Contents

List of Figures

List of Tables

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 movement of neuroprosthetics [4]. Regardless of the application, most BMIs are supported by adaptive filtering algorithms that use the individual firing rates of neurons in the ensemble to estimate the driving biological signal. Whenever there is a requirement to process 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, radar and biomedical engineering [5].

Linear estimation techniques such as the Weiner filter [6], the Kalman filter [7, 8], the least mean squares (LMS) algorithm [9] and point process adaptive filters [10, 11] have been employed to achieve pseudo electro myographic signal reconstruction. Nonlinear approaches to the estimation problem for BMIs include the extended Kalman filter [12], the unscented Kalman filter [13] and particle filters [14, 15].

Research has 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. However, due to the computational complexity of implementing particle filters, they are not employed in current BMI technology. This work presents an application of the Bayesian auxiliary particle filter (BAPF) for estimating neural driving signals. It is demonstrated that the BAPF is capable of superior parameter estimation than other filtering techniques in terms of the error between the true and estimated signal.

A parallel hardware architecture of the BAPF is described that significantly increases throughput over serial processing. The target synthesis platform is a field programmable gate array (FPGA). Even with clock rates slower than conventional processors, FPGA can yield substantially higher throughput when configured to process signals in parallel. The latency of the hardware depends solely on the number of neurons in the observed ensemble and is independent of the number of particles.

2 Background

This chapter presents many of the decoding algorithms that have been applied to brain machine interfaces (BMIs). The advantages and limitations of each algorithm are addressed. Signal processing hardware platforms are also discussed and how reconfigurable logic can improve throughput performance.

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 $\mathbf{u}$(t) where

$$\mathbf{R} = \mathbf{E}[\mathbf{u}(t)\mathbf{u}(t)^T] \tag{1}$$

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

$$\mathbf{p} = \mathbf{E}[\mathbf{u}(t)\,d(t)] \tag{2}$$

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

$$\mathbf{w} = \mathbf{R}^{-1}\mathbf{p} \tag{3}$$

The output $y(\mathrm{t})$ 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 illustrated in Figure 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.

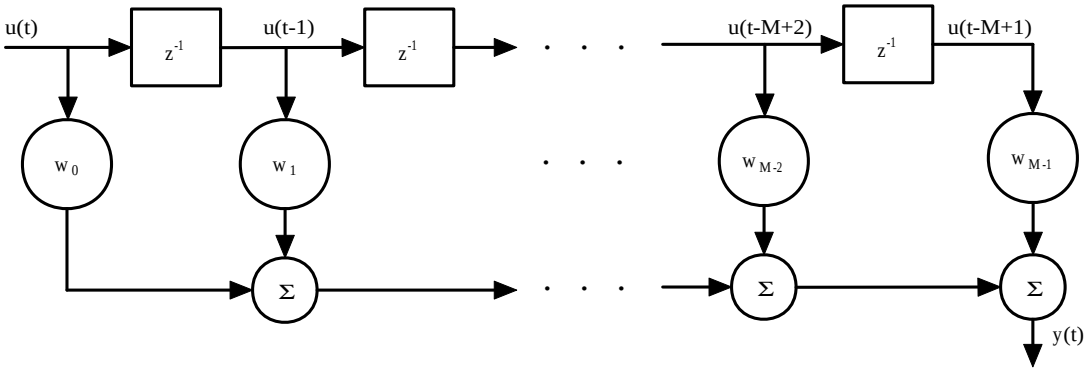

Figure 1: Block diagram of linear traversal filter

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 the linear filter [16]. Recordings from the primary and dorsal premotor corticies have been used as inputs to a WF to recreate center-out reaching tasks [17]. The filter was also applied in [18] to reconstruct 2D cursor movements using electrode arrays implanted in paralyzed human participants.

When estimating the signal, it is often advantageous to observe a large number of neurons, which are believed to be correlated with the signal [19, 20]. 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 [21].

2.2 Least Mean Square

An alternative to the Weiner-Hopf equations is to use stochastic gradient-based algorithms for finding $\mathbf{w}$. These algorithms converge to the optimum Weiner solution based on incoming data. The least-mean-square (LMS) algorithm developed by Widrow and Hoff [22] 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

$$y(t) = \hat{\mathbf{w}}(t)^T \mathbf{u}(t) \tag{4}$$

2. Compute the estimation error

$$e(t) = d(t) - y(t) \tag{5}$$

3. Update the tap weights

$$\hat{\mathbf{w}}(t+1) = \hat{\mathbf{w}}(t) + \epsilon u(t) e(t) \tag{6}$$

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 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 was able to make improvements in that there is no assumption of a stationary environment [23]. 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 7 as

$$\mathbf{x}(t+1) = \Phi(t)\mathbf{x}(t) + \mathbf{h}(t) \tag{7}$$

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 in Equation 8 to be

$$\mathbf{z}(t) = \mathbf{C}(t)\mathbf{x}(t) + \mathbf{v}(t) \tag{8}$$

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

$$\mathbf{P}(t) = E[\mathbf{e}(t)\mathbf{e}(t)^T] = E[(\mathbf{x}(t) - \hat{\mathbf{x}}(t))(\mathbf{x}(t) - \hat{\mathbf{x}}(t))^T] \tag{9}$$

In order to use $\mathbf{z}(t)$ to achieve the best possible $\hat{\mathbf{x}}(t)$, Equation 10 is used [24, 25, 26]

$$\hat{\mathbf{x}}(t) = \hat{\mathbf{x}}(t-1) + \mathbf{K}(t)(\mathbf{z}(t) - \mathbf{C}(t)\hat{\mathbf{x}}(t-1)) \tag{10}$$

The Kalman gain $\mathbf{K}$(t) minimizes the MSE and was found in [23] to be

$$\mathbf{K}(t) = \mathbf{P}(t)\mathbf{H}(t)^T(\mathbf{H}(t)\mathbf{P}(t)\mathbf{H}(t)^T + \mathbf{Q}_v(t))^{-1} \tag{11}$$

The Kalman filtering process is illustrated by the block diagram of Figure 2

It has been shown that it is feasible to acquire 2D cursor control by applying the KF to recordings from the motor cortex [27, 28, 29]. Additional support for using the KF in cursor control can be found in [30], where the

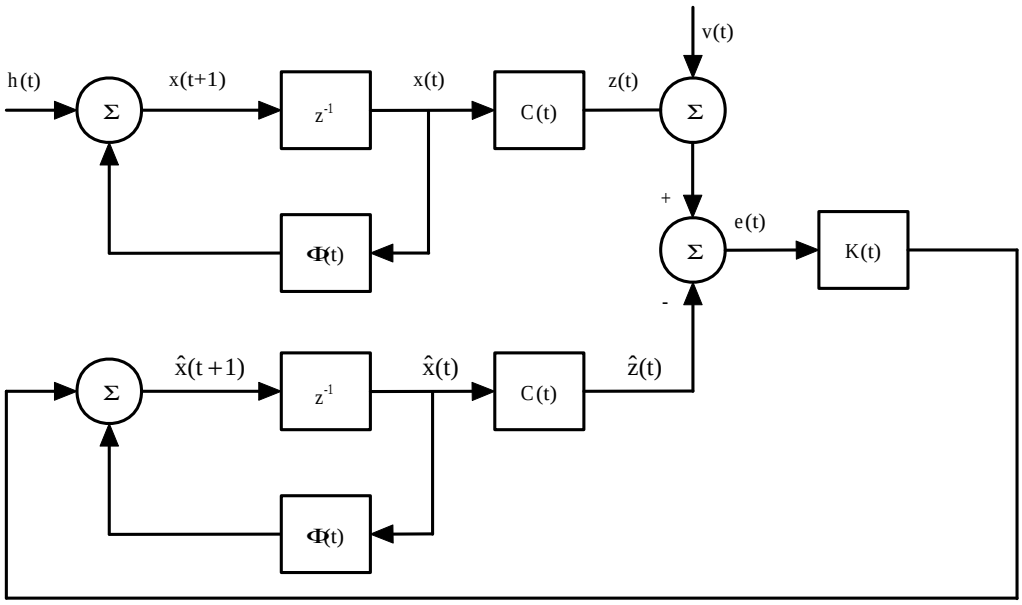

Figure 2: Signal flow graph of Kalman Filtering

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 [31].

By incorporating some previous history of the state estimates, the Nth order KF offers improvement over the standard KF in performing reaching tasks [32]. The switching Kalman filter (SKF) proposed in [33] uses multiple KFs that operate in parallel to estimate hand kinematics with less MSE than the KF.

The extended Kalman filter (EKF) can be applied to situations where the state transition and measurement matrices are nonlinear. 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 [34]. The first application of the EKF for neural signal processing was seen in [35], where control and tracking of spatiotemporal cortical activity are performed.

The unscented Kalman filter (UKF) is another nonlinear version of the KF. 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 [36]. 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 [13] 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 [37]. 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 $\mathbf{K}(t)$ goes to zero and no adaptation will occur [38]. Invertibility issues also arise when the number unknown parameters needing to be estimated become large [39].

2.4 Neural Plasticity

Despite the 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 seem to leave their mark on the brain by altering 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 [40, 41]. 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 [42].

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 [43, 44]. However, when processing an ensemble of neural signals using the algorithms discussed in the previous sections, there is little or no effort to account for the dynamic properties of the individual neurons.

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 [45]. Each neuron has its own model parameters which comprise the state vector $\mathbf{x}(t)$ for that neuron. The following equation defines

a tuning function for a Gaussian-shaped hippocampal place cell.

$$\lambda_(t) = exp\left\{\alpha_(t) - \frac{(s(t) - \mu_(t)^2)}{2\xi_(t)^2}\right\}, \tag{12}$$

Place cells fire when an animal is in a specific region $s(t)$ and are responsible for spatial mapping [46]. 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 3 shows the tuning function of a place cell modeled by Equation 12 with a state of $\mathbf{x}(t_0) = [\alpha = 3.5\ \mu = 10\ \sigma = 10]^{\mathrm{T}}$.

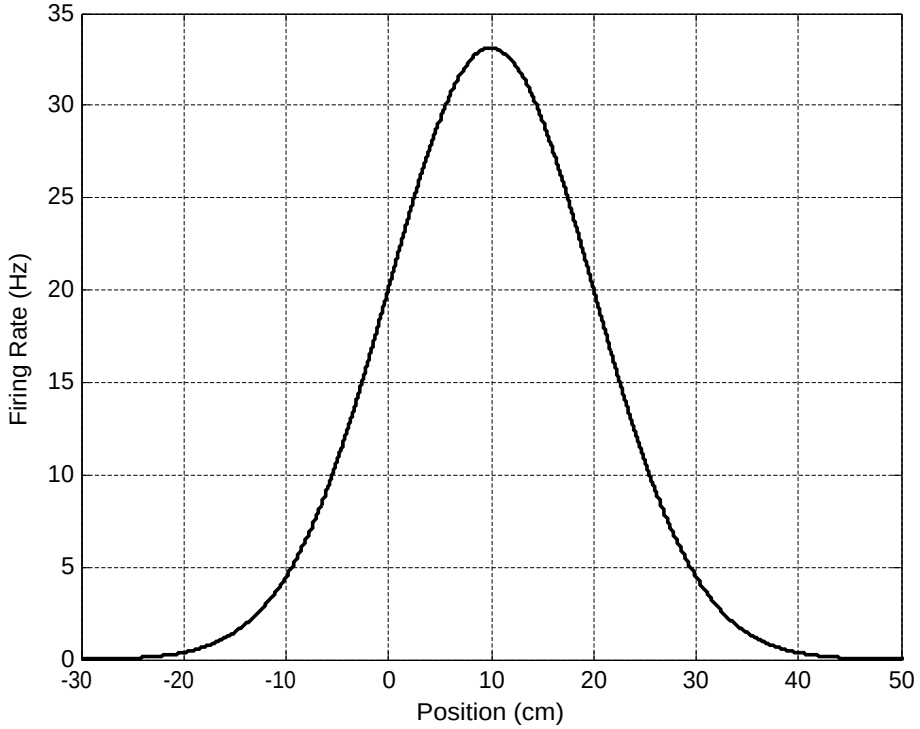

Figure 3: Tuning function for a single hippocampal place cell.

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 4. Here it can be seen how the same neuron responds very differently to the same stimulus over time.

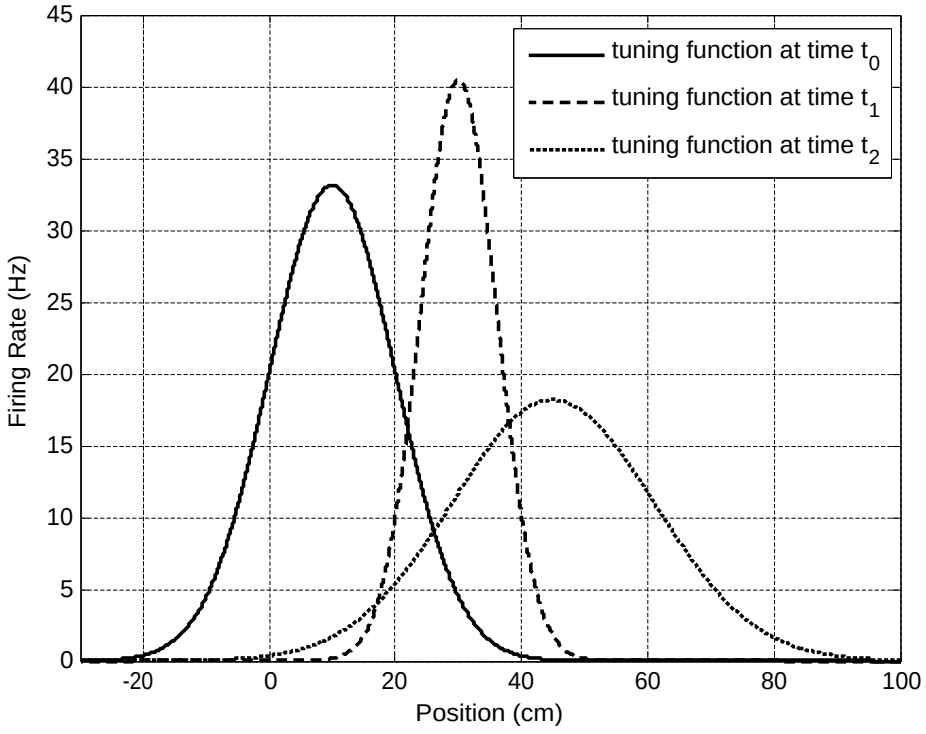

Figure 4: A dynamic tuning function at three different instances of time.

Depending on how much variation the neurons exhibit will dictate 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 of the neuron's tuning function parameters must be approximated as part of

the system state vector. The following signal processing algorithms describe how to track the time-varying parameters of a plastic neuron.

2.5 Steepest Descent Point Process Adaptive Filtering

The algorithm discussed in this section is based on a steepest descent adaptive filtering methodology. It considers the observed neural spike trains as point processes that have a Poisson arrival rate. Using the instantaneous firing rate of a neuron, it seems to be possible to estimate dynamic tuning function parameters [10].

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 $\lambda(t, s(t), \mathbf{x}(t), \mathbf{H}(t))$, 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 $\mathbf{H}(t)$ denotes the history of the spiking process and the biological signal up to time t [47]. 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 13

$$\lambda(t|s(t), \mathbf{x}(t), H(t)) = \overset{lim}{\Delta t} \to 0 \frac{P(N(t+\Delta t) - N(t) = 1|s(t), \mathbf{x}(t), H(t))}{\Delta t} \tag{13}$$

The probability of a neuron spiking over $[t, t+\Delta t)$ can be assumed to be $\lambda(t|s(t), \mathbf{x}(t), H(t))\Delta t$, which generalizes the inhomogeneous Poisson process [48]. 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 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 [49].

Using the conditional intensity function of Equation 13, an expression for the probability of observing a spike over Δt is

$$P(\Delta N_k) = [\lambda(t_k|\mathbf{x}_k, \mathbf{H}_k)\Delta t]^{\Delta N_k}[1 - \lambda(t_k|\mathbf{x}_k, \mathbf{H}_k)\Delta t]^{1-\Delta N_k} + o(\Delta t_k) \quad (14)$$

From [49], it was shown that for small enough values of Δt the conditional probability reduces to

$$P(\Delta N = 1|\mathbf{x}_k\mathbf{H}_k) = exp(\Delta N_k log(\lambda(t|\mathbf{x}_k\mathbf{H}_k)\Delta t) - \lambda(t_k|\mathbf{x}_k\mathbf{H}_k)\Delta t) \quad (15)$$

Since the probability density of Equation 15 depends on the unknown parameter vector $\mathbf{x}_k$, the log of Equation 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

$$l_k(\mathbf{x}_k) = log(\lambda(t|\mathbf{x}_k, \mathbf{H}_k))\Delta t - \lambda(t|\mathbf{x}_k, \mathbf{H}_k) \quad (16)$$

Equation 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 [10].

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

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_{k-1} - \epsilon \left[\frac{\partial J_x \mathbf{x}}{\partial \mathbf{x}} \right]_{\mathbf{x}=\hat{\mathbf{x}}_{k-1}} \tag{17}$$

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 17 is replaced by the instantaneous likelihood in Equation 16 to yield

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_{k-1} - \epsilon \left[\frac{\partial l_x \mathbf{x}}{\partial \mathbf{x}} \right]_{\mathbf{x}=\hat{\mathbf{x}}_{k-1}} \tag{18}$$

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

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_{k-1} - \epsilon \left[\frac{\partial log\lambda}{\partial \mathbf{x}} (\Delta N_k - \lambda \Delta t_k) \right]_{\mathbf{x}=\hat{\mathbf{x}}_{k-1}} \tag{19}$$

Although this filter does not include the encoded signal as part of the state vector, it does provide a powerful technique to observe the affects of neural plasticity on the system model. It also provides insight as to why the processes in the previous sections would tend to degrade over time.

2.6 Stochastic State Point Process Adaptive Filtering

Based on the work in Section 2.5, the algorithm was extended to also include a learning rate that is proportional to the error and is therefore dynamic. 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 in [11] 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 $\mathbf{x}_{k-1}$: $\mathbf{x}_k$ based on $\mathbf{x}_{k-1}$:

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} \tag{20}$$

2. Update the variance with zero-mean Gaussian noise $\mathbf{Q}$:

$$\mathbf{W}_{k|k-1} = \mathbf{F}\mathbf{W}_{k-1}\mathbf{F}^T + \mathbf{Q} \tag{21}$$

3. Update the posterior variance in terms of the probability of firing λ

defined in Equation 15:

$$(\mathbf{W}_{k|k})^{-1} = (\mathbf{W}_{k|k-1})^{-1} + \left[\left(\frac{\partial log\lambda}{\partial \mathbf{x}_k} \right) [\lambda \Delta t] \left(\frac{\partial log\lambda}{\partial \mathbf{x}_k} \right)^T - (\Delta N_k - \lambda \Delta t) \left(\frac{\partial^2 log\lambda}{\partial \mathbf{x}_k \partial \mathbf{x}_k^T} \right) \right]_{x_{k-1}} \quad (22)$$

4. Update the mean (state):

$$(\mathbf{x}_{k|k}) = (\mathbf{x}_{k|k-1}) + \left[\left(\frac{\partial log\lambda}{\partial \mathbf{x}_k} \right)^T (\Delta N_k - \lambda \Delta t) \right]_{x_{k-1}} \quad (23)$$

Unfortunately, this methodology assumes an approximately linear, Gaussian global environment, which fails to be accurate when a large and diverse set of system parameters are present. 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 [50]. An additional drawback of the 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

[51]. 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, from a proposal density that is chosen to approximate the posterior. 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 [52].

Given the previous state and the current observations, recursive Bayesian estimation uses a two-stage process to solve for the posterior distribution. In the first stage, Bayes' rule is used to update the posterior from the previous step. In the second stage, the current posterior is calculated using this updated posterior. The unknown state vector $\mathbf{x}(t)$ is estimated by a set of P random samples, the particles $\mathbf{x}^r(t)$ with $r = 0, \ldots, P-1$, at each iteration of the algorithm. Each particle has its own associated weight $w^r(t)$. It is assumed that each $\mathbf{x}^r(t)$ is the sum of the previous value, a hyperparameter $\Theta = [\theta, \psi_1, \ldots, \psi_K]$ and an error $e(t)$:

$$\mathbf{x}^r(t) = \mathbf{x}^r(t-1) + \Theta + e(t) \tag{24}$$

The particle values are simulated from a proposal density π that approx-

imates the current posterior distribution of the parameters. 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

$$w^r(t) \propto w^r(t-1)\frac{p(N(t)|\mathbf{x}^r(t))p(\mathbf{x}^r(t)\mathbf{x}^r(t-1),\Theta)}{\pi(\mathbf{x}^r(t)|\mathbf{x}^r(t-1),N(t),\Theta)} \tag{25}$$

where $r = 1,\ldots,P$.

Equation 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. When Poisson firing rates are assumed, the posterior density becomes

$$p(N(t)|x^r(t)) = \frac{(\lambda\Delta t)^{\Delta N(t)}e^{-\lambda\Delta t}}{\Delta N(t)!} \tag{26}$$

The particle weights are then normalized so that they sum to unity. Next, the particles are resampled. This process will retain particle values with higher probabilities and discard ones 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) [51] in an attempt to avoid the effects of degeneracy. Degeneracy occurs when all but a few particle weights are close to zero. This results in most of the particles yielding 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

$$\hat{\mathbf{x}}(t) = \sum_{r=0}^{P-1} w^r(t)\mathbf{x}^r(t) \tag{27}$$

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 [53]. Here a simple exponential model was used to represent the firing rate of a single neuron in a cricket sensory system. This was further improved in [54], 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 [55]. 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 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 cells. Each CLB consists of look-up tables (LUTs), multiplexers, flip-flops, basic logic elements (AND, OR and NOT), block

random access memory (BRAM) and shift registers [56, 57]. The number of CLBs contained within a single FPGA can vary from hundreds to tens of thousands. In addition to the CLBs, current generation devices incorporate dedicated fixed point multipliers for efficient implementation of arithmetic functions.

FPGAs offer advantages over other processing devices such as a DSP or application specific integrated circuit (ASIC). DSPs are essentially highly specialized general purpose processors tailored to the needs of signal processing [58]. 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 adversely affects 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 [59]. 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, synthesis of a design can be implemented instantly and any changes that need to made can be reprogrammed into the design.

FPGAs provide an ideal platform for digital signal processing implementation, combining the reprogrammability, architectural flexibility and

system-level integration of general purpose processors with the performance offered by customizable hardware. Even with clock speeds well below conventional processors, reconfigurable logic can yield substantially superior throughputs when made massively parallel [60]. Therefore, the BAPF algorithm can be implemented in hardware for real-time neural signal processing using an FPGA.

2.8.1 Virtex: Fine-grained Device

The fine-grained architecture of the Xilinx Virtex FPGA uses a general routing matrix to configure the interconnects between the CLBs. The only other resources available on first generation Virtex devices is BRAM and clock synchronization control. A top-level topology of the Virtex family is shown in Figure 5.

Each CLB of Virtex family devices is composed of two slices. Each of these slices consists of two logic cells (LCs). The LCs themselves contain a four-input LUTs, control logic and a flip-flop. A detailed schematic of a slice can be seen in Figure 6.

Although Virtex devices lack dedicated arithmetic units, multiplication and full 1-bit adders can be configured through AND gates and XOR gates. Aside from the on-chip BRAM, each of the LUTs can be used to store data as 16 bits or combined with the other LUT in the LC to store data as 32 bits. Maximum clock rates for these devices reach 200 MHz.

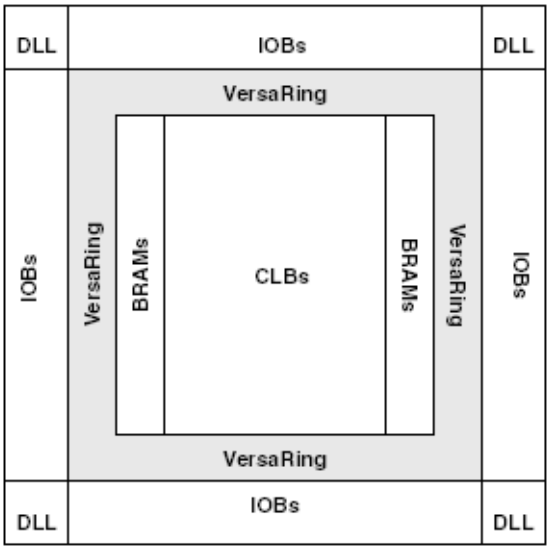

Figure 5: Virtex FPGA architecture

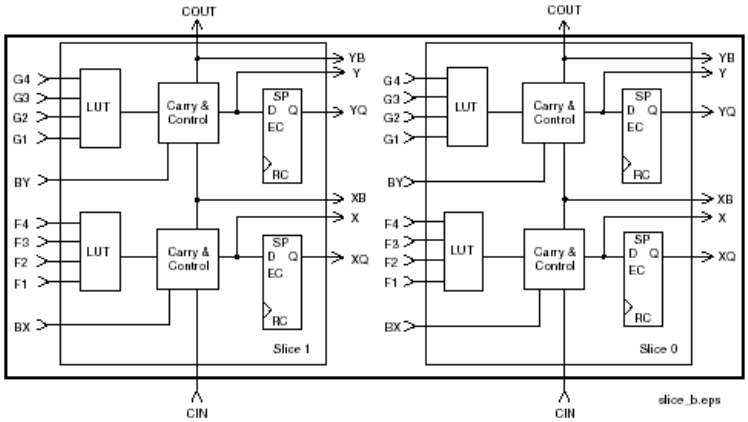

Figure 6: Detailed view of Virtex slice

2.8.2 Virtex-5: Course-grained Device

By interconnecting CLBs to create modules such as adders and multipliers, FPGA performance is reduced [57]. To overcome this, course-grained archi-

tectures incorporate frequently used modules as hard macros. This is seen in the later generation Xilinx Virtex-5 family, where some of the slices contain a DSP. In particular, the Virtex-5 devices have 18-bit x 25-bit embedded multipliers called a DSP48E, whose architecture shown in Figure 7.

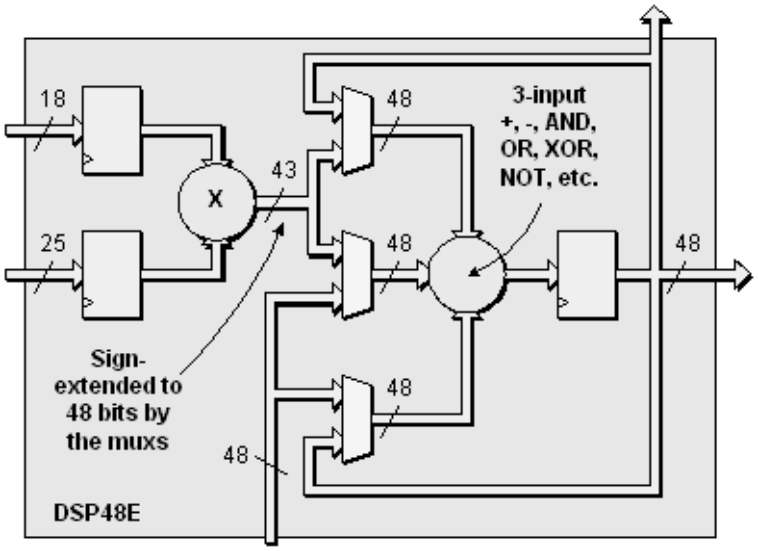

Figure 7: Detailed view of DSP48E

Additional features of Virtex-5 devices include: phase-locked loops to serve as a frequency synthesizer to produce a wide range of 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 8. One practical limitation of the Bayesian auxiliary particle filter (BAPF) is that it is computationally intensive. Ow-

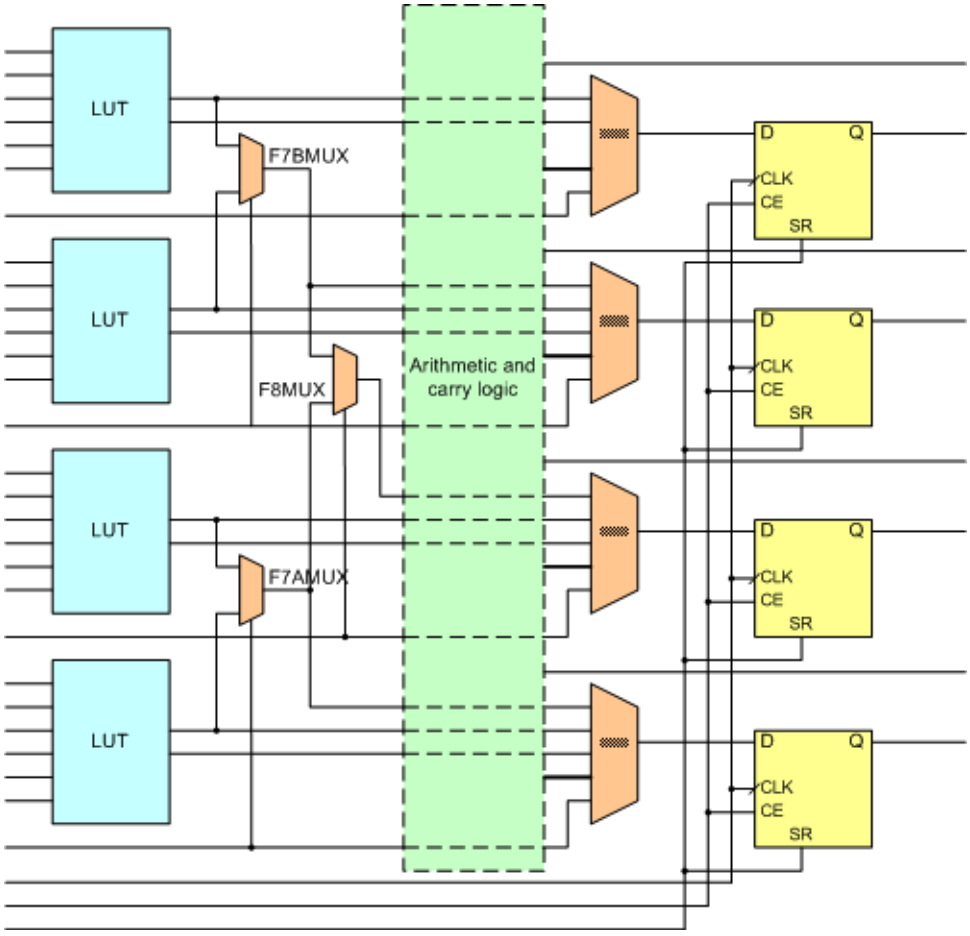

Figure 8: Detailed view of Virtex-5 slice

ing 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 general purpose sequential processing unit. For example, the simulations presented in Section 3.5 tracked up to 100 parameters for 100 neurons, each modeled with 100 particles. It is for this reason that particle filters are not used for real-time BMI systems [13].

3 Preliminary Work

A major deficiency in particle filtering is the effect that outliers or unexpected observations have on filter performance [61]. Auxiliary particle filters 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 [62] uses a two-stage weight update procedure. The first stage weights are used in the resampling process. The second stage weights are used to compute the state estimate as a weighted sum.

The Bayesian auxiliary particle filter is used to track two parameters in each of K hippocampal place cell neurons whose intensity is described according to Equation 35. 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 and results are quantified using both the mean squared error metric and confidence intervals.

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 noise are carried out. Performance of the algorithm is presented and compared against the linear Wiener filter and the sampling importance resampling (SIR) particle filter. It is shown that the BAPF is a more robust decoding algorithm the Wiener and SIR particle filter in terms of MSE.

One focus of this research is to translate the BAPF into programmable hardware such as one of the field programmable gate array (FPGA) devices discussed in Section 2.8. This chapter also presents parallel processing units that are utilized to facilitate the implementation of the BAPF. By exploiting properties of the algorithm which make it conducive to parallel processing, a highly scalable hardware architecture can be developed for its use in real-time neural signal processing. Synthesis results for each module are reported for resource utilization, throughput and clock rate.

3.1 Bayesian Auxiliary Particle Filtering

Successful application of the BAPF was demonstrated in [50] 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 algorithm described in Section 2.6 and was shown to result in less MSE. This section demonstrates how the BAPF can be extended to include the driving signal as part of the estimated state vector using only firing times of an ensemble of neurons as the observations.

The specific implementation of the BAPF for estimating neural driving signals is described next. Note that $\mathcal{N}(0, \sigma)$ refers to a normal random variable with zero mean and standard deviation σ.

1. Simulate the hyperparameters according to:

$$\Theta^r \sim \mathcal{N}(0, \sigma_1), \tag{28}$$

where σ_1 is some fixed constant.

2. Compute non-noisy particle estimates:

$$\hat{\mathbf{x}}^r(t) = \mathbf{x}^r(t-1) + \Theta^r \tag{29}$$

3. Compute the first stage weights $g^r(t)$ using the previous second stage weights $w^r(t-1)$:

$$g^r(t) \propto w^r(t-1)p(N(t)|\hat{\mathbf{x}}^r(t)) \tag{30}$$

Resample $\hat{\mathbf{x}}^r(t)$ according to $g^r(t)$.

4. Simulate the hyperparameters according to:

$$\Theta^r \sim \mathcal{N}(0, \sigma_2), \tag{31}$$

where σ_2 is some fixed constant.

5. Define $\mathbf{x}^r(t)$ by adding noise to the resampled $\hat{\mathbf{x}}^r(t)$ as:

$$\mathbf{x}^r(t) = \hat{\mathbf{x}}^r(t) + \Theta^r \tag{32}$$

6. Compute second stage weights:

$$w^r(t) \propto \frac{p(N(t)|\mathbf{x}^r(t))}{p(N(t)|\hat{\mathbf{x}}^r(t)} \tag{33}$$

7. Compute an estimate of the state as a weighted sum of $\mathbf{x}^r(t)$:

$$\chi(t) = \sum_{r=0}^{P-1} w^r(t)\mathbf{x}^r(t) \tag{34}$$

3.2 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:

$$\lambda_j(t) = exp\left\{\alpha_j(t) - \frac{(s(t) - \mu_j(t)^2)}{2\xi_j(t)^2}\right\},\ j = 1, \ldots, K \tag{35}$$

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} neurons 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 35 by assuming that the parameters α and ξ are (a) constant over time and (b) the

same for all K neurons. Equation 35 therefore simplifies to:

$$\lambda_j(t) = exp\left\{\alpha - \frac{(s(t) - \mu_j(t)^2)}{2\xi^2}\right\},\ j = 1, \ldots, K \tag{36}$$

Note that these simplifications apply only to the Bayesian auxiliary particle filter and the simulated neurons have unique parameters $\alpha_j(t)$ and $\xi_j(t)$.

The assumption is based on prior work concerned with Bayesian auxiliary particle filter tracking [50]. Specifically, it was noted that if the maximum firing rate for a typical neuron is assumed to be 50 spikes per second, then $\alpha \leq ln(50)$, or equivalently $\alpha \leq 3.9$.

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 result in biologically impossible firing rates.

Similarly, the parameter ξ measuring the receptive field width is modeled as a constant because it is observed that 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 [63]. Since α and ξ are assumed constants, the unknown state parameters are $s(t)$ and $\mu_j(t)$ which must be 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 $\lambda_j \Delta t$:

$$p(N_j(t) = 1) = \lambda_j \Delta t \; exp\left\{-\lambda_j(t)\Delta t\right\} \tag{37}$$

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

3.3 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, meaning that the time variable can be redefined to index bin number instead of 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 $t = 1, \ldots, M$.

3.4 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; the random walk standard deviation is selected at random for each time step from a uniform distribution ranging from 0 to 2.0 cm/time step. Each trial simulates 30 seconds of neural behavior with a sampling period of $\Delta t = 2ms$. The BAPF is compared to the linear Weiner filter and standard sampling importance resampling particle filter described in Section 3.4.3. Simulations are coded in MATLAB.

3.4.1 Neural Signal Simulation

Each simulation began 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 30s 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 and rates of change selected at random over the range -0.5 to $+0.5$ cm/s. 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 $\Delta t = 2$ms, it is not necessary to enforce a minimum refractory period between spikes.

3.4.2 Filter Parameters

As stated in Section 3.2, several assumptions are made in designing 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: $\alpha = 3.5$ and $\xi = 4$. 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 $u(0)$, ± 5 (15cm) error. This assumption is based on 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 [50].

As described in Section 3.3, 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 s characterizing mouse position (see Steps 1 and 4 of Section 3.1) are made an order of magnitude larger than their counterparts, ψ_j and $\mu_j(t)$.

3.4.3 Sampling Importance Resampling Particle Filter

The Sampling Importance Resampling (SIR) particle filter [53, 54] is used as a benchmark for the BAPF neural decoder. SIR filters are sequentially designed 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 designed to estimate the signal at the new stage. SIR filters do not employ hyperparameters or any of the auxiliary features characteristic of BAPF algorithms. Studies indicate that SIR filters frequently fail to be robust or adapt to signal changes [61]. This is a consequence of the way in which their weights are calculated. The results show that BAPF filters consistently out-perform their SIR counterparts.

3.4.4 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 [19]. Wiener filters are 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:

$$\hat{u}(t) = \gamma + \sum_{j=1}^{K} \sum_{\tau=-d}^{0} N_j(t-\tau)\ \rho_{j,\tau} \tag{38}$$

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 ms time bins. The number of lagged bins is $d = 20$, hence the Wiener filter operated over the preceding one second of data. Each 30 second simulated data set is divided evenly 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 poor performance on this particular data set, the Wiener Filter results are only presented as a benchmark reference in Figure 9.

3.4.5 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 [64]. In all three cases, spike times are generated as described in Section 3.4.1.

For trials involving missed spike detections, a fraction of the detected spikes are 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 paired together. Then, for a fraction of the times when a spike occurred in one neuron but not its mate, the spike is transferred from the given neuron to its mate. In all tests of robustness, $K = 50$ neurons are used.

3.5 Software Simulation Results

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 9 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 9 represents an average over 100 simulated data

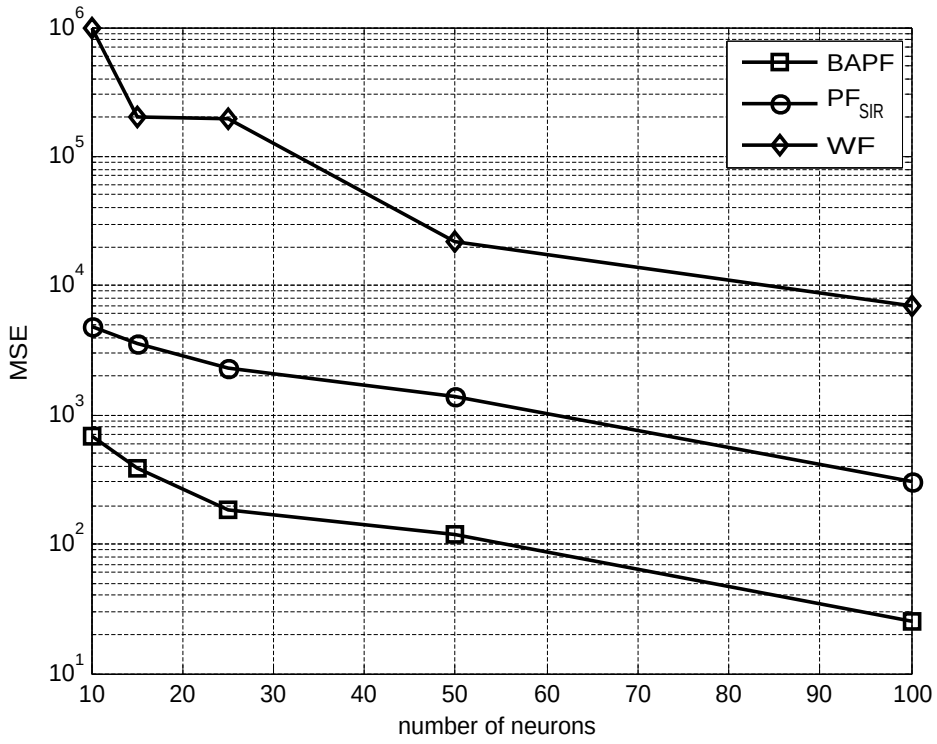

Figure 9: Comparison of MSE vs number of neurons for the SIR particle filter and BAPF

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 3.4.1. By averaging 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 10 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 the BAPF, a 4 second close up in Figure 10 is required to exaggerate the difference on an expanded scale between the BAPF estimate and the true position.

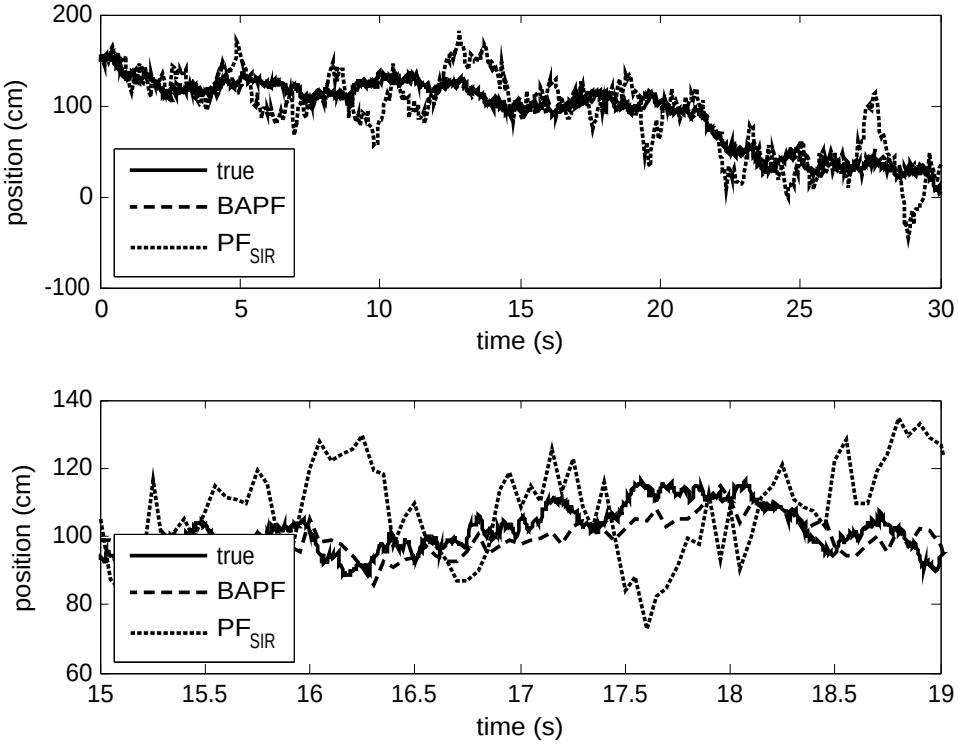

Figure 10: Top:Position estimates for the BAPF and SIR particle filter. Bottom: 4 second window of position estimates.

Figure 11 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).

The confidence interval for the BAPF is substantially narrower than its SIR particle filter counterpart, indicating superior tracking of both mouse position and the dynamic neural firing parameters. In contrast to Figure 9, in which each data point represents the filter performance averaged over 100 data sets, the results in Figure 11 are constructed from 100 simulations of a single data set. Thus, the results in Figure 11 illustrate that, unlike the SIR particle filter, the BAPF is consistently able to reconstruct the mouse trajectory.

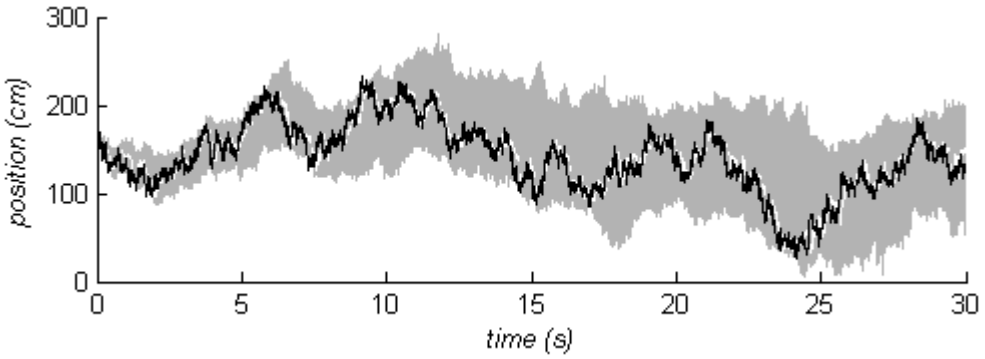

Figure 11: Mouse trajectory reconstruction (expressed using confidence intervals) for a typical random walk using K = 50 neurons. [Black] actual mouse trajectory; [White] BAPF confidence interval; [Gray] SIR particle filter confidence interval. The BAPF confidence interval is nearly indistinguishable from the actual mouse trajectory.

In Figure 12, 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.

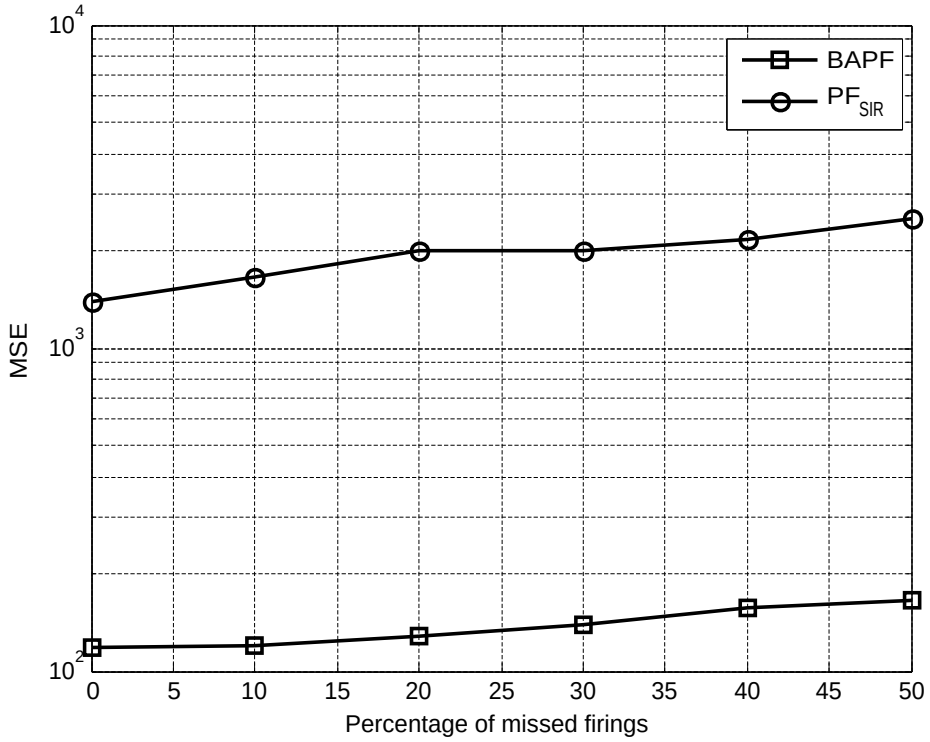

Figure 12: MSE vs % missed detections for a 50 neuron ensemble.

Figure 13 illustrates 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 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.

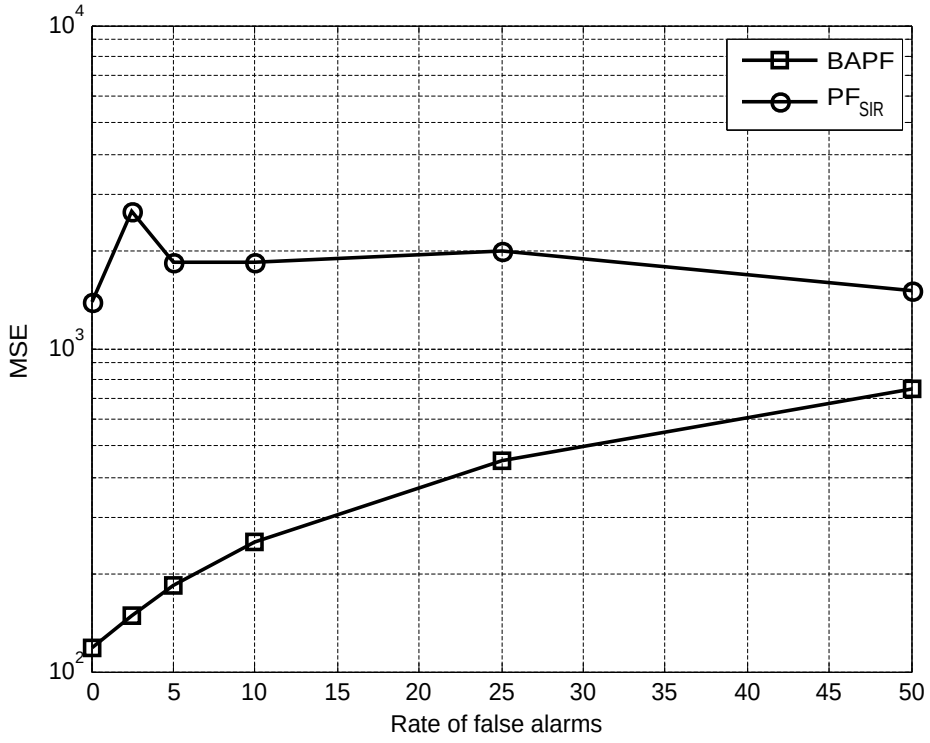

Figure 13: MSE vs false alarm rate (spikes/sec) for a 50 neuron ensemble.

Figure 14 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 experiment, 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%.

There are many existing methods used to decode neural spike trains.

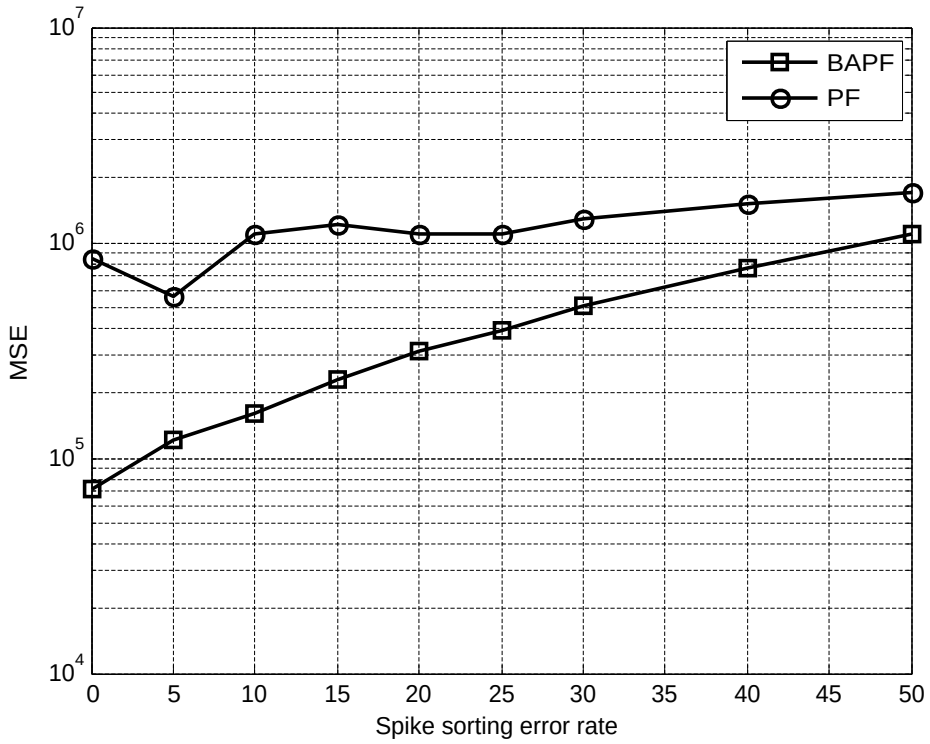

Figure 14: MSE vs % sorting error for a 50 neuron ensemble.

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 assume that all noise is Gaussian [37].

The underlying algorithm of the BAPF addresses many of the limitations

of current neural decoding methods. For example, BAPFs have a built-in 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.

The posterior distribution of the evolving neural parameters given past firing is approximated using particles and their associated weights. Neural parameter estimates are constructed using a compromise between this approximated posterior distribution and the current likelihood of neural firing via the Bayesian paradigm. 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.

In contrast, simple adaptive filters use a single MAP estimate. When this estimate is inaccurate for example, as a consequence of noise, unusual spike activity, or firing dormancy, the system's lack of memory makes it difficult for the filter to correct the estimate in subsequent time steps.

The number of computations required to implement particle filtering algorithms is extremely high compared to some of the estimation techniques mentioned earlier. This is due to the large number of tracked parameters, which are modeled by a large number of particles. For example, the simulations presented here track up to 100 parameters, each modeled with 100 particles. These required 125 seconds to execute 30 seconds of simulated data in the MATLAB environment. Particle filtering algorithms using general

purpose processors for BMI applications is not practical. The serial nature of general purpose processors prevents on-line 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 lend itself for use in real-time application of BMIs. A parallel hardware implementation will also accommodate large neural ensembles as well as an increase in the number of particles used by the filter.

3.6 Parallel BAPF Implementation

Each of the P particles described in Section 3.1 used to estimate the driving signal $s(t)$ are arranged in a parallel fashion as seen in Figure 15. They share the same control logic, which allows all particles to perform the same computations simultaneously. The controller provides enable signals to the datapath based on the sampling rate, the number of neurons and number of particles.

3.6.1 Particle Datapath

A more detailed view of the datapath for a single particle is shown in Figure 16. Here, it can be seen that each particle stores an estimate for $s(t)$ and an estimate of the receptive field centers of all K neurons in the observed ensemble. If the number of particles P is restricted to a power of two, it will significantly reduce the amount of logic required to implement the BAPF.

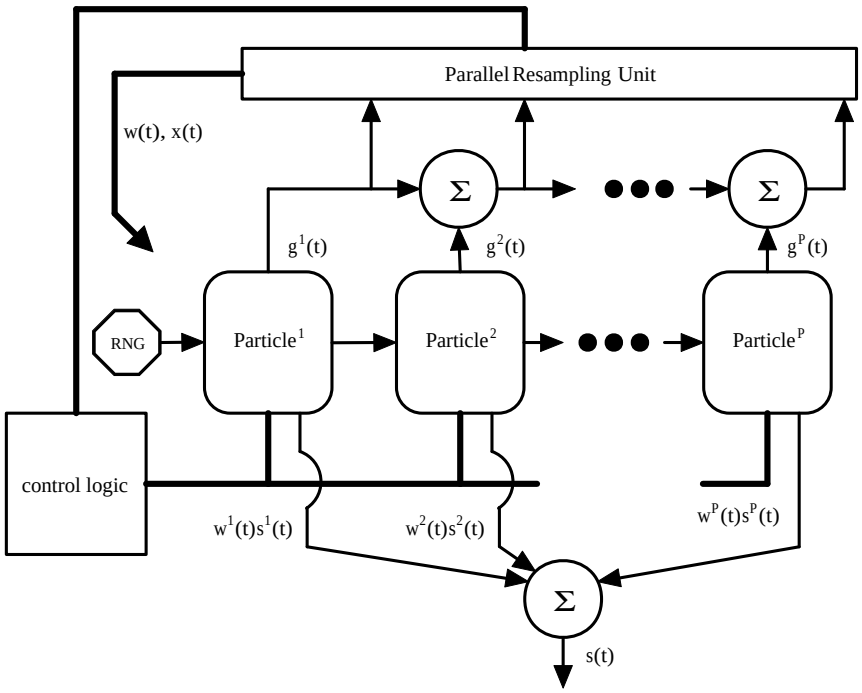

Figure 15: Top level diagram of the Bayesian auxiliary particle filer

The multiplexers assigned to each element of the state vector will pass data according the the controller. Upon reset, all P particles are given the same initial estimate of the state vector by setting the select line shared by all multiplexers to 2. Select is set to 0 to store the results of Equations 29 and 32. Setting select to 1 allows the resampled values of Equation 30 to be stored in the state registers.

The architecture in Figure 17 determines what values get stored for the second stage weights of each particle. The reset signal for the filter acts as the select line for both multiplexers. When reset is asserted, both multiplexers will output data from input 1. In the case of the multiplexer responsible for

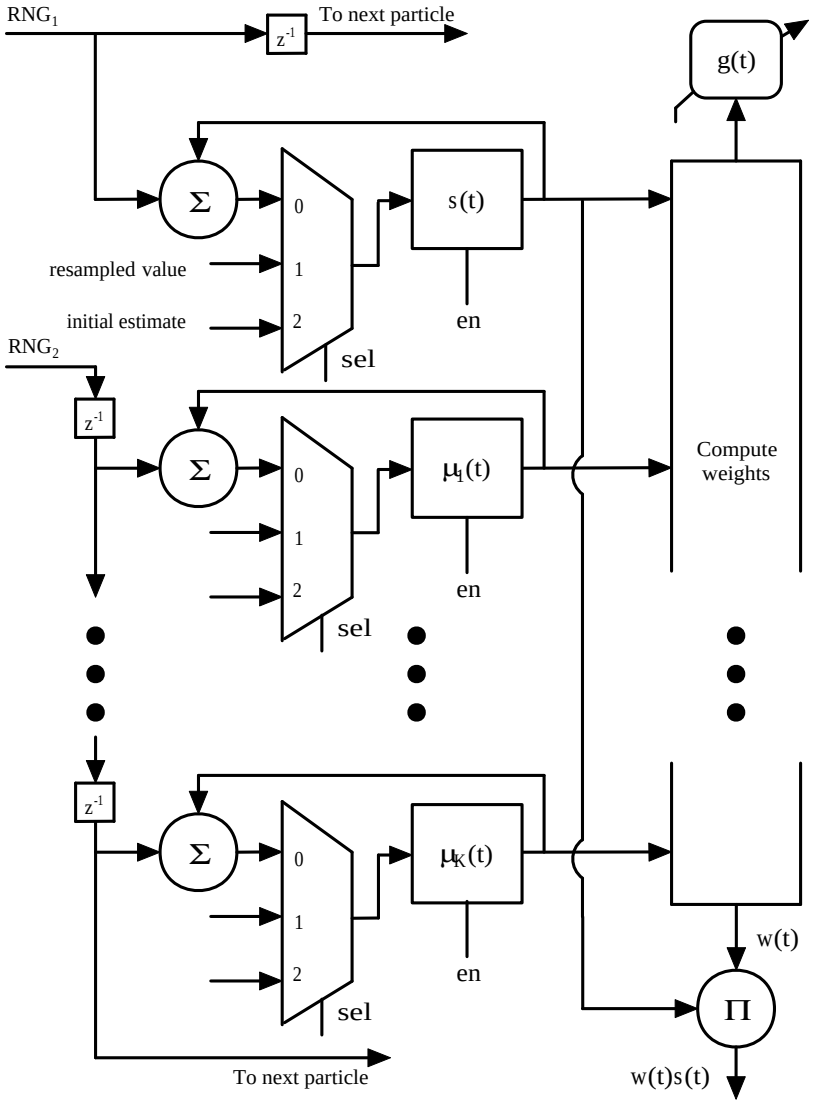

Figure 16: Parallel logic storing the state vector for a single particle

the weight value, the output data is the initial $number\ of\ particles^{-1}$, which is accomplished by shifting a single bit to the right by $log_2(number\ of\ particles)$. For the multiplexer passing the enable signal, the value on input 1 is the ac-

tual reset signal itself, which at this time is equal to 1.

When the reset signal is 0, particle first stage weights defined by Equation 30 are fed back and the enable signal is determined by the controller. Each particle has its own multiplexer for determining weight values. However, the logic for determining the enable signal is common to all particles since all second stage weight values are stored simultaneously. The initialization value of $number\ of\ particles^{-1}$ is also common to all particles.

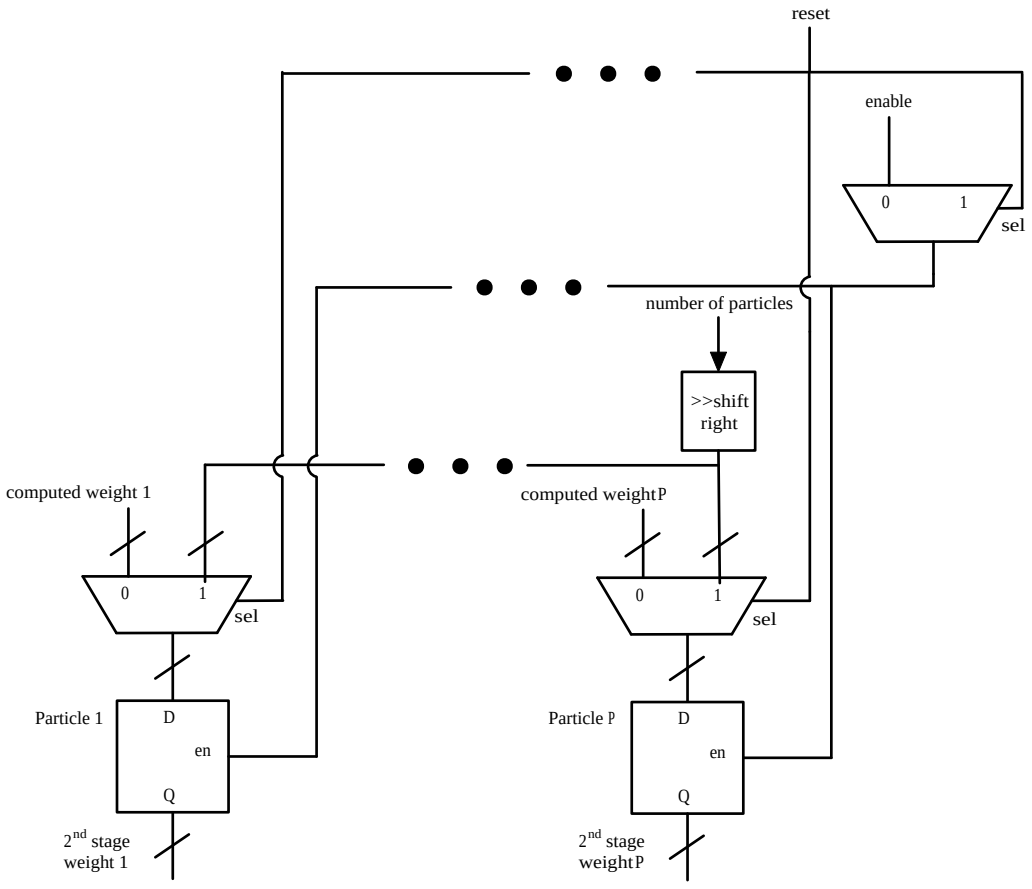

Figure 17: Second stage weights

3.6.2 Random Sampling

According to the filter implementation described in Section 3.1, Equations 28 and 29 define the estimated state to be drawn from a normal distribution. Generation of random variables with distributions such as the normal, Poisson, chi-squared, exponential and many others can be obtained from a uniformly distributed random variable on the interval $(0, 1]$ [65, 66]. A part of this research is concerned with generating standard normal distributions or random variables with zero mean and unit variance. Several techniques for transforming a uniform distribution into a normal distribution have been investigated and their hardware implementations are described. The random number generator in Figure 15 will use either of the methodologies described in Section 3.6.5 or 3.6.6 depending on the required hardware resources.

Storing the random samples along a tapped delay line allows $\hat{\mathbf{x}}^r(t)$ to be formed as the sum of the delayed sample and $\mathbf{x}^r(t-1)$ through a feedback loop as seen in Figure 16. The length of the tapped delay line for $s(t)$ estimates is P. The length of the tapped delay line for $\mu(t)$ estimates is $K\text{x}P$. 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 process is also used to implement Equation 32.

3.6.3 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)[67]. 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. Then on each clock cycle, all bits are shifted one position and the resulting output bit of the feedback logic becomes the input bit to the shift register [68].

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 feedback taps are chosen according to a generator polynomial $g(x)$ which results in a $2^m - 1$ maximal length sequence [69], 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 represents a number between 0 and 1 defined as

$$r = \sum_{n=0}^{m-1} x^n \frac{1}{2^n} \tag{39}$$

Figure 18 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.

The output of a PN generator is assumed to be spectrally white. That is,

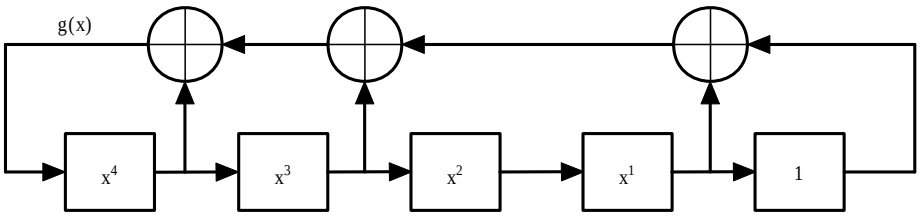

Figure 18: Feedback Circuit of an LFSR

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 [70].

In order to avoid attenuation of the high frequency components and decorrelation of the uniform distribution, skip-ahead logic can be employed [71]. 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

$$\mathbf{x}(t+1) = \mathbf{G}\mathbf{x}(t) \tag{40}$$

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

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{41}$$

Which results in

$$\mathbf{x}(t+1) = \begin{bmatrix} x^3(t+1) = x^3(t) \oplus x^0 t \\ x^2(t+1) = x^3(t) \\ x^1(t+1) = x^2(t) \\ x^0(t+1) = x^1(t) \end{bmatrix} \tag{42}$$

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 41 becomes

$$\mathbf{G}^3 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{43}$$

The resulting state vector is now expressed as

$$\mathbf{x}(t+1) = \begin{bmatrix} x^3(t+1) = x^3(t) \oplus x^2 t \oplus x^1 t \oplus x^0 t \\ x^2(t+1) = x^3(t) \oplus x^1 t \oplus x^0 t \\ x^1(t+1) = x^3(t) \oplus x^0 t \\ x^0(t+1) = x^3(t) \end{bmatrix} \tag{44}$$

An efficient hardware implementation of an LFSR using skip-ahead logic is presented in [72], 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 [73].

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 $k = 13$.

3.6.4 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 [74].

$$x \mapsto F_X(x) = p(X \leq x) \tag{45}$$

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 [75].

$$F_X(x) = \int_{-\infty}^{x} f_X(x)dx \tag{46}$$

The mapping of Equation 45 can be used to produce a value on the real number line from a corresponding probability on the interval (0,1] generated from Equation 39. The pdf of a normal random variable with mean μ and variance σ^2 is defined as

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} exp\left\{\frac{(x-\mu)^2}{2\sigma^2}\right\} \tag{47}$$

Unfortunately, there is no closed form solution to the integral of Equa-

tion 46 for normally distributed random variates [76]. Numerical integration techniques are therefore often used to obtain a solution to this problem. One such solution is to use the error function $erf(x)$ [77]. The error function is defined for positive values of x as

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-u^2} du \tag{48}$$

The CDF of X for standard normals can be determined using Equation 48 as

$$F_X(x) = \frac{1}{2} \left[1 + erf\left(\frac{x}{\sqrt{2}} \right) \right] \tag{49}$$

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

Approximations to Equation 48 can be obtained using a Taylor series expansion, Chebyshev expansion or continued fractions [78]. Exhaustive tables which list the values of $F_X(x)$ for values of x with high resolution are widely available [79]. 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.

3.6.5 Composite Look-up-table

As discussed in Section 3.6.4 approximations to Equation 48 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 [80]. 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 [80].

3.6.6 Box-Muller

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 [81]. The Box-Muller transformation generates two normally distributed numbers $\mathcal{N}_1$ and $\mathcal{N}_2$ from two uniformly distributed numbers U_1 and U_2. The resulting values represent co-ordinates

in a 2D plane within a circle of radius 1. The two uniform distributions are transformed into normal random variables by the Box-Muller method using the following steps:

$$let \;\; \omega = 2\pi U_1 \tag{50}$$

$$let \;\; R = \sqrt{-2ln(U_2)} \tag{51}$$

$$\mathcal{N}_1(0,1) = R\cos\omega \tag{52}$$

$$\mathcal{N}_2(0,1) = R\sin\omega \tag{53}$$

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 [82]. 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:

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \tag{54}$$

Using the identity: $\cos\theta = 1/\sqrt{1+\tan^2\theta}$ and factoring out $\cos\theta$ Equa-

tion 54 can be modified as follows:

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \frac{1}{\sqrt{1+tan^2\theta}} \begin{bmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \tag{55}$$

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

$$K_n R\sin(\theta \pm \phi) = R\sin\theta \pm 2^{-n} R\cos\theta \tag{56}$$

$$K_n R\sin(\theta \pm \phi) = R\cos\theta \mp 2^{-n} R\sin\theta \tag{57}$$

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 [83].

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 [84]. 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 51 through 53 of the Box-

Muller algorithm as illustrated by Figure 19. Starting with two 16-bit LFSRs described in Section 3.6.3, 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 50. 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 51.

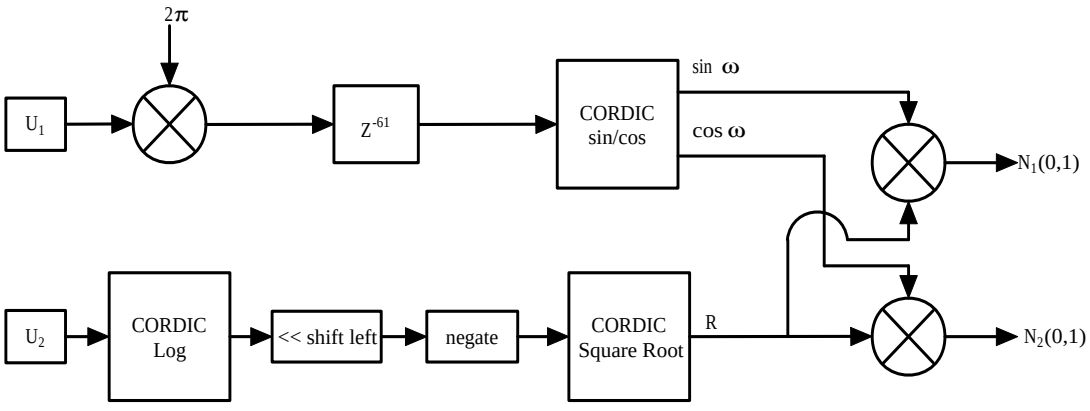

Figure 19: Box-Muller using CORDIC processors

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 reason for the 61 cycle delay in the ω datapath is that computation of $\sin\omega$ and $\cos\omega$ require only 11 clock cycles total and computing R takes 72 clock cycles. Two multiplications can now be executed in parallel to produce the two sequences defined by Equations 52 and 53, 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 is only one clock cycle. The processing architecture of the parallel Box-Muller method implemented with two LFSRs described in section 3.6.3 utilized 2922 of an available 15360 slices on the Xilinx Virtex4 XC4VSX35 with a clock frequency of 100 MHz.

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.

3.6.7 Computing the First Stage Weights

The first stage weight $g^r(t)$ for the r^{th} particle is defined by Equation 30 to be a product of the second stage weight from the previous iteration $w^r(t-1)$ and the likelihood $p(N_j(t)|\mu_j^r(t), s^r(t))$ of each neuron firing during a sample period given the estimated state vector of that particle. Assuming a Poisson firing model the conditional probability for neuron j is computed as a product over a block duration as

$$p(N_j(t)|\mu_j^r(t), s^r(t)) = \prod_{j,t\in B} (\lambda_j \Delta t)^{\Delta N_j(t)} e^{-\lambda_j \Delta t} \tag{58}$$

3.6.8 Computing Exponentials

Computing e^x is performed using two look-up tables (LUT). Let $x = w + v$, where w is the integer part of x and v is the fractional bits of x. Since $e^w e^v = e^{w+v}$, computing e^x can be implemented as the product of two terms. The architecture in Figure 20 is capable of computing e^x for $-12 \leq x < 4$ in increments of 2^{-18}. Since the maximum firing rate for a neuron is $e^\alpha \approx e^{3.9}$, the integer LUT does not need to store values for $x > e^3$.

Two LUTs are used to store values for e^x in read only memory (ROM).

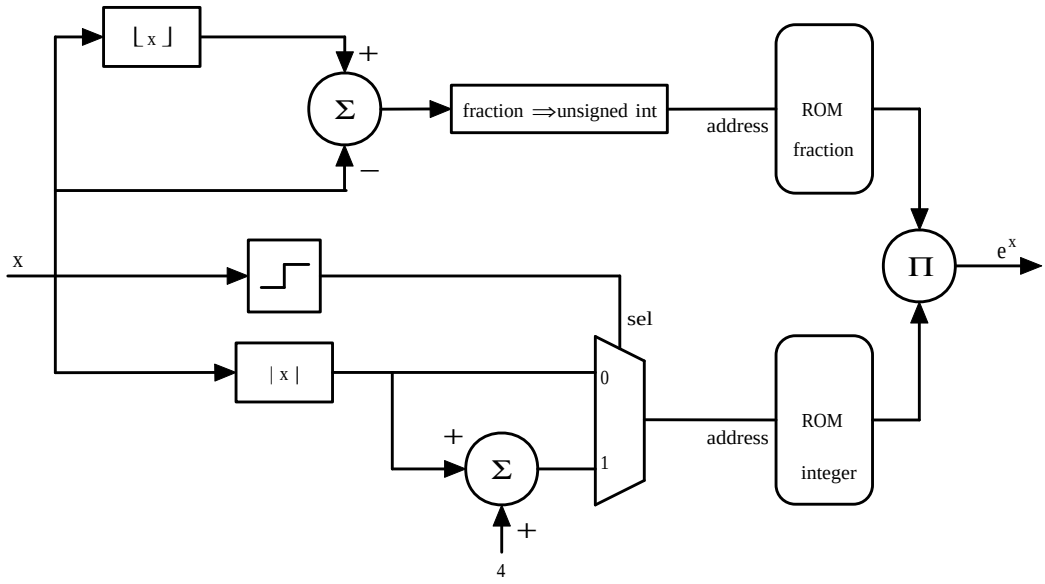

Figure 20: Architecture for computing e^x

The first stores values of e^x for integers between -12 and 3. The second stores values of e^x for $0 \leq x \leq (1 - 2^{-18})$. Memory addresses 0 through 3 for the integer based ROM hold the values of e^0 through e^3. Memory addresses 4 through 12 hold values of e^{-1} through e^{-12}. The sign bit of x is used as the select line of a multiplexer that passes the 4-bit memory address to the ROM.

For positive values of x, the bits representing the integer portion point to the actual memory location to produce the desired e^w. For negative values of x, its absolute value is offset by 5 to produce e^{w-1}. This is done by taking the two's complement of the integer bits and adding 4.

By including an adder, size of the ROM for e^v was reduced from storing values of $e^{\pm 1}$ to storing values of $e^{0 \leq x < 1}$. Assume the desired quantity is $e^{-1.5}$.

Since $e^{-1}e^{-0.5} = e^{-2}e^{0.5}$, only the exponential positive fraction values are required. By subtracting $\lfloor x \rfloor$ from x, the value $|v|$ is obtained. This quantity is then converted to an unsigned integer that points to the appropriate memory address.

The values e^v and e^w are next multiplied to compute e^x. For negative values of x, the result is processed as $e^{w-1}e^{1-|v|}$. This unique processing architecture reduces ROM by a factor of 2 and requires only 5 clock cycles to compute an exponential.

3.6.9 Computing the Likelihood

Initially, the term $\lambda_j = exp\left\{\alpha - \frac{(\mu_j^r(t) - s^r(t))^2}{2\xi^2}\right\}$ must be computed. The quantity $\mu_j^r(t) - s^r(t)$ is the output of the subtracter in Figure 21. The multiplier output is $\left(\mu_j^r(t) - s^r(t)\right)^2$. As discussed in Section 3.4.2, $\alpha = 3.5$ and $\xi = 4$. Therefore, division by by $2\xi^2$ is performed by right shifting 5 bit positions, which is then subtracted from α. This process takes 3 clock cycles and the output is passed to the exponential unit in Figure 20.

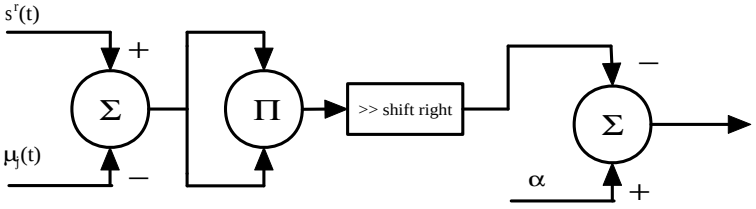

Figure 21: Architecture for computing $\alpha - \frac{(s^r(t) - \mu_j^r(t))^2}{2\xi^2}$

By combining the architectures of Figure 20 and Figure 21 the term λ_j

of Equation 58 can be obtained. As described in Sections 3.3 and 3.4, for a block size of 25 with a 500 Hz sampling rate $\Delta t = 0.05$ seconds. The term $\lambda_j \Delta t$ is then computed using a constant multiplier. By reusing the hardware in Figure 20 with $x = -\lambda_j \Delta t$, the value $e^{-\lambda_j \Delta t}$ is produced.

Raising $\lambda_j \Delta t$ to the ΔN_j power is implemented using the processing architecture of Figure 22. Here, the register is initially set to equal 1. The number of firings for neuron j ΔN_j is multiplied by 4, which is implemented as a left shift of two bit positions. This value is compared to a counter which stops the register from storing the multiplier output. ΔN is initially multiplied by 4 because it takes 4 cycles for the product of the multiplier to be formed. If no firings occurred, $\Delta N = 0$ and the output of Figure 22 is 1.

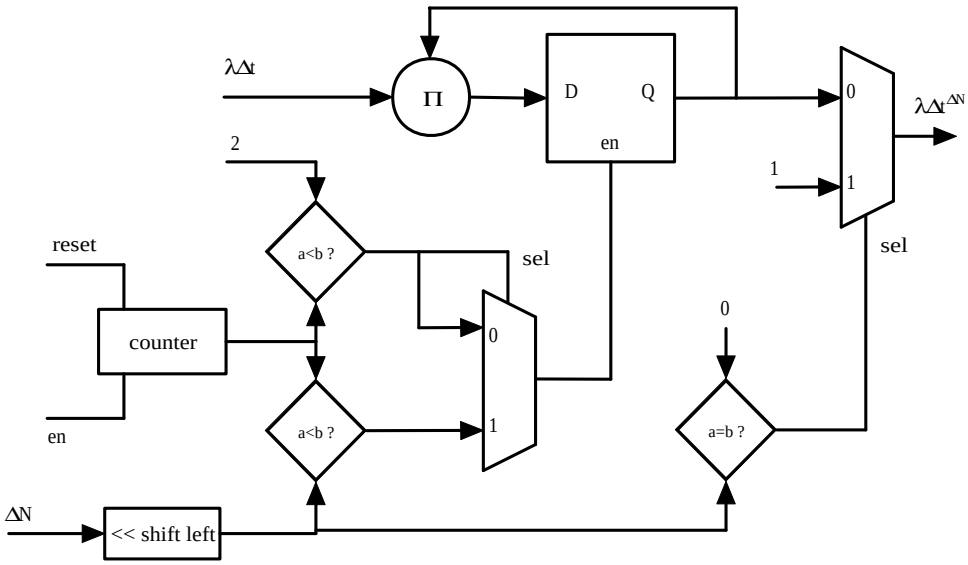

Figure 22: Architecture for computing $\lambda \Delta t^{\Delta N}$

The processing units of Figures 20, 21 and 22 can be configured to com-

pute the conditional probability of Equation 58 for a single neuron over one block period. This process is carried out in parallel for all K neurons for all P particles. The likelihoods for each neuron of particle r are then multiplied together with $w^r(t)$ to implement Equation 30.

3.7 Hardware Synthesis Results

Table 1 presents the required hardware resources for each of the processing modules described in this Chapter. The target device for implementing the modules was a Xilinx XC5VSX50T field programmable gate array. This specific device is capable of running at clock rates of up to 100 MHz. The available hardware resources consists of 288 fixed point multipliers, 34 Mbits of block random access memory and 32640 slices and flip-flops.

Table 1: Hardware resource utilization for particle processing units

	Slices	DSP48Es	clock-cycles	latency
Figure 19	3506	0	1 (after pipelining)	3.7 ns
Figure 20	55	1	5	1.4 ns
Figure 21	12	2	3	3.0 ns
Figure 22	51	4	4/sample	1.6 ns

Each neuron will require at least 118 logic slices and 7 embedded multipliers per particle. To accommodate 50 parallel neural signals estimated using 100 particles would require 590k slices. This would exceed the amount available on the XC5VSX50T. A processor this large would need to be partitioned over multiple devices. However, current generation FPGAs have over

100K slices and 2k embedded multipliers. A smaller processor, one that could support 16 parallel channels estimated with 50 particles would not consume all available resources. Future devices will be likely to provide even more resources to support larger designs.

3.8 IEEE 754 Floating Point Package

The IEEE-754 single precision floating point standard is considered for performing computations for the BAPF. The floating point system provides a variable resolution for the range of numbers being represented. In this format, a 32-bit number is divided into fields as seen in Figure 23 [85]. One bit is the sign bit S, where 0 makes the value positive and a 1 designates it as negative, eight bits are assigned as the exponent and the remaining twenty three bits are the mantissa. The exponent E is coded in a biased form as E-127. Using this scheme a value n is represented as:

$$n = (-1)^S 2^{E-127} M \tag{59}$$

Modules for performing floating calculations according the IEEE-754 stan-

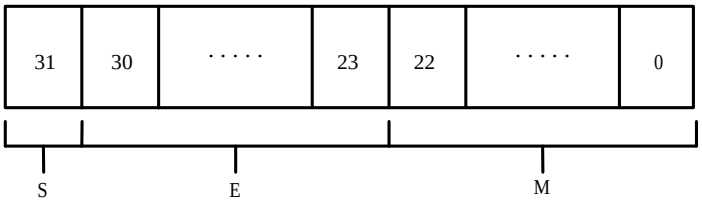

Figure 23: IEEE 32-bit floating point format representation

dard for multiplication/division and addition/subtraction have been designed using the Verilog HDL. Modules for converting between 18-bit fixed point and 32-bit floating point numbers have also been designed using the Verilog language.

4 Future Work

This chapter discusses work yet to be completed. Described here are the remaining processing elements that need to be synthesized. The filter estimates computed using the hardware architecture are compared the estimates obtained in the MATLAB floating point environment. A proposed time line is also presented for completing the research.

4.1 Resampling

The resampling process of particle filters is the computing bottle neck of the algorithm. Here, particles with high weights are retained while particles with low weights are discarded and replaced with more likely state estimates. Performing this process sequentially can become problematic in terms of processing time when the number of particles is large. A parallel resampling process would be more appealing for real-time applications of particle filtering.

By providing each particle with knowledge of all other particle weight and state values, every particle can be resampled simultaneously in two clock cycles. The current particle weights and state estimates are stored in temporary registers. This requires one clock cycle. Then through the use of comparators, multiplexers and uniform random numbers, all particles are resampled on the next clock cycle.

4.2 Computing the Second Stage Weights

Equation 32 defines $w^r(t)$ as the ratio of the likelihoods of the initial estimate and the resampled noisy estimate. By reusing the hardware described in Figures 20, 21 and 22 and following the process described for computing the likelihoods, the ratio can be obtained using a CORDIC divider.

4.3 Signal estimate as a weighted sum

According to Equation 33, the filter output is computed as a weighted sum of the second stage weights and the particle estimates. Multiplying $w^r(t)$ and $s^r(t)$ can be performed as seen in Figure 16 and summed in parallel as in Figure 15. This process could be implemented with P multipliers and $P-1$ adders which could execute over 3 clock cycles. It could also be implemented with a single MAC to execute over $P+3$ clock cycles.

4.4 Automated Controller

Based on the sampling period, block size, number of neurons and number of particles, a state machine will be designed to provide control signals to the datapath. These signals will include select lines for multiplexers, enable lines for date registers, as well as reset signals for multipliers and counters. The controller can be developed through the use of counter, comparators and multiplexers.

4.5 Verification

Once the controller and datapath have been designed, the filter output will be compared with the simulated MATLAB results to verify that the hardware architecture is operating correctly. It is expected that there will be a difference between the two due to quantization error. Further evaluation of the architecture will be performed in order to determine the number of bits needed to provide acceptable precision of the estimated driving signal. Additionally, verification will done to evaluate the performance of the filter as the number of particles increases and decreases. Additionally, real animal data will be used as the input to the filter to observe how

4.6 Throughput Comparison

Once it has been determined that the BAPF is functional, the parallel processing architecture will be compared to a sequential implementation using overall latency as the metric. This will show the benefits of a parallel scheme and provide support for the use of the BAPF for real-time applications. Comparisons will be made for throughput as a function of particles as well as a number of neurons.

4.7 Proposed Timeline

The synthesis of all remaining processing elements is to be completed by June 1, 2010. It is expected that the controller will be designed and synthesized

by July 15, 2010. Verification is anticipated to be finished before October 1, 2010. Research is to be completed and defended by the end of the 2010 Fall semester.

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 non-linear 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] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[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] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

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

[23] 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.

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

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

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

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

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

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

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

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

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

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

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

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

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

[48] 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.

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

[50] 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.

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

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

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

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

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

[60] R. Esposito, J. Mountney, L. Bai, and D. , "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.

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

[62] 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.

[63] 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.

[64] 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.

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

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

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

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

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

[70] 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.

[71] 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.

[72] 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.

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

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

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

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

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

[78] 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.

[79] 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.

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

[81] 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.

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

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

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

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