SYDE 556 Final Project

Visually Guided and Predicted Smooth Pursuit Dynamics using the Neural Engineering Framework

A Report Submitted in Partial Fulfillment of the Requirements for SYDE 556.

By: Yazan Obeidi, 4B

Faculty of Engineering
Department of Systems Design Engineering

April 19 2018.
Professor: Terrence Stewart

Table of Contents

Abstract

In this paper a biologically plausible implementation of a unified sensory and predictive smooth eye pursuit model is proposed using Nengo. The relevant dynamics are expressed in terms of the Neural Engineering Framework as a network of transformations between neuronal populations. The resulting neural network is compared with the MATLAB implementation of the same model, as well as with experimental animal data. The Nengo implementation performs well against constant target velocity inputs; in the cases of sinusoidally varying target velocity, or an accelerating target, the model performs poorly, resulting in high frequency oscillations and sudden rapid divergences. Further research is recommended to establish the nature of these unwanted dynamic effects, and to explore whether changes may be made to the proposed model to greater exploit NEF dynamics such as intrinsic neuronal noise.

1.0 Introduction

Visual animals use smooth eye pursuit to track moving objects across a stationary background. The purpose of this is to keep the image of the object near the fovea [1]. These eye movements may also be accompanied by related head movements to increase visual range. Smooth eye pursuit must compensate for saccadic eye movements, visual noise, sensory noise, neuronal noise, and other sources of noise within and between the visual and motor systems [2, 3]. Image stability is achieved in the brain by integrating stimulus information over time across large populations of neurons, improving the reliability of sensory information [3]. Smooth eye pursuit has successfully been modeled as a closed-loop system which achieve stylization by continuously transforming deviations from ideal eye trajectory into compensatory eye movements [4, 5]. More recent models have attempted to combine multimodal sensory and predictive information [6], as well as using Kalman filtering to actively compensate for noise [3, 7]. These models generally consist of a single input, stimulus target velocity, a signal which would be generated from lower level features in the primary visual system, and a single output that is the resultant eye velocity. These filtered and multimodal smooth pursuit models demonstrate high degree of robustness against noise while still offering comparable performance to human ability [3].

A recent state-of-the-art model was proposed with an architecture consisting of two Kalman filters: one for processing visual information, and another for maintaining dynamic internal memory of target motion which includes a predictive component [3]. The original paper implements it using MATLAB/Simulink as a feedback control system, and claims it demonstrates a general framework on how the brain may combine sensory information with memory to produce motor commands [3].

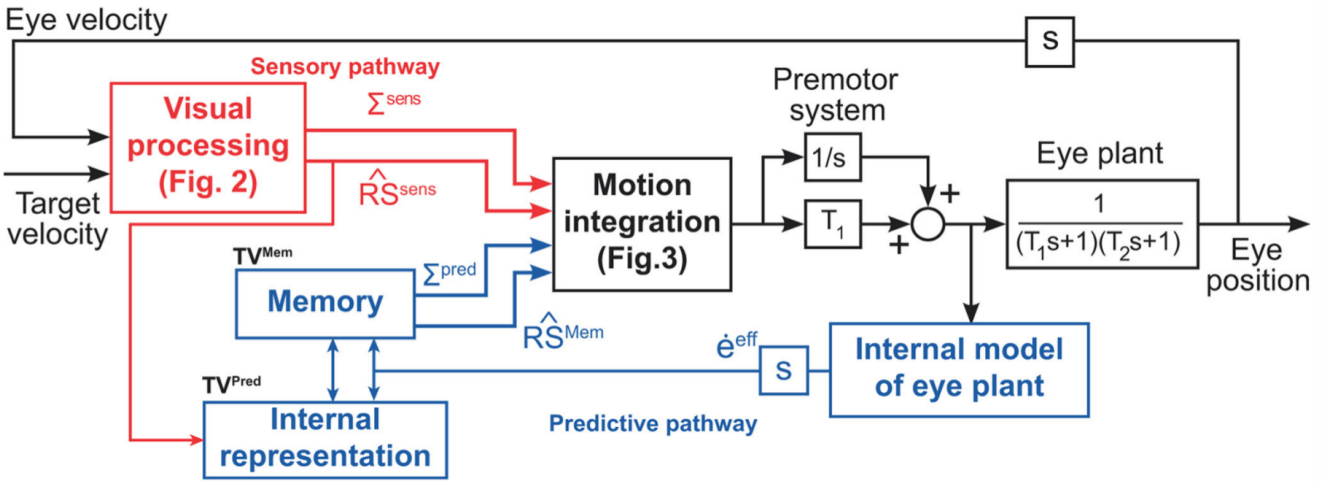

Figure 1: Model architecture. Sensory pathway (red): estimated retinal slip and confidence is generated from target velocity and eye velocity. Predictive pathway (blue): an estimated eye velocity is used to generate a predicted target motion for the next trial; this is kept in memory. The motion integration step generates the motor signal combining both sensory and predictive pathways. This is sent to the premotor system and then to the eye. [3]

The model takes in a single input, target velocity, and produces a single output, eye velocity. The model creates an estimate of retinal slip (RS) using both the sensory and predictive pathways. Kalman filtering effectively combines previous RS estimates with noisy sensory inputs, in the case of the sensory pathway, and noisy internal target velocity representations, in the case of the predictive pathway. Bayesian integration is used to combine the two pathways in a statistically optimal manner. The combined signal is sent to a premotor system that generates the motor signal. The motor signal controls the eye position which results in some eye velocity. Further details are provided in the next section.

In this paper, the same unified sensory and predictive smooth pursuit model is implemented using the Neural Engineering Framework (NEF) in Nengo [8]. This will allow results to be compared a biologically plausible implementation of the unified smooth pursuit model to both the MATLAB implementation and clinical data smooth eye pursuit data.

2.0 Materials and Methods

2.1 System Description

As described in the previous section, the goal is to build an NEF implementation of the unified smooth eye pursuit model. The network will consist of a single input node: target velocity. The idea behind this signal is it comes from another area of the brain, outside the scope of the smooth pursuit model. The output of the model is eye velocity, which is integrated from eye position obtained from sending the premotor command to the eye. The significance of the model is the motion integration component, shown in Figure 3, which combines the sensory and predictive pathways in a statistically optimally manner to be sent to the premotor system. Both pathways are Kalman filtered using certain noise assumptions, namely that the different noises are Gaussian and uncorrelated. They are combined in a weighted function of their estimated variability.

2.2 Design Specification

2.2.1 Sensory Pathway

The sensory pathway process is summarized in the figure below:

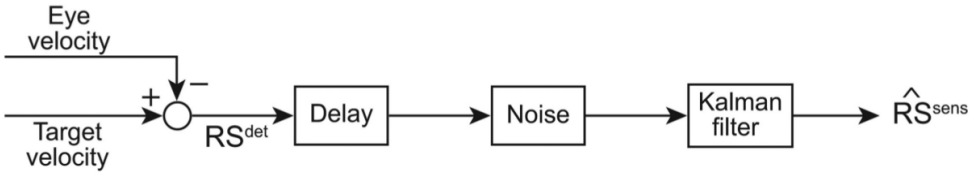

Figure 2: Visual processing pathway; additive and multiplicative noises are added to RS and then Kalman filtered to estimate the sensory RS. [3]

Noisy sensory input is obtained using additive and multiplicative noise on the true value of retinal slip (Rs^{det}), shown in Eq 1. These noise components allowed for baseline and signal-dependent noise.

$$\mathrm{RS}_k^{\mathrm{Noisy}} = \mathrm{RS}_k^{\mathrm{det}} + \delta_{\mathrm{mult}}\mathrm{RS}_k^{\mathrm{det}} + \delta_{\mathrm{add}}, \tag{1}$$

where $\delta_{\mathrm{add}} \sim N(0, \sigma^2_{\mathrm{sens,add}})$, and $\delta_{\mathrm{mult}} \sim N(0, \sigma^2_{\mathrm{sens,mult}})$, with $\sigma_{\mathrm{sens,add}} = 10$, $\sigma_{\mathrm{sens,mult}} = 1.5$.

The evolution of the sensory RS is described as a random walk with some process noise $\theta_k \sim N(0, \mathrm{Q}^2)$ representing the state variability. In addition, the brain only receives a corrupted copy of the sensory RS, also containing additive and multiplicative measurement noises, shown in Eq. 2.

$$\mathrm{RS}_k^{\mathrm{Obs}} = \mathrm{RS}_k^{\mathrm{Sens}} + \gamma_k\mathrm{RS}_k^{\mathrm{Sens}} + \nu_k, \tag{2}$$

Here, $\gamma_k \sim N(0, D^2)$ and $\nu_k \sim N(0, R^2)$. These two components represent the generative model, with noise characteristics Q, D, and R that are necessary for the implementation of the Kalman filter. These are set as $R = \sigma_{\mathrm{sens,add}}$ and $D = \sigma_{\mathrm{sens,mult}}$, and $Q = 1$.

Assuming the brain knows the approximate statistics of the noise that perturbs the retinal signal, the sensory RS may be Kalman filtered using the noisy RS input:

$$\widehat{\mathrm{RS}}_{k+1}^{\mathrm{Sens}} = \widehat{\mathrm{RS}}_k^{\mathrm{Sens}} + K_k(\mathrm{RS}_k^{\mathrm{Noisy}} - \widehat{\mathrm{RS}}_k^{\mathrm{Sens}}) + \eta_k, \tag{3}$$

with $\eta_t \sim N(0, \Omega_n^2)$ as the internal noise of the estimation, and K_k as the Kalman gain:

$$K_k = \Sigma_k^{\mathrm{Sens}}(\Sigma_k^{\mathrm{Sens}} + R^2 + D^2(\Sigma_k^{\mathrm{Sens}} + \widehat{\mathrm{RS}}_k^{\mathrm{Sens}}\,\mathrm{RS}_k^{\mathrm{Sens}T})\mathrm{D}^{2^T})^{-1}, \tag{4}$$

and finally, the estimated error variance used to assess the uncertainty associated with the estimated RS is computed as follows:

$$\Sigma_{k+1}^{\mathrm{Sens}} = Q^2 + \Omega_n^2 + (1 - K_k)\Sigma_k^{\mathrm{Sens}}, \tag{5}$$

Here, the Kalman estimation noise is set as $\Omega_n = 0.3$.

2.2.2 Predictive Pathway

The output of the predictive pathway may be interpreted as the measured RS that is expected 150 ms later if the eye velocity did not change. As an estimated 80 ms is required for visual feedback delays, this provides the predictive pathway with an advance of 70 ms. This is provided in Eq. 6:

$$\widehat{\mathrm{RS}}_k^{\mathrm{Mem}} = \widehat{\mathrm{TV}}_k^{\mathrm{Mem}} - \dot{e}_k^{\mathrm{eff}}, \tag{6}$$

Here, the estimated RS in memory is computed by the difference between the estimated target velocity in memory and current eye velocity. In the original paper, current eye velocity is estimated through an internal model of the eye plant, however for the purposes of simplification and a lack of detail in the paper this has been omitted. Therefore the current eye velocity itself is used.

Similarly as in the sensory pathway, the model assumes the predictive pathway receives a noisy target velocity with additive and multiplicative noise:

$$\mathrm{TV}_k^{\mathrm{Noisy}} = \mathrm{TV}_k + \varepsilon_{\mathrm{mult}}\mathrm{TV}_k + \varepsilon_{\mathrm{add}}, \tag{7}$$

where TV is the actual target velocity and $\varepsilon_{\mathrm{add}} \sim N(0, \sigma^2_{\mathrm{pred,add}})$, and $\varepsilon_{\mathrm{mult}} \sim N(0, \sigma^2_{\mathrm{pred,mult}})$. The original paper approximates TV by summing the estimated sensory RS and the current eye velocity estimation. Here, $\sigma_{\mathrm{pred,add}} = 5$ and $\sigma_{\mathrm{pred,mult}} = 0.75$.

A generative model is proposed regarding the predicted target velocity:

$$\mathrm{TV}_{k+1}^{\mathrm{Pred}} = \mathrm{TV}_k^{\mathrm{Pred}} + B_{\mathrm{int}}\, u_k + \alpha_k, \text{ where } u_k = \Delta(\widehat{\mathrm{TV}}_k^{\mathrm{Mem}}), \tag{8}$$

This model represents the current knowledge of the target velocity including the change in velocity that is computed from a prediction of target motion, and with $\alpha_k \sim N(0, Q^2_{\mathrm{pred}})$ as the process noise and with $B_{\mathrm{int}} = 1$ and $Q_{pred} = 1$.

As before in the sensory pathway, the model assumes the the observation is corrupted by noise:

$$\mathrm{TV}_k^{\mathrm{Obs}} = \mathrm{TV}_k + \phi_k\,\mathrm{TV}_k + \beta_k, \tag{9}$$

with $\beta_k \sim N(0, R^2_{\mathrm{pred}})$ and $\phi_k \sim N(0, D^2_{\mathrm{pred}})$ as the additive and multiplicative noises, and with values $R^{\mathrm{Pred}} = \sigma_{\mathrm{pred,add}}$ and $D^{\mathrm{Pred}} = \sigma_{\mathrm{pred,mult}}$.

Thus, in the same way as before, the noise-compensated prediction of target velocity may be obtained using Kalman filtering:

$$\widehat{\mathrm{TV}}_{k+1}^{\mathrm{Pred}} = \widehat{\mathrm{TV}}_k^{\mathrm{Pred}} + B_{\mathrm{int}}\, u_k + K_k^{\mathrm{Pred}}(\mathrm{TV}_k^{\mathrm{Obs}} - \widehat{\mathrm{TV}}_k^{\mathrm{Pred}}) + \varepsilon_k, \tag{10}$$

$$K_k^{\mathrm{Pred}} = \Sigma_k^{\mathrm{Pred}}(\Sigma_k^{\mathrm{Pred}} + R^2_{\mathrm{pred}} + D^2_{\mathrm{pred}}(\Sigma_k^{\mathrm{Pred}} + \widehat{\mathrm{TV}}_k^{\mathrm{Pred}}\,\widehat{\mathrm{TV}}_k^{\mathrm{Pred}T})D^{2\,T}_{\mathrm{pred}})^{-1} \tag{11}$$

The internal noise is given as $\hat{\varepsilon}^k \sim N(0, \Omega_e^2)$ with the predictive uncertainty given as:

$$\Sigma_{k+1}^{\mathrm{Pred}} = Q^2_{\mathrm{pred}} + \Omega_e^2 + (1 - K_k^{\mathrm{Pred}}) \tag{12}$$

As before, the Kalman estimation noise is set a $\Omega_e = 0.3$.

In the original paper, the target velocity in memory is set as the target velocity t+150 ms in the future for the previous trial, with additive and multiplicative noise:

$$(\widehat{\mathrm{TV}}_k^{\mathrm{Mem}})_{\mathrm{Trial}\,j+1} = (\widehat{\mathrm{TV}}_{k+150}^{\mathrm{Pred}})_{\mathrm{Trial}\,j}(1 + \delta_{\mathrm{mult}}^{\mathrm{Pred}}) + \delta_{\mathrm{add}}^{\mathrm{Pred}} \tag{13}$$

where $\delta_{\mathrm{mult}}^{\mathrm{Pred}} \sim N(0, \mathrm{Int}^2_{\mathrm{mult}})$ and $\delta_{\mathrm{add}}^{\mathrm{Pred}} \sim N(0, \mathrm{Int}^2_{\mathrm{add}})$, with $\mathrm{Int}_{\mathrm{add}} = 1$ and $\mathrm{Int}_{\mathrm{mult}} = 0.1$.

2.2.3 Motion Pathway

The motion pathway is summarized in the following figure:

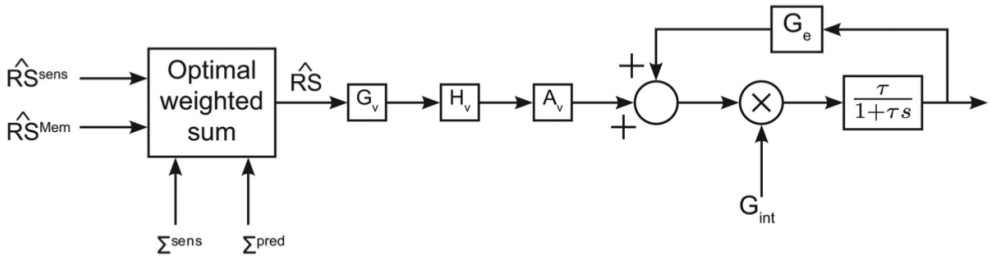

Figure 3: The motion pathway. Sensory and predictive RS are weighted by uncertainty, sent to a classical pusuit model, and then a leaky integrator. [3]

The original paper bases the motion pathway off of much prior art. First, the two pathways are combined into a single estimated RS value weighed using the computed confidences:

$$\widehat{\mathrm{RS}}_k = \frac{\sum^{\mathrm{Pred}}}{\sum^{\mathrm{Pred}} + \sum^{\mathrm{Sens}}} \widehat{\mathrm{RS}}_k^{\mathrm{Sens}} + \frac{\sum^{\mathrm{Sens}}}{\sum^{\mathrm{Pred}} + \sum^{\mathrm{Sens}}} \widehat{\mathrm{RS}}_k^{\mathrm{Mem}} \tag{14}$$

Next, the signal is filtered with a linear function, second order filter, and a gain element:

$$G_v(u) = 7u, \tag{15}$$

$$H_v(s) = \frac{35^2}{s^2 + 2 \times 0.8 \times 35 \times s + 35^2}, \tag{16}$$

$$A_v = 0.9 \tag{17}$$

Finally, the signal was then sent to a leaky integrator called a positive efferent copy feedback loop, characterized by two variables G_{int} and τ. The gain element was set to $G_e = \frac{1}{\tau}$, $\tau = 100$ ms, and G_{int} set as either 0.5 or 0.6 dynamically modulated in the paper to explore effects such as predictive recovery of eye velocity upon blanking. The leaky integrator was set with a leaky factor of 0.5.

2.3 Implementation

A biological plausible implementation of the unified sensory and predicted smooth pursuit model is possible by expressing the dynamics in terms of the NEF. This broadly consists of defining a network of neuron populations with connection transformations that map to the equations described above. Refer to Appendix A for reference to the code. Nengo, a python based library implementing NEF, was used in the implementation.

First, variances were defined as reported in the original paper, along with the other hyperparameters. Functions were defined to return random variables from normal distributions with the appropriate standard deviations and means. These values were scaled down to a factor of 10 to reduce noise to an appropriate level; this was done as an alternative to increasing neuron population sizes to optimize computation. Next, a Nengo node was created for target velocity, the only model input. Other variables necessary to compute the model dynamics were defined as Nengo ensembles. In some cases,

multiple dimensions were needed in the cases of multi-variable equations, such as during the calculation of the Kalman gain. In these cases the dimensionality was increased to hold the necessary signals for computation. Neuron population sizes were allocated 1000 neurons per dimension when collecting results; network effects with smaller and larger population sizes are also explored in the next section. Finally, connections were made between neuron populations to transform their signals as described by the dynamics in the previous section.

Synaptic timings were chosen in a fashion that added up to an 80 ms sensory delay, which is described as typical for the visual system [3, 4, 5], and then modulated as necessary to minimize noise. Notably, the K_{pred} synapse was set to 0.5 s due to high frequency oscillations. K_{sens} and Tv_{ored} synaptic time constants were set to 0.1, and the others set to 0.05 seconds. For the case of the differentiator transformations, which are discussed below, synaptic time constants were set at 0.2 to filter high inherent noise in the approximate differentiation process.

Due to insufficient details on what transformations are necessary to define the internal eye plant, the eye velocity was directly used in the predictive pathway instead of the approximate effective velocity $\dot{e}_k^{\text{eff}}$ used in the paper. The effects of this are less uncertainties within the predictive pathways, which could be added in at a later point. As this value was not computed. Eq. 9 is implemented to use the actual value of TV instead of what the paper describes: summing $\widehat{\text{RS}}_k^{\text{Sens}}$ and $\dot{e}_k^{\text{eff}}$. Again, the effects of this are reduced uncertainties in the predictive pathway.

In two cases there were improper transfer functions that could not be implanted using the `nengo.LinearFilter` class. In these cases, an approximation of first differences was used with a high synaptic time constant of 0.2 was used to minimize noise while still exhibiting the desired effects of a differentiator.

These components assembled together resulted in the global network shown below:

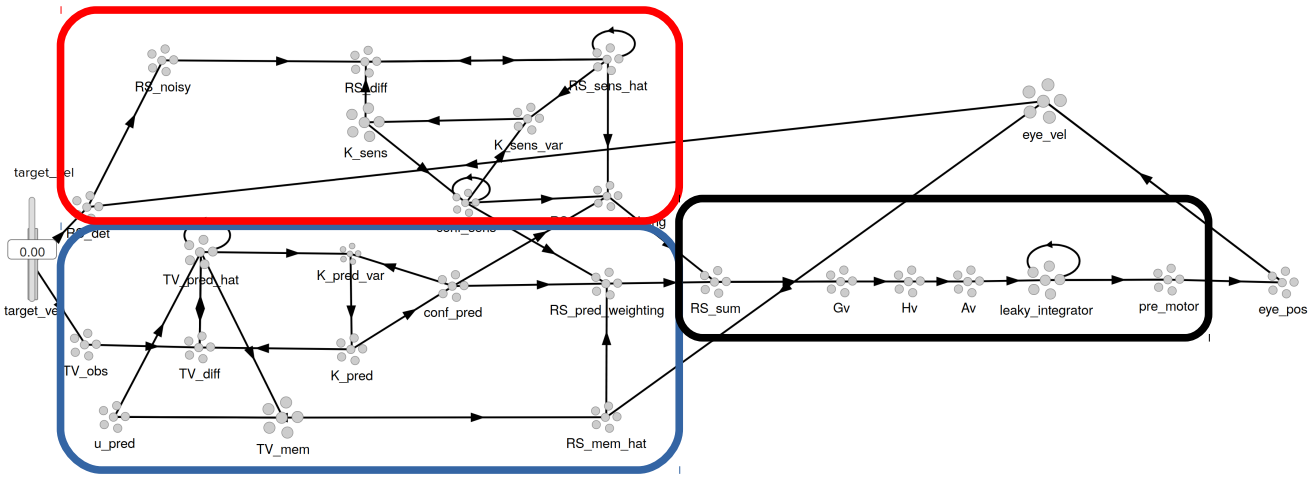

Figure 4: Nengo implementation of unified smooth pursuit netowkr: sensory pathway (red) and predictive pathway (blue) leading to premotor pathway (black) resulting in eye velocity. See Appendix A for code.

3.0 Results

Two main comparisons are made to validate the produced model. The first is against the model described in the paper, and the second is against experimental data.

First, a discussion is provided of the results of the Nengo model when subjected to a constant target velocity input to demonstrate the general case. The outputs of this test is provided below in Figure 5:

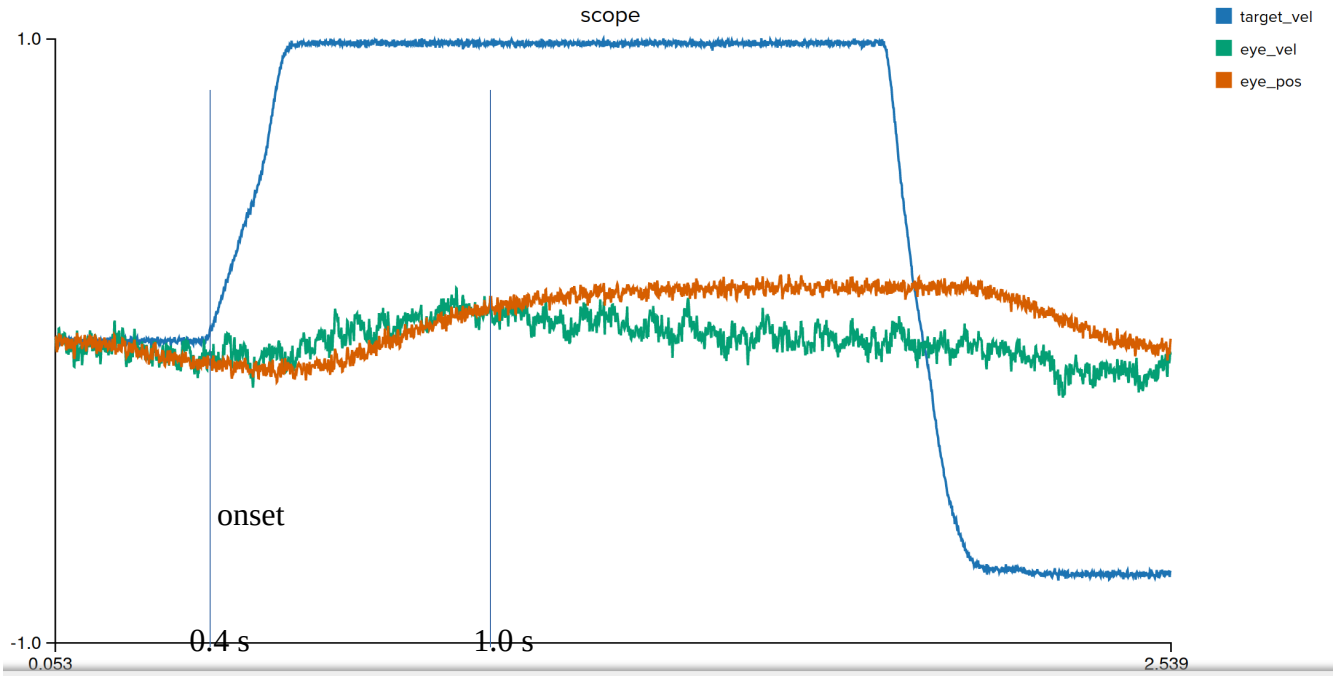

Figure 5: Eye position (red) and velocity (green) when subjected to a target velocity of 100 deg/sec (blue).

While the target velocity is held at 0 between 0 and 0.4 seconds some negative drift of the eye position is observed. A constant target velocity input of the maximum 100 deg/sec results in a positive eye velocity after about 0.2 seconds. The eye position reaches a limit at about 18 deg.

In this case, TV_{mem} lags behind TV_{pred} by a few milliseconds. This corresponds with the notion that if the input were momentarily removed, an internal representation would remain allowing for smooth eye pursuit to continue. This is shown below in Figure 6:

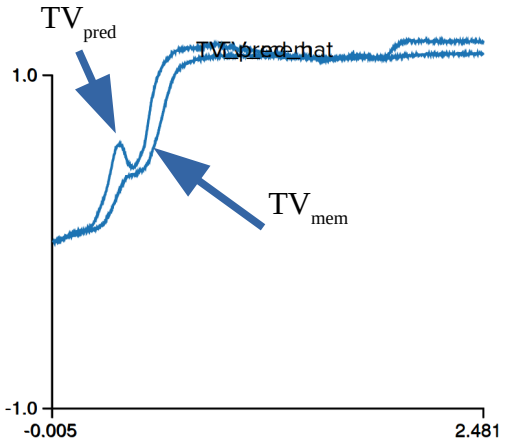

Figure 6: Predicted and memory target velocities with the same input as Figure 5

The corresponding predictive control signal *u* is shown below in Figure 7. Note that upon stability of the input target velocity, the signal slowly leaks and reduces to zero.

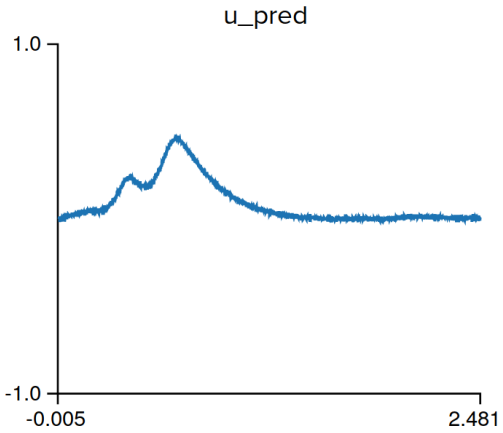

Figure 7: Predictive Control Signal u

3.1 Comparison with MATLAB

To validate that the model implementation was successful, a comparison is first made against the output of the simulations done in the original paper. Two comparisons are made: the first is against a constant input target velocity, and the second is a sinusoidal varying input target velocity with peak velocity of ±6.7 deg/s and a frequency of 0.4 Hz.

The first comparison test, a constant target velocity input, results in the desired model behaviour. The Nengo model produces an output that moves slightly faster than the MATLAB model, with the rise in velocity occurring within 100 ms, whereas the MATLAB model takes nearly 200 ms to react. Some additional overshoot is evident, with a slightly prolonged settling time. In both cases only a single oscillation is observed in the tracking dynamics.

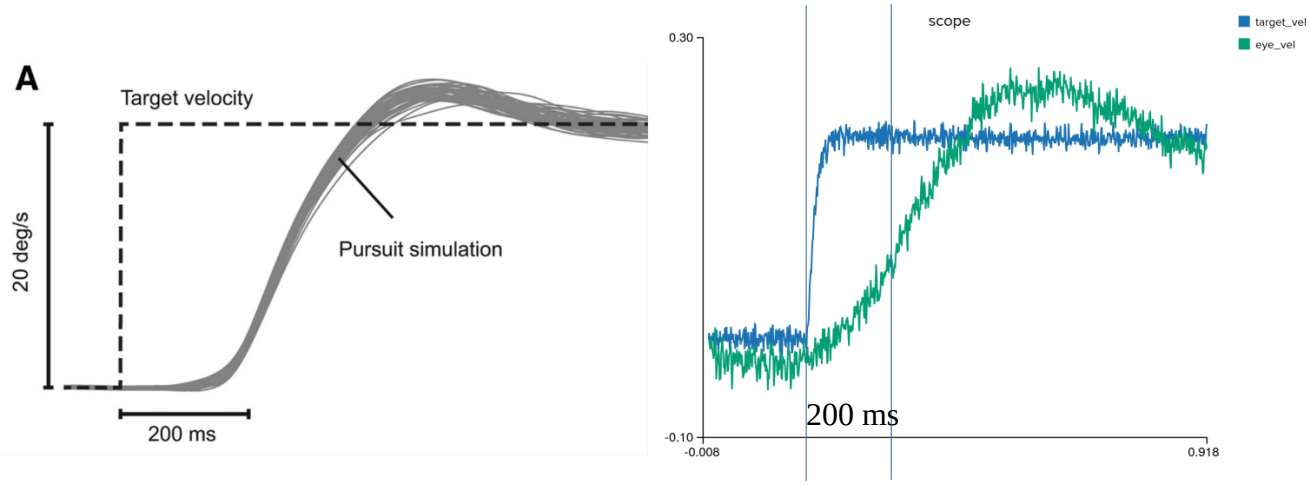

Figure 8: Comparison between MATLAB (left) [3] and Nengo (right) eye velocity outputs given a constant target velocity input.

On the other hand, the Nengo implementation did not manage to perform well upon a sinusoidal stimulus. In this case, a lagging higher frequency sinusoidal output eye velocity was obtained at around 4 times the frequency of the input signal.

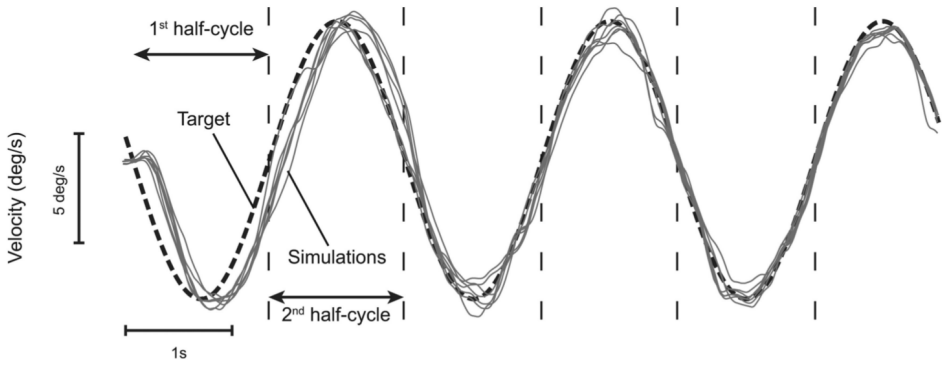

Figure 9: MATLAB simulation of smooth pursuit velocity tracking a sinusoidal input of ±6.7 deg/s and a frequency of 0.4 Hz

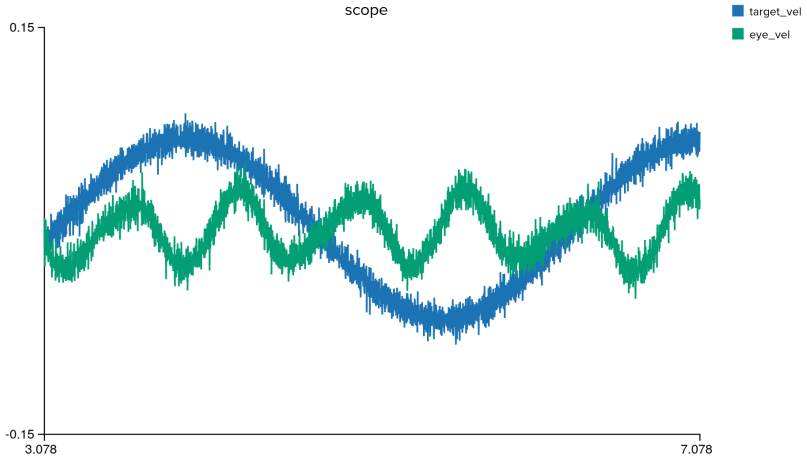

Figure 10: Nengo implementation output tracking the same sinusoid as in Figure 9; results in high frequency oscillations

3.2 Comparison with Experimental Data

To validate the smooth eye pursuit model implemented in Nengo against physiological data, a test was established where an input target velocity would undergo constant acceleration of 8 deg/sec^2 for 2 seconds after the initial stimulus. The results of this are shown below in Figure 11:

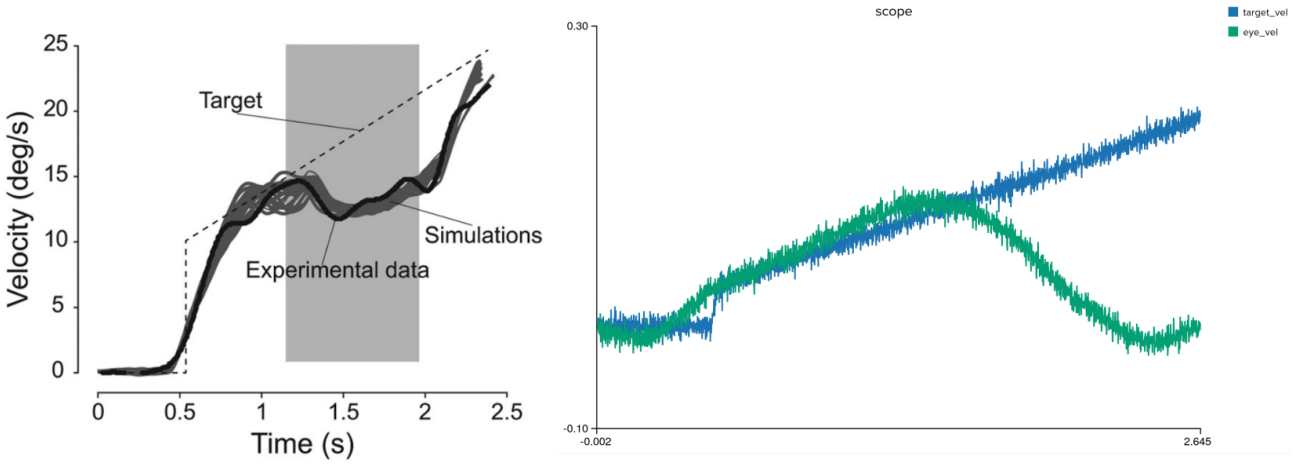

Figure 11: Experimental Data (left) [3] comparision with Nengo smooth pursuit model output (right) upon a constant target acceleration of 8 deg/sec/sec between 0.5 and 2.5 seconds

In this case only some of the desired behaviour is observed. The eye velocity predicts the positive target velocity by several tenths of a second, and matches the acceleration quite well until about 1.7 seconds, at which point it undergoes negative acceleration until past zero, at which point it continues upwards again. This dynamic abnormality of a sudden divergence occurs a majority of the time upon an accelerating target velocity input. The reasons for this are unclear but are suspected to be related to the stability of the dynamics expressed in the Nengo model, including the robustness to noise. Tuning parameters such as synaptic time constants to achieve proper signal processing timings, and hyperparameters such as filter time constants, and eye plant parameters. These all contribute to the dynamics observed and the values that worked for the MATLAB implementation may be suboptimal for the Nengo implementation.

4.0 Conclusions

This paper proposes a novel NEF based implementation of a unified smooth pursuit model that combines noisy sensory inputs using a predictive internal representation. A complete network is produced which exhibits some of the desired dynamics of a smooth eye pursuit system. The implementation of the system using the NEF required several modifications from the original work. These are: custom implementation of the leaky integrator in the premotor circuit, choice of post synaptic time constants, custom implementation of a differentiator, and a reduction of the sizes of most of the random variables representing normal standard deviations.

The produced system performed well under a constant target input, and closely matched the MATLAB implementation results, as shown in Figure 8. However it did not perform well given a sinusoidal input, nor did it perform well compared to both the MATLAB results as well as experimental data for a target undergoing constant acceleration. In the former case, a higher frequency output was observed, suspected to come from oscillatory noise within the feedback loops, and in the latter case a sudden rapid divergence in the wrong direction was observed about 1 second into tracking with good performance. In this case it is suspected that the noise contributed to the unwanted effects.

However, the Nengo implementation, despite not matching all simulated and experimental smooth eye pursuit dynamics, did confirm that the architecture consisting of two Kalman filters in a statistically optimally manner does indeed mitigate against process and observation noise. As well, the predictive pathway component provides a useful input into the system, most notably useful when the input target velocity is occluded, as demonstrated in Figure 7 which shows a resulting helpful predictive control signal upon a constant positive target velocity.

5.0 Recommendations

The following recommendations are proposed for future work continuing with the goal of implementing a unified smooth pursuit model using the NEF:

1. Improve the optimal estimate in the next trial from a simple 150 ms delay implemented as a linear synaptic filter to something more sophisticated. For example, instead of simply using the previous trial, additional information regarding the eye position, or target acceleration may be used to enhance the predicted target motion.
2. Quantitatively determine the effects of noise, including the sensitivity against noise, inherent noise introduced due to the NEF, and the effects of post synaptic timings which act as linear filters.
3. Explore whether timings may be more closely matched to experimental data by modifying the post synaptic timings.
4. Determine whether or not scaling down the noise parameters are a suitable alternative to increasing the neuron population sizes, as described in Section 2.3.
5. Determine if model dynamics may be described in an alternative fashion more suitable for the NEF, for example, an identity transformation between two neuron population introduces some noise; perhaps it is possible to exploit this inherent noise instead of simulating artificial noise.
6. Continue to explore the differences between experimental data, simulated data, and the data obtained using the Nengo implementation; for example, determine exactly the source of the high frequency noise upon sinusoidal input target velocity. Furthermore, determine an appropriate mechanism to mitigate against these unwanted dynamic effects.

References

1. S. G. Lisberger and L. E. Westbrook, Properties of Visual Inputs that Initiate Horizontal Smooth Pursuit Eye Movements in Monkeys" in *Society for Neuroscience*, vol. 5, pp. 1662-1673, June 1985.
2. D. A. Robinson *et. al.*, "A model of the Smooth Eye Pursuit Eye Movement System" in Biological Cybernetics, vol. 55, pp. 43-57, 1986.
3. J. J. Orban de Xivry *et. al.*, "Kalman Filtering Naturally Accounts for Visually Guided and Predictive Smooth Pursuit Dynamics" in *J. Neuroscience*, vol 33, pp. 17301-17313, Oct. 30, 2013.
4. P. Thier and U. J. Ilg, "The Neural Basis for Smooth-Pursuit Eye Movements" in *Current Opinion on Neurobiology*, vol. 15, pp. 645-652, Nov. 2005.
5. A. T. Bahill, "Modeling the Smooth-Pursuit Eye-Movement System" in *Automedica*, vol. 19, pp. 1-37, 2000.
6. A. T. Bahill and J. D. McDonald, "Smooth Pursuit Eye Movement In Response To Predictable Target Motions" in *Vision Res.*, vol. 23, pp. 1573-1583, Apr. 1983.
7. O. V. Komogortsev and J. I. Khan, "Eye Movement Prediction by Occularmotor Plant Kalman Filter with Brainstem Control" in *J. of Control Theory and Applications*, vol. 7, Apr. 2008.
8. T. Bekolay, *et. al.*, "Nengo: A Python Tool for Building Large-Scale Functional Brain Models" in *Frontiers in Neuroinformatics*, vol. 7, Jan. 2012.

Appendix

Appendix A: Nengo Code

```
import nengo
import numpy as np

__author__ = 'yazan'
# Based on "Kalman Filtering Naturally Accounts for Visually Guided and
#           Predictive Smooth Pursuit Dynamics" by Orban de Xivry et. al.

model = nengo.Network()

with model:

    # set standard deviations for random distributions
    psi_sens_add, psi_sens_mult = 10/10, 1.5/10
    int_pred_add, int_pred_mult = psi_sens_add/2, psi_sens_mult/2
    Q, R, D, OM = 1, psi_sens_add, psi_sens_mult, 0.3
    Q2, R2, D2, OM2 = 1, int_pred_add, int_pred_mult, 0.3
    tv_pred_synapse = 0.05
    leaky_int_synapse = 0.05
    eye_vel_synapse = 0.2 # for differentiator
    Gv_synapse = 0.2 # for differentiator
    u_pred_synapse = 0.2 # for differentiator
    conf_sens_synapse = 0.05
    RS_sens_hat_synapse = 0.05
    K_pred_synapse = 0.5
    K_sens_synapse = 0.1
    TV_pred_synapse = 0.1
    B_int = 1
    def di_add():
```

```
    return np.random.normal(scale=psi_sens_add)
def di_mult():
    return np.random.normal(scale=psi_sens_mult)
def beta_pred():
    return np.random.normal(scale=R2)
def psi_pred():
    return np.random.normal(scale=D2)
def epsilon_pred():
    return np.random.normal(scale=OM2)
def nt_sens():
    return np.random.normal(scale=OM)
def di_add_pred():
    return np.random.normal(scale=int_pred_add)
def di_mult_pred():
    return np.random.normal(scale=int_pred_mult)
G_int = 0.5 # also used 0.6
tau_motor = 0.1 # seconds
G_e = 1/tau_motor
T_1 = 0.170 # seconds
T_2 = 0.013 # seconds

mult = 2

RS_det = nengo.Ensemble(n_neurons=500*mult, dimensions=1)
RS_noisy = nengo.Ensemble(n_neurons=500*mult, dimensions=1)
K_sens_var = nengo.Ensemble(n_neurons=1000*mult, dimensions=2)
K_sens = nengo.Ensemble(n_neurons=1000*mult, dimensions=1, radius=1)
# first dim holds conf_sens, second holds [1-k]
conf_sens = nengo.Ensemble(n_neurons=1000*mult, dimensions=2, radius=1)
RS_sens_hat = nengo.Ensemble(n_neurons=1000*mult, dimensions=1)
# first dim holds RS_noisy-RS_sens_hat, second holds k
RS_diff = nengo.Ensemble(n_neurons=1000*mult, dimensions=2)
```

```
eye_vel = nengo.Ensemble(n_neurons=500*mult, dimensions=1)
target_vel = nengo.Node([0])

TV_pred_hat = nengo.Ensemble(n_neurons=1000*mult, dimensions=1)
TV_diff = nengo.Ensemble(n_neurons=1000*mult, dimensions=2)
TV_obs = nengo.Ensemble(n_neurons=500*mult, dimensions=1)
K_pred = nengo.Ensemble(n_neurons=1000*mult, dimensions=1)
K_pred_var = nengo.Ensemble(n_neurons=1000*mult, dimensions=2)
conf_pred = nengo.Ensemble(n_neurons=500*mult, dimensions=1)
u_pred = nengo.Ensemble(n_neurons=500*mult, dimensions=1)
TV_mem = nengo.Ensemble(n_neurons=500*mult, dimensions=1)
RS_mem_hat = nengo.Ensemble(n_neurons=1000*mult, dimensions=1)

RS_sens_weighting = nengo.Ensemble(n_neurons=1500*mult, dimensions=3)
RS_pred_weighting = nengo.Ensemble(n_neurons=1500*mult, dimensions=3)
RS_sum = nengo.Ensemble(n_neurons=500*mult, dimensions=1)
Gv = nengo.Ensemble(n_neurons=500*mult, dimensions=1)
Hv = nengo.Ensemble(n_neurons=500*mult, dimensions=1)
Av = nengo.Ensemble(n_neurons=500*mult, dimensions=1)

leaky_integrator = nengo.Ensemble(n_neurons=500*mult, dimensions=1)
filt_leaky_integrator = nengo.Ensemble(n_neurons=500*mult, dimensions=1)

pre_motor = nengo.Ensemble(n_neurons=500*mult, dimensions=1)

eye_pos = nengo.Ensemble(n_neurons=500*mult, dimensions=1, radius=10)

# Equation 1
nengo.Connection(eye_vel, RS_det, transform=-1, synapse=None)
nengo.Connection(target_vel, RS_det, synapse=None)
nengo.Connection(RS_det, RS_noisy,
                 function=lambda x: x+x*di_mult()+di_add())
```

```
# Equation 5
nengo.Connection(conf_sens[0], K_sens_var[0])
nengo.Connection(RS_sens_hat, K_sens_var[1])
nengo.Connection(K_sens_var, K_sens, synapse=K_sens_synapse,
                 function=lambda x: x[0]*(1/(x[0]+(R**2)+\
                                      (D**2)*(x[0]+x[1]*x[1])*(D**2))))
# Equation 6
nengo.Connection(K_sens, conf_sens[1], synapse=conf_sens_synapse,
                 function=lambda x: conf_sens_synapse*(1-x))
nengo.Connection(conf_sens, conf_sens[0], synapse=conf_sens_synapse,
                 function=lambda x: Q**2+OM**2+x[0]*x[1])
# Equation 4
nengo.Connection(RS_noisy, RS_diff[0])
nengo.Connection(RS_sens_hat, RS_diff[0], transform=-1)
nengo.Connection(K_sens, RS_diff[1])
nengo.Connection(RS_diff, RS_sens_hat, synapse=RS_sens_hat_synapse,
            # scaling by synaptic time constant performs poorly
            #function=lambda x: RS_sens_hat_synapse*(x[0]*x[1]+nt_sens()))
                 function=lambda x: x[0]*x[1]+nt_sens())
nengo.Connection(RS_sens_hat, RS_sens_hat, synapse=RS_sens_hat_synapse)
# Equation 7
nengo.Connection(TV_mem, RS_mem_hat)
# we just directly use eye_vel as it is not clear
# what "internal eye plant" is in the paper for effective eye velocity
nengo.Connection(eye_vel, RS_mem_hat, transform=-1)
# Equation 9
nengo.Connection(TV_mem, u_pred, synapse=None)
nengo.Connection(TV_mem, u_pred, transform=-1, synapse=u_pred_synapse)
# Equation 10
# later to change target_vel to
# be the sum of RS_sens_hat and pred_eye_vel
# see Eq. 8 explanation
```

```
nengo.Connection(target_vel, TV_obs,
                 function=lambda x: x+x*beta_pred()+psi_pred())
# Equation 11
nengo.Connection(TV_pred_hat, TV_diff[0], transform=-1)
nengo.Connection(TV_obs, TV_diff[0])
nengo.Connection(K_pred, TV_diff[1])
nengo.Connection(TV_diff, TV_pred_hat, synapse=tv_pred_synapse,
            # as before, multiplying by tv_pred_synapse performs poorly
                 function=lambda x: x[0]*x[1]+epsilon_pred())
nengo.Connection(TV_pred_hat, TV_pred_hat, synapse=tv_pred_synapse)
nengo.Connection(u_pred, TV_pred_hat, transform=B_int,
                 synapse=tv_pred_synapse)
# Equation 12
nengo.Connection(conf_pred, K_pred_var[0])
nengo.Connection(TV_pred_hat, K_pred_var[1])
nengo.Connection(K_pred_var, K_pred, synapse=K_pred_synapse,
                 function=lambda x: x[0]*(1/(x[0]+(R2**2)+\
                                     (D2**2)*(x[0]+x[1]*x[1])*(D2**2))))
# Equation 13
nengo.Connection(K_pred, conf_pred, function=lambda x: Q2**2+OM2**2+(1-x))
# Equation 14
nengo.Connection(TV_pred_hat, TV_mem, synapse=TV_pred_synapse,
                function=lambda x: x*(1+di_mult_pred())+di_add_pred())
# Equation 15
nengo.Connection(conf_sens[0], RS_sens_weighting[0])
nengo.Connection(conf_pred, RS_sens_weighting[1])
nengo.Connection(RS_sens_hat, RS_sens_weighting[2])
nengo.Connection(RS_sens_weighting, RS_sum,
                function=lambda x: (x[1]/(x[0]+x[1]))*x[2])
nengo.Connection(conf_sens[0], RS_pred_weighting[0])
nengo.Connection(conf_pred, RS_pred_weighting[1])
nengo.Connection(RS_mem_hat, RS_pred_weighting[2])
```

```
    nengo.Connection(RS_pred_weighting, RS_sum,
                     function=lambda x: (x[0]/(x[0]+x[1]))*x[2])
    # Equation 16
    # since the following does not work (improper transfer function)
    #nengo.Connection(RS_sum, Gv,
    #                 synapse=nengo.LinearFilter([7, 0], [0, 1]))
    nengo.Connection(RS_sum, Gv)
    nengo.Connection(RS_sum, Gv, transform=-1, synapse=Gv_synapse)
    # Equation 17
    nengo.Connection(Gv, Hv,
                     synapse=nengo.LinearFilter([35**2], [1,2*0.8*35, 35**2]))
    # Equation 18
    nengo.Connection(Hv, Av, transform=0.7)

    # Leaky Integrator, Fig 3
    #nengo.Connection(filt_leaky_integrator, leaky_integrator,
    #                 synapse=leaky_int_synapse, transform=G_e)
    #nengo.Connection(Av, leaky_integrator,
    #                 synapse=leaky_int_synapse)
    #nengo.Connection(leaky_integrator, filt_leaky_integrator, transform=G_int,
    #                 synapse=nengo.LinearFilter([1],
    #                                            [1, 1/tau_motor]))
    # the above leaky integrator does not work, as described in the paper
    # so we create our own using a basic NEF neural integrator
    nengo.Connection(Av, leaky_integrator, synapse=leaky_int_synapse,
                     transform=leaky_int_synapse*10)
    nengo.Connection(leaky_integrator, leaky_integrator,
synapse=leaky_int_synapse)

    # Premotor system, Fig 1
    nengo.Connection(leaky_integrator, pre_motor, transform=T_1)
    nengo.Connection(leaky_integrator, pre_motor,
```

```
                synapse=nengo.LinearFilter([1], [1, 0]))

# Eye Plant
nengo.Connection(pre_motor, eye_pos,
                synapse=nengo.LinearFilter([1], [T_1*T_2, T_1+T_2, 1]))

# Eye Velocity
# since the following does not work (improper transfer function)
#nengo.Connection(eye_pos, eye_vel,
#                 synapse=nengo.LinearFilter([1, 0], [0, 1]))
nengo.Connection(eye_pos, eye_vel, synapse=None)
nengo.Connection(eye_pos, eye_vel, transform=-1, synapse=eye_vel_synapse)
```