filter configuration requires more logic slices than are available in the 8200 slices of the XC5VSX50T device. Table 4.4 lists the resources required for the hardware modules comprising the BAPF architecture described earlier in this Chapter. These values can be used to determine an appropriate FPGA that provides sufficient resources to implement a specific parallel BAPF filter architecture.

Each neuron in the observed ensemble contributes 1 element to the state

Table 4.4: Synthesis results for hardware utilization of each of the modules.

Module	Logic Slices
Controller (FSM)- Figure 4.5	2653
RNG-Figure 4.9	3506
State Vector-Figure 4.10	34/parameter
Exp arg-Figure 4.11	91
Exp-Figure 4.15	2500

vector estimated by each particle. According to Table 4.4, this entails 34 slices for each neuron in each particle. Each observed neuron also requires 2591 slices to compute the exponential function composed of the exponential argument and the expomemtial function of Figures 4.11 and 4.15. Therefore, each neuron requires 2625 logic slices in each particle. The resources consumed by the RNG and FSM are an initial resource cost (6159 slices) in developing a hardware configuration of the BAPF. The number of particles P and number of neurons K will dictate the total number of slices:

$$\text{Slices}_{\text{total}} = 6159 + PK(2625) \tag{4.37}$$

The configuration of Section 4.2.11 would therefore require approximately 120000 slices, with $P = 4$ and $K = 10$.

Figure 4.27 shows the number of logic slices provided by the various Xilinx Virtex family FPGAs, with the Virtex first released in 2001 and Virtex-7 released in 2011. This suggests that Virtex-6 devices with at most 380000

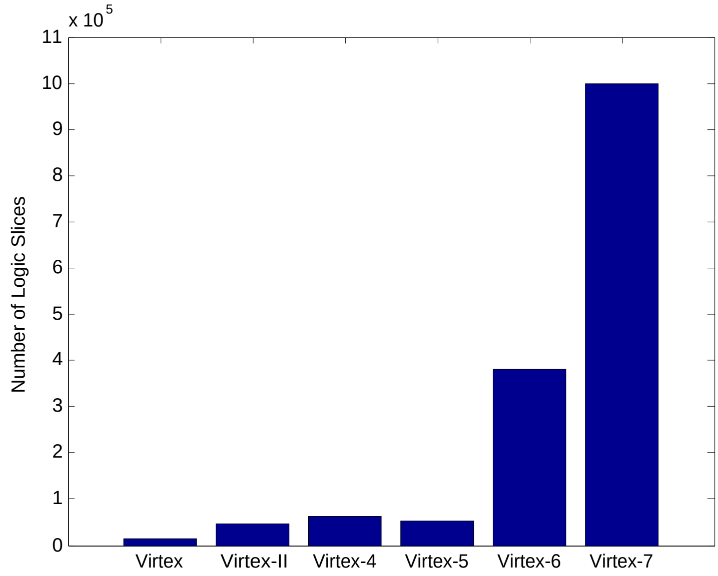

Figure 4.27: Number of logic slices residing on current and next generation Virtex FPGA families.

slices would accommodate the intended design as wells as larger configurations. Some Virtex-7 FPGAs are comprised of over 1 million slices. These next generation FPGAs indicate that the number of slices contained within a single device will continue to grow rapidly. It is evident from this trend that more neurons and more particles can be incorporated into a design as new FPGAs are developed.

CHAPTER 5

Discussion

There are many existing methods used to decode neural spike trains. These include both linear estimators such as the Weiner filter, the least mean squares (LMS) algorithm and the Kalman filter. Nonlinear estimation algorithms consist of the extended Kalman filter, the unscented Kalman filter, point process adaptive filters and particle filters. Although these approaches represent important strides towards understanding the underlying neural processes, their drawbacks limit their applicability and stability.

For example, the Weiner filter and Kalman filters can only be applied to linear environments with independent Gaussian noise processes. These estimators also require matrix inversions in their implementation. When low neural firing activity is encountered, the observation matrices become sparse and may become singular. This causes the filters to become unstable and the filter estimates diverge. The LMS algorithm eliminates matrix inversion computations by iteratively converging to the Weiner solution. However, if the input data is sparse, an indeterminate number of iterations is needed to converge to such a solution resulting in long training periods.

The extended Kalman filter, unscented Kalman filter and stochastic state point process adaptive filter have been successfully applied to nonlinear decoding problems. Unfortunately, these decoders still require matrix inversions which may not be possible. The estimators also assume independent Gaussian noise.

The sampling importance resampling (SIR) particle filtering eliminates the need to perform any matrix inversions. Particle filters attempt to approximate the probability distribution of the system state by random measures (particles) drawn from samples around the mean of the estimated state. The Bayesian auxiliary particle filter (BAPF) improves on the SIR methodology by incorporating a two-stage weighting process.

The BAPF is applied to the problem of neural decoding. In Section 3.4, the BAPF predicted animal position by observing the firing times of simulated dynamic hippocampal place cells. It is shown that the BAPF resulted in less mean squared error (MSE) than both the linear Wiener filter and the SIR particle filter. These preliminary results indicated that the BAPF would be a preferable algorithm to decode neural activity for the brain machine interface.

In Section 3.5, the BAPF is employed to predict elbow angle ϕ during a center-out reaching task. Neural firing activity was recorded through a micro-electrode array. The observed neurons were assumed to be sinusoidally tuned according to Equation 3.15 with an expected Poisson arrival rate of $\lambda(t)$. The BAPF was shown to yield more accurate estimates of ϕ than the Kalman filter. This was due to some possible deficiencies in the Kalman filter such as the assumed Gaussian environment and the inversion of a sparse matrix in Equation 2.11.

A practical limitation of the BAPF for real-time neural decoding in BMIs using a sequential processor is the computational complexity. It was shown in Section 3.4 that the decoding accuracy of the BAPF improves as the number of observed neurons in the ensemble increases as well as the number of particles used in the decoding process. This results in long processing times. Therefore, in Chapter 4 a parallel field programmable gate array (FPGA) hardware architecture is presented that exponentially increases throughput over conventional methods.

This increase in throughput is achieved by computing the state estimate $\hat{\mathbf{x}}^r(t)$ of all P particles, $r = 1 \ldots P$, simultaneously in parallel. By utilizing the parallel pipelined random number generator of Section 4.2.3.4, all elements of $\hat{\mathbf{x}}^r(t)$ can be resampled according to Equation 3.6 and 3.10 in a single clock cycle. Furthermore, the conditional probability of Equation 3.9 requires two executions of the exponential function $f(x) = e^x$. The hardware module of Section 4.2.5.1 reduces the number of clock cycles required perform this process over existing methodologies.

The parallel computations are facilitated by reconfigurable FPGA hardware modules. This enables the hardware design to accommodate a variety of neural ensembles corresponding to a diverse set of neural tuning functions. New BAPF filter designs can be implemented by replacing the *state vector* and *compute likelihood* modules in Figure 4.7 with a different module.

The BAPF has been shown to be an effective method for tracking non-stationary neural parameters and for estimating the common kinematic signal encoded by an ensemble of neurons. This method is shown here to outperform the Weiner filter, the Kalman filter and the SIR particle filter. This suggests that the BAPF is a favorable method for decoding neural signals in the BMI. The computational burden of the BAPF is alleviated by a parallel FPGA hardware architecture. This architecture increases throughput sufficiently to allow the BAPF to be applied in real-time neural signal processing.

5.1 Future Work

5.1.1 Tuning Function Models

In Section 3.5, it was assumed that each observed neuron fired maximally when the elbow angle of the subject was at a particular angle or preferred direction ϕ_p. The firing behavior of each neuron was characterized by the sinusoidal tuning function of Equation 3.15. Although positive results were obtained under this assumption, it may be possible that firing behavior of these neurons can be better described by a different tuning model.

Other functions that may better quantify the firing behavior of neurons tuned to a preferred angle could be defined by a circular distribution or a bounded distribution on the interval $[0, 2\pi]$. Even though the following functions are used to define the probability distributions of random variables, they can be scaled to define a neural tuning function. These functions include but are not limited to:

1. the von Mises distribution [106]

2. the Kumaraswamy distribution [107]

3. the wrapped Cauchy distribution [108]

4. or the beta distribution [109]

5.1.2 Variance-Adaptive BAPF

The standard deviation σ chosen for the random distributions in Sections 3.3 and 3.5 were required to be large enough to capture significant changes in the state vector but small enough to provide accurate estimates in $\hat{\mathbf{x}}(t)$. For the most accurate $\hat{\mathbf{x}}(t)$, an optimal choice of σ in Equations 3.6 and 3.10 is required.

It was demonstrated in Section 3.4 that neural plasticity causes changes in neural tuning parameters. These changes are reflected in the dynamic state vector $\mathbf{x}(t)$. If a maximum rate at which a plastic neuron evolves can be determined, then implications can be made as to a maximum choice for σ. If the rate of change can be correlated to specific affecter signals $s(t)$, then the value of σ could be set as a function of $s(t)$ resulting in better estimates of $\mathbf{x}(t)$.

REFERENCES

[1] J. Sanchez and J. Pricipe, *Brain-Machine Interface Engineering.* Princeton, New Jersey: Morgan and Claypool, 2007.

[2] J. Wanq, M. Gulari, P. Bhatti, B. Arcand, K. Beach, C. Friedrich, and K. Wise, "A cochlear electrode array with built-in position sensing," in *Micro Electro Mechanical Systems, 2005. MEMS 2005. 18th IEEE International Conference on*, pp. 786–789, Jan.-3 Feb. 2005.

[3] J. Rizzo, J. Wyatt, J. Loewenstein, S. Kelly, and D. Shire, "Perceptual efficacy of electrical stimulation of human retina with a microelectrode array6 during short-term surgical trials," *Investigative ophthalmology and visual science*, vol. 44, no. 12, pp. 5362–5369, 2003.

[4] D. Taylor, S. Tillery, and A. Schwartz, "Direct cortical control of 3d neuroprosthetic devices," *Science*, vol. 296, no. 5574, pp. 1829–1832, 2002.

[5] S. Haykin, *Adaptive Filter Theory second edition*. Englewood Cliffs, New Jersey: Prentice Hall, 1984.

[6] J. Sanchez, D. Erdogmus, Y. Rao, S. P. Kim, M. Nicolelis, J. Wessberg, and J. Principe, “Interpreting neural activity through linear and nonlinear models for brain machine interfaces,” in *Engineering in Medicine and Biology Society, 2003. Proceedings of the 25th Annual International Conference of the IEEE*, vol. 3, pp. 2160–2163 Vol.3, Sept. 2003.

[7] J. Wolpaw and D. McFarland, “Control of a two-dimensional movement signal by a noninvasive brain-computer interface in humans,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101, no. 51, pp. 17849–17854, 2004.

[8] W. Wu, M. Black, Y. Gao, E. Bienenstock, M. Serruya, and J. Donoghue, “Inferring hand motion from multi-cell recordings in motor cortex using a kalman filter,” in *SAB02Workshop on Motor Control in Humans and Robots: On the Interplay of Real Brains and Artificial Devices*, pp. 66–73, 2002.

[9] WeiWu, Y. Gao, E. Bienenstock, J. Donoghue, and M. Black, “Bayesian population decoding of motor cortical activity using a kalman filter,” *Neural Computation*, vol. 18, no. 1, pp. 80–118, 2006.

[10] F. H. Guenther, J. S. Brumberg, E. J. Wright, A. Nieto-Castanon, J. A. Tourville, M. Panko, R. Law, S. A. Siebert, J. L. Bartels, D. S. Andreasen, P. Ehirim, H. Mao, and P. R. Kennedy, “A wireless brain-machine interface for real-time speech synthesis,” *PLoS ONE*, vol. 4, p. e8218, 12 2009.

[11] Z. Li, J. O’Doherty, T. Hanson, M. Lebedev, C. Henriquez, and M. Nicolelis, “Unscented kalman filter for brain-machine interfaces,” *PLoS One*, vol. 4, no. 7, p. e6243, 2009.

[12] E. Brown, D. Nguyen, L. Frank, M. Wilson, and V. Solo, “An analysis of neural receptive fields by point process adaptive filtering,” *PNAS*, vol. 98, no. 21, pp. 12261–122266, 2001.

[13] U. Eden, L. Frank, R. Barbieri, V. Solo, and E. Brown, “Dynamic analysis of neural encoding by point process adaptive filtering,” *Neural Computation*, vol. 16, pp. 971–998, 2004.

[14] A. Brockwell, A. Rojas, and R. Kass, “Recursive bayesian decoding of motor cortical signals by particle filterin,” *Journal of Neurophisology*, vol. 91, no. 4, pp. 1899–1907, 2004.

[15] Y. Gao, M. Black, E. Bienenstock, S. Shoham, and J. Donoghue, “Probabilistic inference of hand motion from neural activity in motor

cortex," *Advances in Neural Information Processing Systems*, vol. 14, pp. 213–20, 2002.

[16] B. Ristic, S. Arulampalam, and Neil, *Beyond the Kalman Filter: Particle Filters for Tracking Applications.* Norwood, Massachusetts: Artech House, 2004.

[17] J. Wessberg, C. Stambaugh, J. Kralik, P. Beck, M. Laubach, J. Chapin, J. Kim, J. Biggs, M. Srinivasan, and M. Nicolelis, "Real-time prediction of hand trajectory by ensembles of cortical neurons in primates," *Nature*, vol. 408, pp. 361–365, November 2000.

[18] N. Hatsopoulos, J. Joshi, and J. O'Leary, "Decoding Continuous and Discrete Motor Behaviors Using Motor and Premotor Cortical Ensembles," *J Neurophysiol*, vol. 92, no. 2, pp. 1165–1174, 2004.

[19] L. H. et al, "Neural Ensemble Control of Prosthetic Devices by a Human with Tetraplegia," *Nature*, vol. 442, no. 13, pp. 164–171, 2006.

[20] J. Carmena, M. Lebedev, R. Crist, J. O'Doherty, D. Santucci, D. Dimitrov, P. Patil, C. Henriquez, and M. Nicolelis, "Learning to control a brainmachine interface for reaching and grasping by primates," *PLoS Biol*, vol. 1, p. e42, 10 2003.

[21] M. Lebedev, R. Crist, and M. Nicolelis, "Brain Machine Interfaces

to Restore Neurological Function and Probe Neural Circuits," *Nature Reviews Neuroscience*, vol. 4, pp. 417–422, 2003.

[22] *Handbook of Neural Engineering.* Hoboken, New Jersey: Wiley and Sons, 2007.

[23] B. Widrow and M. Hoff, "Adaptive switching circuits," *Neurocomputing: foundations of research*, pp. 123–134, 1988.

[24] R. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME Journal of Basic Engineering*, no. 82 (Series D), pp. 35–45, 1960.

[25] R. Brown and P. Hwang, *Introduction to Random Signals and Applied Kalman Filtering.* Hoboken, New Jersey: Wiley and Sons, 1997.

[26] S. Kay, *Fundamentals of Statistical Signal Processing: Estimation Theory.* Upper Saddle River, New Jersey: Prentice Hall, 1993.

[27] A. Sayed, *Fundamentals of Adaptive Filtering.* Hoboken, New Jersey: Wiley and Sons, 2003.

[28] M. Black, E. Bienenstock, J. Donoghue, M. Serruya, W. Wu, and Y. Gao, "Connecting brains with machines: The neural control of 2d cursor movement michael j. black," 2003.

[29] W. Wu, M. Black, Y. Gao, E. Bienenstock, M. Serruya, A. Shaikhouni, and J. Donoghue, "Neural decoding of cursor motion using a kalman filter," 2003.

[30] Y. Gao, E. Bienenstock, J. Donoghue, M. Black, and W. Wu, "Bayesian population decoding of motor cortical activity using a kalman filter," 2005.

[31] W. Wu, A. Shaikhouni, J. Donoghue, and M. Black, "Closed-loop neural control of cursor motion using a Kalman filter," *Engineering in Medicine and Biology Society, 2004. EMBC 2004. Conference Proceedings. 26th Annual International Conference of the*, vol. 2, 2004.

[32] W. Wu, M. Black, Y. Gao, E. Bienenstock, M. Serruya, and J. Donoghue, "Inferring hand motion from multi-cell recordings in motor cortex using a kalman filter," in *SAB02Workshop on Motor Control in Humans and Robots: On the Interplay of Real Brains and Artificial Devices*, pp. 66–73, 2002.

[33] Z. Li, J. O'Doherty, T. Hanson, M. Lebedev, C. Henriquez, and M. Nicolelis, "Nth order kalman filter improves the performance of a brain machine interface for reaching," in *Society for Neuroscience 2007 Annual Meeting, San Diego, California*, 2007.

[34] W. Wu, M. Black, D. Mumford, Y. Gao, E. Bienenstock, and

J. Donoghue, “Modeling and decoding motor cortical activity using a switching kalman filter,” *IEEE Transactions on Biomedical Engineering*, vol. 51, pp. 933–942, 2004.

[35] H. Sorenson, *Kalman filtering:Theory and application.* IEEE Press Selected Reprint Series, The Institute of Electrical and Electronics Engineers.

[36] T. Lefebvre, H. Bruyninckx, J. Schutter, T. Lefebvre, and H. Bruyninckx, “Kalman filters for nonlinear systems: a comparison of performance.,” tech. rep., International Journal of Control, 2001.

[37] S. Schiff and T. Sauer, “Kalman filter control of a model of spatiotemporal cortical dynamics,” *Journal of Neural Engineering*, vol. 5, no. 1, p. 1, 2008.

[38] B. Quine, J. Uhlmann, and H. Durrant-Whyte, “Implicit jacobian for linearised state estimation in nonlinear systems,” in *American Control Conference, 1995. Proceedings of the*, vol. 3, pp. 1645–1646 vol.3, Jun 1995.

[39] J. Speyer and W. Chung, *Stochastic Process, Estimation and Control.* Philadelphia, Pennsylvania: SIAM, 2008.

[40] D. Watkins, *Fundamentals of Matrix Computaion.* Hoboken, New Jersey: John Wiley and Sons, 2002.

[41] P. Dayan and L. Abbott, *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems.* Cambridge, Massachusetts: MIT Press, 2002.

[42] J. Smithies, *The Dynamic Neuron.* Cambridge, Massachusetts: MIT Press, 2002.

[43] J. Kaas, *The Mutable Brain.* Amsterdam, The Netherlands: Harwood Academic Publishers, 2001.

[44] G. Filogamo, A. Vernadakis, and A. Privat, *Brain Plasticity: Development and Aging.* New York, New York: Plenum Press, 1997.

[45] P. Hutenlocher, *Neural Plasticity.* Cambridge, Massechusetts: Havard University Press, 2002.

[46] M. S. Gazzaniga, ed., *The cognitive neurosciences.* Cambridge, MA: MIT Press, 1995.

[47] J. O'Keefe and L. Nadel, *The Hippocampus as a Cognitive Map.* New York, New York: Oxford University Press, 1978.

[48] D. Snyder and M. Miller, *Random Point Processes in Time and Space.* New York, New York: Springer, 1991.

[49] D. Daley and D. Vere-Jones, *An Introduction to the Theory of Point Processes: Volume 1 Elementary Theory and Methods, Second Edition.* New York, New York: Springer, 2005.

[50] E. Brown, R. Barbieri, U. Eden, and L. Frank, "Liklihood methods for neural spike train data analysis," *Computational Neuroscinece*, pp. 253–286, 2004.

[51] J. Mountney, M. Sobel, and I. Obeid, "Bayesian auxiliary particle filters for estimating neural tuning parameters," in *Engineering in Medicine and Biology Society, 2009. EMBC 2009. Annual International Conference of the IEEE*, pp. 5705–5708, Sept. 2009.

[52] N. Gordon, D. Salmond, and A. Smith, "Novel approach to nonlinear/non-gaussian bayesian state estimation," *Radar and Signal Processing, IEE Proceedings F*, vol. 140, pp. 107–113, Apr 1993.

[53] P. Moral, A. Doucet, and A. Jarsa, "Sequential monte carlo for bayesain computation," *Bayesain Statistecs*, vol. 8, 2007.

[54] A. Ergun, R. Barbieri, U. Eden, M. Wilson, and E. Brown, "Construction of point process adaptive filter algorithms for neural systems using sequential monte carlo methods," *Biomedical Engineering, IEEE Transactions on*, vol. 54, pp. 419–428, March 2007.

[55] Y. Wang, A. Paiva, and J. Principe, "A monte carlo sequential estimation for point process optimum filtering," in *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, pp. 1846–1850, 0-0 2006.

[56] Y. Wang, A. Paiva, J. Principe, and J. Sanchez, "A monte carlo sequential estimation of point process optimum filtering for brain machine interfaces," in *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*, pp. 2250–2255, Aug. 2007.

[57] J. Candy, *Bayesian Signal Processing: Classical, Modern and Particle Filtering Methods.* Hoboken, New Jersey: Wiley and Sons, 2009.

[58] J. Liu and M. West, "Combined parameter and state estimation in simulation-based filtering," in *Sequential Monte Carlo Methods in Practice. New York* (A. Doucet, J. Freitas, and N. Gordon, eds.), Springer-Verlag, New York, 2000.

[59] M. Jung, S. Wiener, and B. McNaughton, "Comparison of spatial firing characteristics of units in dorsal and ventral hippocampus of the rat," *The Journal of Neuroscience*, vol. 14, no. 12, pp. 7347–7356, 1994.

[60] J. Shen, C. A. Barnes, B. L. McNaughton, W. E. Skaggs, and K. L. Weaver, "The Effect of Aging on Experience-Dependent Plasticity of

Hippocampal Place Cells," *J. Neurosci.*, vol. 17, no. 17, pp. 6769–6782, 1997.

[61] D. Won, P. Tiesinga, C. Henriquez, and P. Wolf, "An analytic comparison of the information in sorted and non-sorted cosine-tuned spiking activity," *Journal of Neural Engineering*, vol. 4, no. 3, pp. 322–335, 2007.

[62] K. Ganguly, L. Secundo, G. Ranade, A. Orsborn, E. F. Chang, D. F. Dimitrov, J. D. Wallis, N. M. Barbaro, R. T. Knight, and J. M. Carmena, "Cortical representation of ipsilateral arm movements in monkey and man," *J. Neurosci.*, vol. 29, no. 41, pp. 12948–12956, 2009.

[63] P. Inc, "'multi channel acquistition processor,"

[64] A. B and G. A, "Directional tuning profiles of motor cortical cells," *Neuroscience research*, vol. 36, pp. 73–79, 2000.

[65] E. Todorov, "Cosine tuning minimizes motor errors," *Neural Computation*, vol. 14, no. 6, pp. 1233–1260, 2002.

[66] M. K. Pitt and N. Shephard, "Filtering via simulation: Auxiliary particle filters," *Journal of the American Statistical Association*, vol. 94, no. 446, pp. pp. 590–599, 1999.

[67] J. Mountney, D. Silage, and I. Obeid, "Parallel field programmable

gate array particle filtering architecture for real-time neural signal processing," in *Annual International Conference of the IEEE on Engineering in Medicine and Biology Society, 2010. EMBC 2010.*, 2010.

[68] S. Brown, *Field Programmable Gate Arrays.* Dordecht, The Netherlands: Kluwer Academic Publishers, 1992.

[69] C. Bobda, *Introduction to Reconfigurable Computing.* Dordecht, The Netherlands: Springer, 2007.

[70] L. Wanhammar, *DSP Integrated Circuits.* London, UK: Academic Press, 1999.

[71] P. Lekkas, *Network Processors: Architectures, Protocols and Platforms.* New York, New York: McGraw-Hill, 2003.

[72] R. Esposito, J. Mountney, L. Bai, and D. S. , "Parallel architecture implementation of a reliable (k,n) image sharing scheme," in *Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on*, pp. 27–34, Dec. 2008.

[73] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1 –58, 29 2008.

[74] S. Mitra, *Digital Signal Processing: A Computer Based Approach 2nd Edition.* New York, New York: McGraw-Hill, 2001.

[75] T. Newman and P. Odell, *The Generation of Random Variates.* New York, New York: Hafner Publishing Company, 1971.

[76] L. Devroye, *Non-Uniform Random Variate Genereation.* New York, New York: Springer-Verlag, 1986.

[77] A. Menezes, P. V. Oorschot, and S. Vanstone, *Handbook of Applied Cryptography.* Boca Raton, Florida: CRC Press, 2001.

[78] N. Zierler, "Several binary sequence generators," tech. rep., Lincoln Labs MIT 95, 1955.

[79] W. Gilbert and W. Nicholson, *Modern Algebra with Applications 2nd Edition.* Hoboken, New Jersey: Wiley and Sons, 2004.

[80] F. Ionescu, "Theory and practice of a fully controllable white noise generator," in *Semiconductor Conference, 1996., International*, vol. 2, pp. 319–322 vol.2, Oct 1996.

[81] J. Carletta and C. Papachristou, "Structural constraints for circular self-test paths," in *VLSI Test Symposium, 1995. Proceedings., 13th IEEE*, pp. 486–491, Apr-3 May 1995.

[82] L. Colavito and D. Silage, "Efficient pga lfsr implementation whitens pseudorandom numbers," *International Conference on Reconfigurable Computing and FPGAs, 2009. ReConFig '09.*, pp. 308 –313, Dec. 2009.

[83] S. Haykin, *Communications Systems 4th Edition.* New York, New York: Wiley and Sons, 2001.

[84] S. Ghahramani, *Fundamentals of Probability.* Upper Saddle River, New Jersey: Prentice Hall, 2000.

[85] H. Stark and J. Woods, *Probability and Random Processes with Applications to Signal Processing Third Edition.* Upper Saddle River, New Jersey: Prentice Hall, 2002.

[86] G. Cooper and C. McGillem, *Probability Methods of Signal and System Analysis Third Edition.* New York, New York: Oxford University Press, 1999.

[87] C. Therrien and M. Tummala, *Probability for Electrical and Computer Engineers.* Boca Raton, Florida: CRC Press, 2004.

[88] C. van der Laan and N. Temme, *Calculation of Special Functions: the Gamma Function, the Exponential Integrals and Error-like Functions.* Amsterdam, The Netherlands: Mathematisch Centrum, 1984.

[89] T. S. of the Computation Laboratory, *The Annals of the Computation Laboratory of Harvard University XXIII: Tables of the Error Function and Its First Twenty Derivatives.* Cambridge, Massachusetts: Harvard University Press, 1952.

[90] L. Colavito and D. Silage, "Composite look-up table gaussian pseudo-random number generator," pp. 314 –319, December 2009.

[91] G. Box and M. Muller, "A note on the generation of random normal deviates," *Ann.˜Math.˜Stat.*, vol. 29, no. 2, pp. 610–611, 1958.

[92] J. Volder, "The cordic trigonometeric computing technique," *IRE Transactions on Electronic Computers*, pp. 330–334, 1959.

[93] J. E. Volder, "The birth of cordic," *J. VLSI Signal Process. Syst.*, vol. 25, no. 2, pp. 101–105, 2000.

[94] J. Reed, *Software Radio: A Modern Approach to Radio Engineering.* Upper Saddle River: Prentice Hall, 2002.

[95] G. Marsaglia, "Normal (gaussian) random variable for supercomputers," *J. Supercomput.*, vol. 5, pp. 49–55, June 1991.

[96] K. Lange, *Numerical Analysis for Stasticians.* New York, New York: Springer-Verlage, 1999.

[97] B. Morgan, *Elements of Simulation.* Boca Raton, Florida: CRC, 1995.

[98] R. Jain and S. Iyengar, *Advanced Engineering Mathemtaics 3rd Edition.* Oxford, UK: Alpha Science, 2007.

[99] D. Jordan and P. Smith, *Matheatical Techniques 4th Edition.* Oxford, UK: Oxford Press, 2008.

[100] M. Ciletti, *Advanced Digital Design with the Verilog HDL*. Upper Saddle River, NJ: Prentice Hall, 2003.

[101] C. Doss and J. Riley, R.L., "Fpga-based implementation of a robust ieee-754 exponential unit," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pp. 229 – 238, april 2004.

[102] M. Wielgosz, E. Jamro, and K. Wiatr, "Highly efficient structure of 64-bit exponential function implemented in fpgas," in *Reconfigurable Computing: Architectures, Tools and Applications* (R. Woods, K. Compton, C. Bouganis, and P. Diniz, eds.), vol. 4943 of *Lecture Notes in Computer Science*, pp. 274–279, Springer Berlin / Heidelberg, 2008.

[103] M. Bolic, P. Djuric, and S. Hong, "Resampling algorithms for particle filters: A computational complexity perspective," *EURASIP Journal on Applied Signal Processing*, vol. 2004, no. 15, pp. 2267–2277, 2004.

[104] A. Athalye, M. Bolic, S. Hong, and P. Djuric, "Architectures and memory schemes for sampling and resampling in particle filters," in *Digital Signal Processing Workshop, 2004 and the 3rd IEEE Signal Processing Education Workshop. 2004 IEEE 11th*, pp. 92 – 96, august 2004.

[105] S. Hong, S.-S. Chin, P. Djuric, and M. Bolic, "Design and

implementation of flexible resampling mechanism for high-speed parallel particle filters," *The Journal of VLSI Signal Processing*, vol. 44, pp. 47–62, 2006.

[106] K. V. Mardia and P. J. Zemroch, "Algorithm as 86: The von mises distribution function," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 24, no. 2, pp. pp. 268–272, 1975.

[107] P. Kumaraswamy, "A generalized probability density function for double-bounded random processes," *Journal of Hydrology*, vol. 46, no. 1-2, pp. 79 – 88, 1980.

[108] S. Jammalamadaka and A. Sengupta, *Topics in Circular Statistics*. River Edge, New Jersey: World Scientific Publishing Company, 2001.

[109] A. Gupta and S. Nadaraja, *Handbook of Beta Distribution and Its Applications*. New York, New York: Marcel Dekker, 2004.