2-Dimensional Spiking Neural Network Expansion for a Neuromorphic Controller of an Autonomous PushBot Robot

D-ITET Bachelor, Group Project - 6 ECTS

Nikola Bolt, ETH Zürich
boltn@ethz.ch

Claudio Fanconi, ETH Zürich
fanconic@ethz.ch

Spring Semester 2019

Abstract

Spiking Neural Networks running on neuromorphic hardware are a relatively new type of artificial neural networks, consisting of neurons and synapses that interact based on equations that are highly inspired by the biological brain. Thus, the factor of time is also being considered and the synapses fire only upon reaching a certain threshold voltage, which are their main differences as compared to "classical" multilayer perceptrons. Spiking Neural Networks are considered to be the third generation of machine learning models and due to their power efficiency, it is especially interesting to use these architectures for embedded systems and robotic applications.
This project is a continuation and extension of [3]. The aim is to implement a two-dimensional Spiking Neural Network architecture that can be trained on measured robot data to learn the forward and especially inverse mappings between angular and translational velocities and motor commands using the Intel Loihi neuromorphic chip.
After expanding the scripts of [3] into two dimensions and successfully testing our 2D networks on artificial data with respect to selectivity and learning ability, we set up experiments to record real data from the PushBot robot. We were able to achieve moderate accuracy that was varying for different example routes taken by the robot.

Contents

1 Introduction

Naturally, any technical tool or piece of equipment is only useful if one is able to control what it does. This is especially true for robotics, which is why knowledge of the relation between desired actions of the robot and possible command inputs is crucial. The relation describing the robot's movement given the motor commands is called forward mapping; conversely, the inverse mapping describes the required inputs depending on the desired actions.

Many of the controllers that are currently in use, e.g. in industrial systems, are based on some kind of PID or LQR control mechanisms. While these controllers can and do achieve great results even with model inaccuracies and potential disturbances present, their abilities to adapt to major changes in the environment are limited. To overcome this problem, a lot of research is being done in terms of using neural networks for control systems. However, another problem arises when using classical multilayer perceptrons: They are computationally, and thus also energetically expensive, often needing to run on graphical processing units (GPU).

This is where spiking neural networks (SNN) running on neuromorphic hardware come in. As their event-based mode of operation is fundamentally different from that of classical artificial neural networks (see section 4), both software- and hardware-wise, they allow for the creation of much more energy-efficient robotic controllers. A proof of concept for this kind of controller has already been provided by [5], where the researchers were able to successfully perform online training of a spiking neural network on the ROLLS [8] device to control a PushBot robot (see section 2). It is especially remarkable that practically the whole closed loop system (with the exception of communication with the robot) has been implemented using only an SNN architecture; in particular, also a PI-controller used for the initial space exploration.

In this project, we will examine two-dimensional spiking neural networks on the Intel Loihi neuromorphic chip (see section 3), where each dimension in the network directly corresponds to one of the input or output dimensions, respectively. This will be done by first expanding the winner-take-all (WTA) architectures built and analysed in [3] and then initially testing the selectivity and learning abilities of the two-dimensional networks on artificial inputs. Subsequently, we will record data on the PushBot robot for four specific routes and examine whether a network consisting of two two-dimensional WTA layers connected via plastic synapses is able to learn the forward and inverse mappings for these routes in an offline fashion using real-world recorded data.

2 PushBot

For this project, the PushBot[1] created by TU Munich was used since it has two degrees of freedom which matches the dimensionality of the network that we want to build. It can be controlled by setting the angular velocities of the lateral motors on each of its two sides independently. In total, there are 4 motors; although, the corresponding two motors that are on the same side of the robot are connected via continuous tracks, creating tank-like movement. It is also equipped with an Inertial Measurement Unit (IMU), which consists of an accelerometer and a gyroscope. Thus, measurements of linear acceleration in all three Cartesian directions and measurements of pitch, roll and yaw angular velocities can be extracted.
Additionally, the PushBot features an event-based camera, as well as two laser pointers. These won't be used in this project, though.

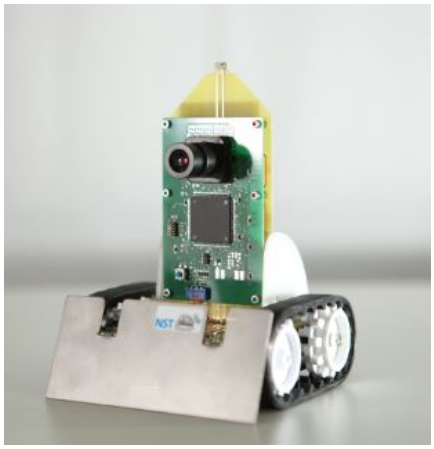


Figure 1: PushBot Robot

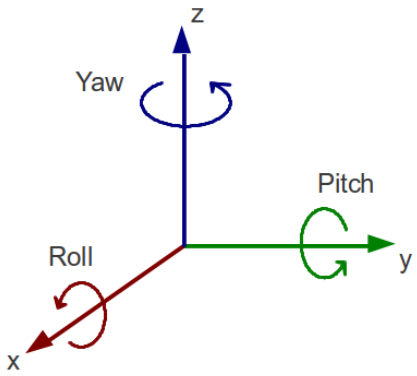


Figure 2: Pitch, roll and yaw

Thanks to a previous project, the PushBot can be connected to a computer via TCP connection through a wireless access point. The robot's actions can then be passed via terminal input or by implementing movement-functions in C++.
The programmable interface allows for collecting the various measurements from the IMU, the event-based camera or the wheel encoders, which measure the number of rotations of the PushBot's wheels.

3 Intel Loihi

Loihi[2] is a neuromorphic device created by Intel Labs for research purposes. It is used to build Spiking Neural Networks and performs discrete and asynchronous computations to implement learning and inference efficiently.
The chip can be accessed and programmed from a computer via a USB interface called Kapoho Bay, pictured below.

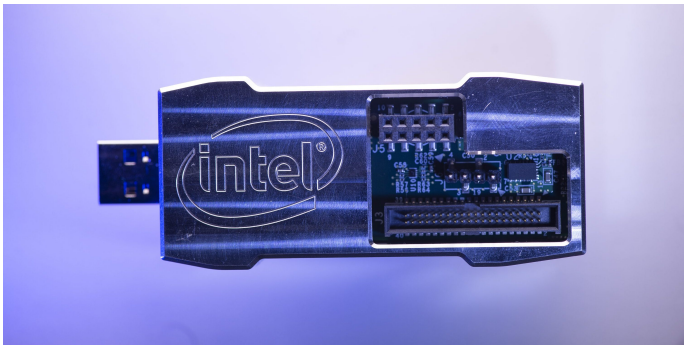


Figure 3: Kapoho Bay USB drive interface for Loihi

3.1 Architecture

Loihi is equipped with a fully asynchronous many-core mesh of 128 neuromorphic cores with 1,024 primitive spiking neural units each. This means that the hardware itself implements a Spiking Neural Network, where its neurons send impulses to their neighbours with directed links. The protocol used allows for the connection of up to 16'384 chips with up to 4096 cores per chip.
What sets the Intel Loihi chip apart from other neuromorphic devices, is that its neuromorphic cores are equipped with integrated learning engines, which enable full on-chip learning for the desired networks.

3.2 Programming

Intel provides multiple APIs on various levels to enable developers to take advantage of Loihi. There exist a C library and a Python core API for lower level purposes. Moreover, there is also a high-level Python API, which simplifies the construction and execution of spiking neural networks and plotting of the results. The networks can then be represented as graphs with custom configurations.

4 Spiking Neural Network Architecture

4.1 General Mode of Operation of SNNs

At first glance, it may seem more canonical to initially explain how information is encoded before moving on to describe how it propagates through the network. In the case of Spiking Neural Networks, however, we first need to understand the basic laws that govern the activations of its neurons before we can think about how to exploit them to encode information and train them to perform calculations.

4.1.1 Propagation of Information

As mentioned in the abstract, a Spiking Neural Network is a special kind of Artifical Neural Network which incorporates various time dependencies in its architecture. Since these networks should mimic the biological brain very closely, neuron activity is modelled according to various differential equations describing the potential build up in the neurons. The one most used in this context is the Leaky-integrate-and-fire differential equation:

$$I(t) - \frac{V_m(t)}{R_m} = C_m \frac{dV_m}{dt} \tag{1}$$

$I(t)$ is the membrane current, $V_m(t)$ the membrane potential, C_m the membrane capacity, and finally, R_m is the membrane Resistance.
Whereas in a classical multilayer perceptron's feedforward algorithm all neurons are active and propagate their activation from one layer to the next one in every step, in a SNN a neuron first needs to build up enough membrane potential $V_m(t)$ over time and reach a certain threshold voltage in order for it to spike and produce an output; it remains inactive otherwise. Potential increases whenever there is a high enough input current $I(t)$ to the respective neuron; it decreases exponentially with time otherwise. If a neuron does spike, it releases a membrane current which propagates through its synapses to all connected neurons scaled by the connection weights. This current, in turn, increases the membrane potential in the post-synaptic neurons, which can then lead to more spikes and so forth. As membrane potential decreases exponentially with time, the spike rates of the neurons are what describe the overall activity and behaviour of the network.

4.1.2 Encoding of Information

In classical neural networks, information is represented naturally by the activation values of its neurons. For SNNs, on the other hand, it wouldn't be sensible at all to use e.g. the exact membrane potential of a neuron to encode information, since it is subject to constant change.
There are two major ways to encode information in Spiking Neural Networks: Rate-coding and Space-coding. As the name suggests, Rate-coding uses the firing frequency of a specific neuron to encode information. For this approach, the spike rate has to be calculated very precisely and it is thus prone to noise and perturbations. For our network, however, we will use Space-coding. With this coding scheme, the information value can be found in the location of the activation peaks of a neuron population. Assuming there is only one

such peak (as in a WTA network, see section 4.2 below), its location can be determined by summing up the location (i.e. the indices) of all neurons F_i multiplied by their firing rate m_i and divided by the sum of all firing rates m_i

$$F = \frac{\sum_i m_i F_i}{\sum_i m_i} \tag{2}$$

The obtained value of F is called the Population Vector. This method should be more robust against noise since we are now using a weighted average. Robustness, especially against outliers, can be further improved by using a Winner-take-all architecture, which will be explained in more detail later.
It should be noted that the explicit calculation of the centre of mass of the activity peak F will not be necessary in a fully neuronal architecture, which must be the ultimate goal in order to harvest the full potential of the energy-efficiency of SNNs running on neuromorphic hardware. In such a scenario, the motors will be driven in a spike-based manner directed by the activity of certain neurons. Intuitively, the spike-rates could be directly setting the motor-voltage, for example, which would then internally also make use of rate-coding.

4.1.3 Training an SNN

General Ideas In supervised learning, a standard feed-forward neural network (DNN) is trained by comparing the model's output with the ground truth and applying the back-propagation algorithm. In an SNN, however, due to its fundamentally different functionality, the model is trained by applying ground truth spikes to the output layer, representing the desired output for given (more or less simultaneous) input spikes. The weights of the synapses are then updated according to a learning rule, which can depend on various factors, such as pre- and postsynaptic spike times and spike rates of the neurons that are connected to a certain synapse.
There exists a variety of learning rules for spiking neural networks. The most basic form of learning is Hebbian learning, which is based on the principle "Neurons that fire together, wire together", meaning that the strength of a synapse is in- or decreased based on the correlation of the outputs of its connected neurons. Then, there is Spike-timing-dependent plasticity (STDP), which builds on Hebbian learning but takes into account the exact times between pre- and postsynaptic spikes to determine the weight update. The last form of learning that will be mentioned here is Rate based learning. As suggested by the name, this kind of learning incorporates the firing rates of pre- and postsynaptic neurons into the weight updates.[4]

Learning Rule For our networks, we will be using a mixture of all the above principles. The structure of the learning rule will be the same as in [3] since they were able to generate satisfactory results using this kind of rule:

$$\Delta w = c_1 * (x_1 - x_{min}) * (w_{max} - w) * y_0 - c_2 * x_0 \tag{3}$$

w describes the weight of the synapse and thus Δw is the weight update, computed at every timestep and accumulated until it is updated after t_{epoch} timesteps, where t_{epoch}

can also be set manually on Loihi when specifying the learning rule for the network. x_0 and y_0 are either 0 if no spike has occurred in the previous timestep or 1 if there was a spike in the pre- (for x_0) respectively post-synaptic (for y_0) neuron. x_1 is the so-called pre-synaptic spike trace, which is increased by an impulse for every pre-synaptic spike and decays exponentially if no spikes arrive. The decay constant is chosen such that x_1 models the pre-synaptic spike rate. w_{max} limits the growth of the weight and x_{min} sets a minimum threshold for the presynaptic spike rate needed for a positive weight update in case of a post-synaptic spike. The last term $(-c_2 * x_0)$ ensures that there is weight depreciation if a pre-synaptic spike doesn't result in the post-synaptic neuron spiking.

4.2 Winner-take-all (WTA) Networks

Winner-take-all describes the property of a single layer of neurons to allow for only one activation peak in the whole layer, i.e. only one coherent group of neurons to be firing at a time. This property can be obtained by setting the synaptic weights in a way that leads to local excitation and global inhibition, meaning the weights of a single neuron to its nearest neighbours should be positive and the synapses connecting it to further away neurons should have negative weights with a gradual transition.[1]
A popular kernel used to achieve this kind of connectivity is the Gaussian kernel, which we will be utilising as well. The connection weights are then set according to a down-shifted Gaussian distribution that depends on the distance between the two connected neurons.

4.2.1 1 Dimensional WTA

An in-depth analysis of the properties of one-dimensional WTA networks has been conducted by the authors of the group project [3] upon which this work builds. Among other things, they have first successfully built a hard WTA on Loihi and demonstrated its selective properties. Subsequently, they were able to approximate different functions by training a network consisting of an input and an output layer WTA connected by plastic synapses. Finally, they were also able to show that generalization is possible under the right conditions using a third "Radial Basis Function" neuron layer in the middle that is connected to the input WTA via Gaussian receptive fields and has plastic connections to an additional "readout" layer in front of the output layer.

4.2.2 2 Dimensional WTA

Our first task was to expand the scripts and functions that [3] have already built for one-dimensional WTAs into two dimensions and establish some of the properties of the new networks.

Implementation We wanted the multidimensional functionality to be seamlessly integrated into the already existing library of the previous project. Thus, all the necessary

[1] More formally, this kind of WTA is called a selective or hard WTA. There are also soft WTAs which allow for multiple peaks of activity to form, but we will not cover these as they won't be used in our network.

functional extensions had to be made in the script `wta1d.py`, which contains the wrapper class and helper functions for creating WTA networks and adding spikes to them. We implemented the new features in such a way that it is of course still possible to use all the 1D analysis scripts without any changes but it is now also possible to build and analyse 2D networks using the same underlying `wta1d.py`[2] script. The upgraded underlying script will differentiate between the cases of one and two dimensions implicitly using the arguments provided by the calling script and act accordingly. Namely, if the size argument is a tuple of dimension two instead of a single integer, it is clear that the user wishes to build a two-dimensional network. To build 2D WTAs, under the hood we used a C-style row-major representation. This in particular influences how the connections of the neurons in the WTA layer need to be set in order to obtain the expected multidimensional behaviour. As an example, we have appended the code used for generating the connection matrix, see Appendix [A].

Testing for Selectivity As a next step, we tested our implementation of two-dimensional WTAs with respect to their ability to select one of the multiple peaks of activations and also filter out noise. To do so, similarly to the approach that [3] used, we injected artificial spikes with multiple centres and Gaussian noise into the network using a spike generator and observed the resulting spikes in the WTA layer. In order to include the time axis in the plot as well, the network is shown flatted out in Figure 4, which is why there are always three distinct connected spike trains, corresponding to the three rows of nearest neighbours to a certain neuron. The last plot in Figure 4 shows the error evolution over time. It is important to note that for the error we, of course, didn't use the flatted out population vectors (as depicted in the two plots above the error plot) but rather the Euclidean distance between the expected and actual two-dimensional population vectors.
To further aid visualisation, Figure 5 shows a 2D representation of two pairs of snapshots of the input spikes next to the filtered WTA spikes at different times. The lighter a square in the image, the higher the firing rate of the neuron in question.

[2]The name should probably be changed, though.

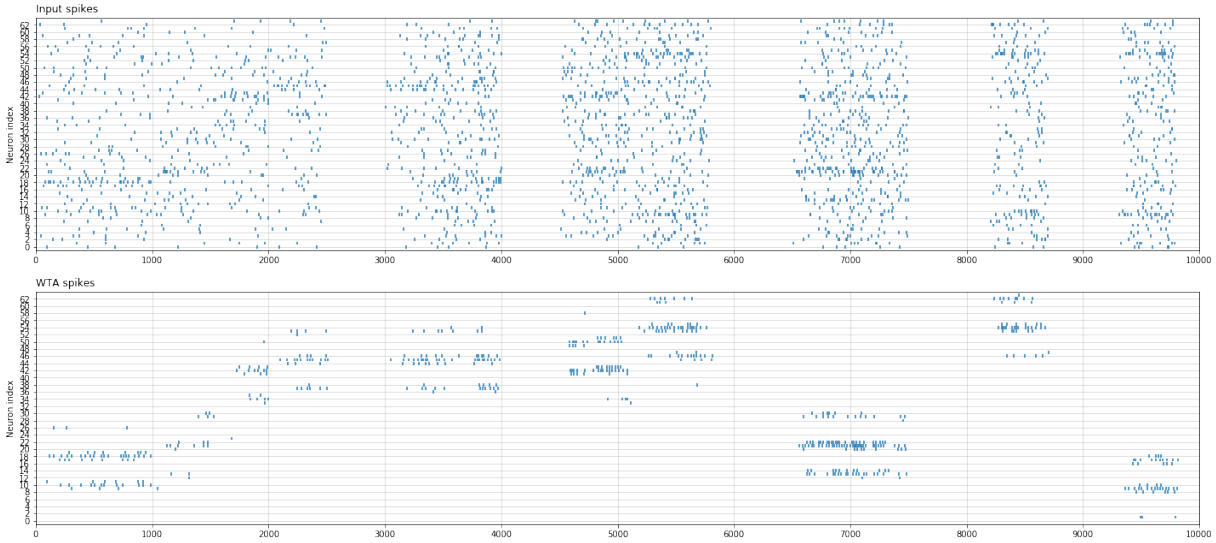


Figure 4: Spikes injected into network and resulting filtered spikes in WTA

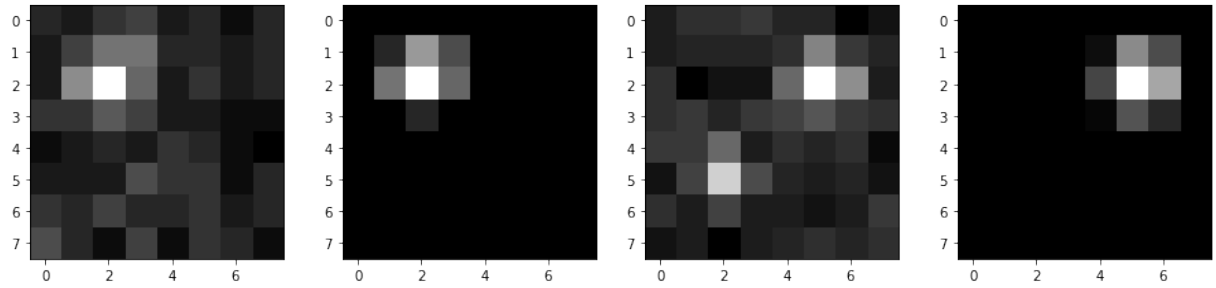


Figure 5: Two pairs of input and WTA spikes at different times

It can clearly be observed that, on one hand, the network is able to select one of the multiple activation peaks and, on the other hand, it successfully filters out most of the noise.
The above plots have been generated using a single WTA layer with 8×8 neurons connected via a two-dimensional Gaussian kernel with a standard deviation of $1.5I$, maximum synaptic weights of 200 and minimum weights of -160.

Testing learning ability Consequently, we proceeded to examine whether a network consisting of two two-dimensional WTA layers connected via all-to-all plastic synapses would be able to learn to approximate a simple linear function such as the identity with noise present. This experiment was performed in two stages: In the training phase (Timesteps 0 to 16500), we iterated through all neurons and injected spikes both in the input and output layers simultaneously according to the identity mapping; additionally, Gaussian noise spikes were injected as well. This was followed by a testing phase (16500 - 31000), where the same spikes were injected but only in the input layer in order to observe whether the newly learned plastic synapses will lead to the expected spikes in the output layer. We again provide a graph of the flatted out view of the network over time and two

pairs of two-dimensional snapshots.

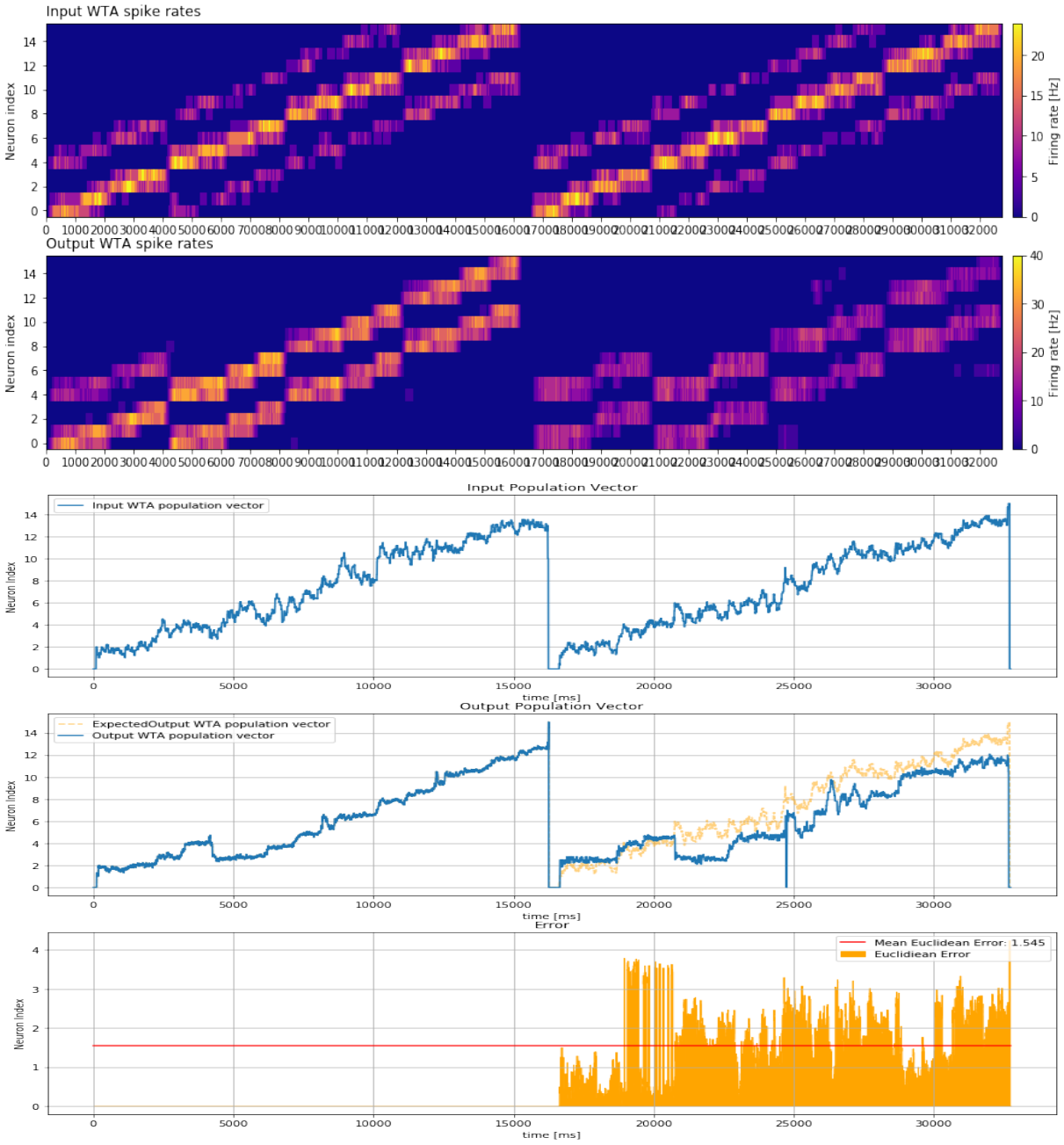


Figure 6: Approximating the identity function

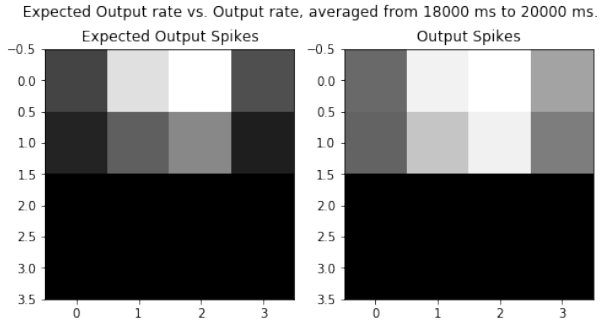


Figure 7: Outputs between 18 and 20 s.

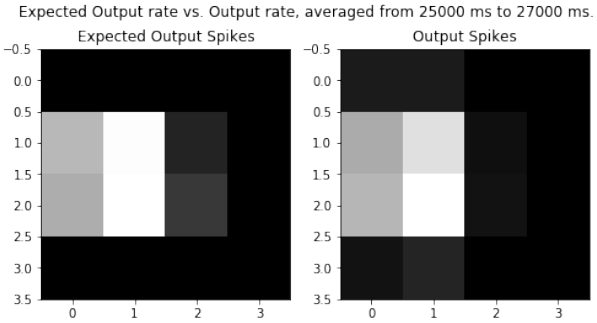


Figure 8: Ouputs between 25 s and 27 s.

As can be seen, the 2D network is able to learn the identity mapping up to some degree of accuracy. We can observe that the population vector is mostly to low (with respect to the neuron index) which could be due to the fact that we start training from the bottom. These plots have been generated with the same parameters as before in a 4×4 network. We also tried using more neurons but we weren't able to achieve the same accuracy; more tweaking of the parameters would have been necessary (both of the WTAs and the learning rule) to optimize the network for a different size.

5 Spiking Neural Network and PushBot Interaction

5.1 General Approach

The basic idea is to represent each pair of translation and rotation speeds as a single neuron in a two-dimensional WTA layer and each pair of motor commands as a single neuron in a second WTA. The two WTA layers are then connected according to a 4-dimensional plastic connection weight matrix (see Figure 9).
If we take the forward mapping, for example, the input to the network would be the motor commands and the output would be the rotational and translational velocities resulting from these motor commands (taken from the IMU data). An input to the network can be generated by externally injecting spikes into the input layer WTA; on Loihi, this can be done by using a SpikeGenerator. Thus, we first need to translate motor commands of the PushBot (as well as IMU data for training of the network) to spikes in the network, which are characterised by the neuron spiking and the time of the spike. The spike time is naturally given by the time at which the respective command was sent (for the PushBot, this can be logged). Additionally, we implemented the functionality to upscale the spike rate by duplicating certain spikes. The rate is at 50 Hertz by default, which corresponds to the frequency of motor commands and IMU measurements. To determine which neuron to send the spike to, we need to bin all the motor commands into $m*n$ bins where $m \times n$ are the input WTA dimensions, which of course limits the resolution. These "motor command spikes" will then lead to various activation peaks throughout time in the input layer, which will then propagate through the network and lead to activation peaks in the output WTA. Assuming the plastic synapses have already been trained successfully, the peaks in the output layer will be representing the expected IMU data for the given commands - again, with a lowered resolution corresponding to the size of the output layer. The same reasoning, of course, applies for the inverse mapping as well. The exact parameters used will be described in section 6.2. The code used for translation of motor command and IMU data to spikes in the SNN can be found in Appendix [B].

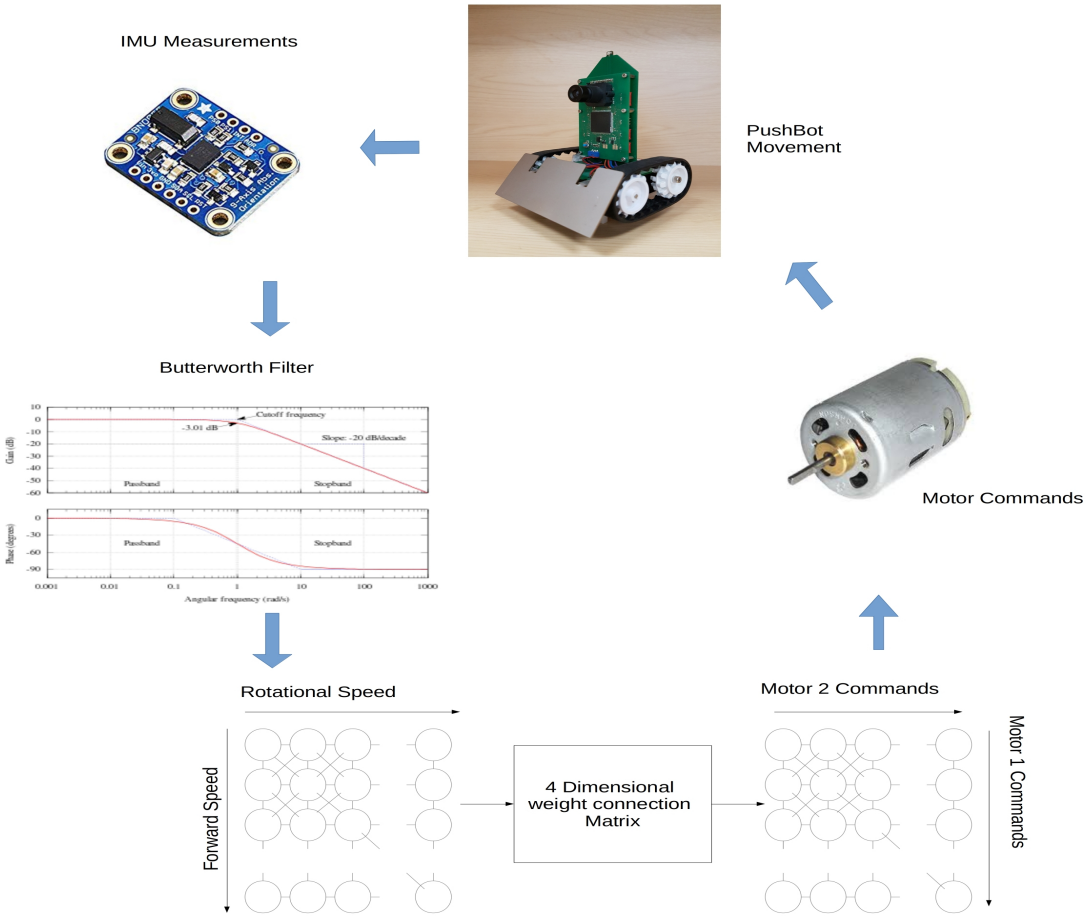
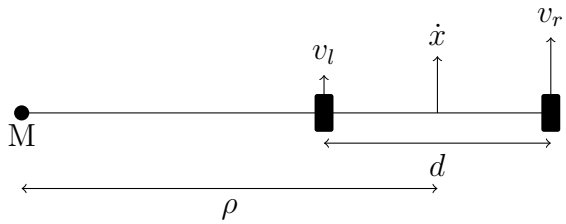


Figure 9: Workflow of Data Collection and Learning

5.2 Equations of motion of the PushBot

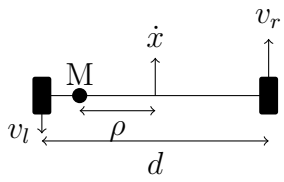
We can use the following abstracted model to describe the physical relation between the velocities of the left and right wheels (tracks)[3] v_l and v_r and the angular and forward velocities ω and $\dot{x}$, respectively:



At any particular instant, the movement of the PushBot can be described as a circular revolution with radius ρ around a midpoint M, where $|\rho| \to \infty$ in the limit of linear

[3]With respect to military tanks, they are technically called "road wheels".

movement. We can use this model due to the fixed axle of the PushBot between its two tracks, which cannot be turned. It should be noted that the above picture only applies for $v_l < v_r$ and $v_l > 0$. For $v_l < 0$ and $v_r > 0$ with $|v_r| > |v_l|$ we would have something like:



All the other cases can be derived symmetrically and the following formulas are valid for all combinations of v_l and v_r:

$$v_l = (\rho - \frac{d}{2})\omega \tag{4}$$

$$\dot{x} = \rho\omega \tag{5}$$

$$v_r = (\rho + \frac{d}{2})\omega \tag{6}$$

where $d = 8.9cm$ is the distance between the wheels of the PushBot and ρ is defined to be the distance (to the left) from the midpoint of the PushBot's wheel axle to the centre of its rotation M (note that $\rho < 0$ is possible).

Using (4) and (6) we find that

$$\rho = \frac{dv_l}{v_r - v_l} + \frac{d}{2} = d\frac{v_l + v_r}{2(v_r - v_l)} \tag{7}$$

and

$$w = \frac{v_r - v_l}{d} \tag{8}$$

Thus, substituting for ρ and ω in (5) we arrive at

$$\dot{x} = \frac{v_r + v_l}{2} \tag{9}$$

The goal is for the spiking neural network to learn the forward and especially inverse mappings between angular and forward velocities and motor commands (angular velocities) ω_l and ω_r of the left and right motors on the PushBot. These are linearly linked to the velocities v_l and v_r of the tracks via the equations $v_l = r\omega_l$ and $v_r = r\omega_r$, where $r = 1.9cm$ is the radius of the wheels of the PushBot. Thus, for the forward mapping, the following equations will have to be memorised by the network:

$$w = r\frac{\omega_r - \omega_l}{d} \tag{10}$$

$$\dot{x} = r\frac{\omega_r + \omega_l}{2} \tag{11}$$

For the inverse mapping, the equations will be:

$$\omega_l = \frac{1}{r}(\dot{x} - \frac{d\omega}{2}) \tag{12}$$

$$\omega_r = \frac{1}{r}(\dot{x} + \frac{d\omega}{2}) \tag{13}$$

where we used the fact that $\rho = \frac{\dot{x}}{w}$.

6 Experiment using real IMU and Motor Command Data

6.1 Aquiring Data from PushBot

The first step is to record IMU measurements and the corresponding motor commands. We sought out to make the network learn the mappings for four simple movement patterns, based on this collected data. These four routes are:

- Stop and Go
- Square
- Circle
- Spiral

As mentioned in section 2, the PushBot can be accessed via a TCP connection and the commands are then set by functions written for a C++ interface. The measurements are written to a CSV file every 20ms (50 Hz rate), including accelerometer and gyroscope data in x, y, z directions, according to the current timestamp.
Any turns made by the PushBot were always in the same direction - clockwise to be precise (thus, mathematically negative). This was only due to the fact that the spectrum of values would be smaller which would, in turn, decrease the error from binning the data onto discrete neurons (see Appendix [B]). Having a smaller range of values enabled us to also use a relatively small number of neurons in the SNN.

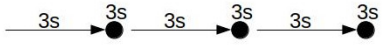


Figure 10: Stop and Go Pattern

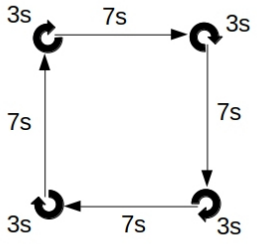


Figure 11: Square Pattern

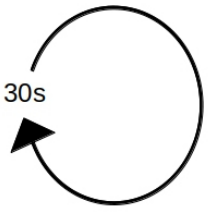


Figure 12: Circle Pattern

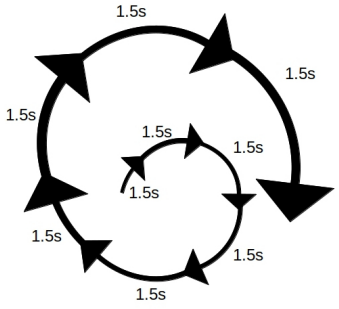


Figure 13: Spiral Pattern

6.1.1 Simulated Input Data

To have a general benchmark for the recorded data, we first simulated the expected IMU measurements of the robot for given motor commands by applying the equations of motion (10) and (11) outlined in section 5.2 to the commands that were given to the PushBot.

6.1.2 Measured Input Data

Due to the large inaccuracies of the IMU, each of the aforementioned routes was performed several times and finally, the most robust measurement was chosen to be used for being fed into the SNN.
After the data was collected via the PushBot interface, it showed a high level of noise. To smoothen it, a Butterworth low pass filter of order 6, with cutoff frequency $\omega_{co} = 2Hz$, was applied (see implementation in Appendix [C]). This is shown in Figure 14 below, where the filter is used to smooth the measurements of the gyroscope's z-axis of the square movement as an example.

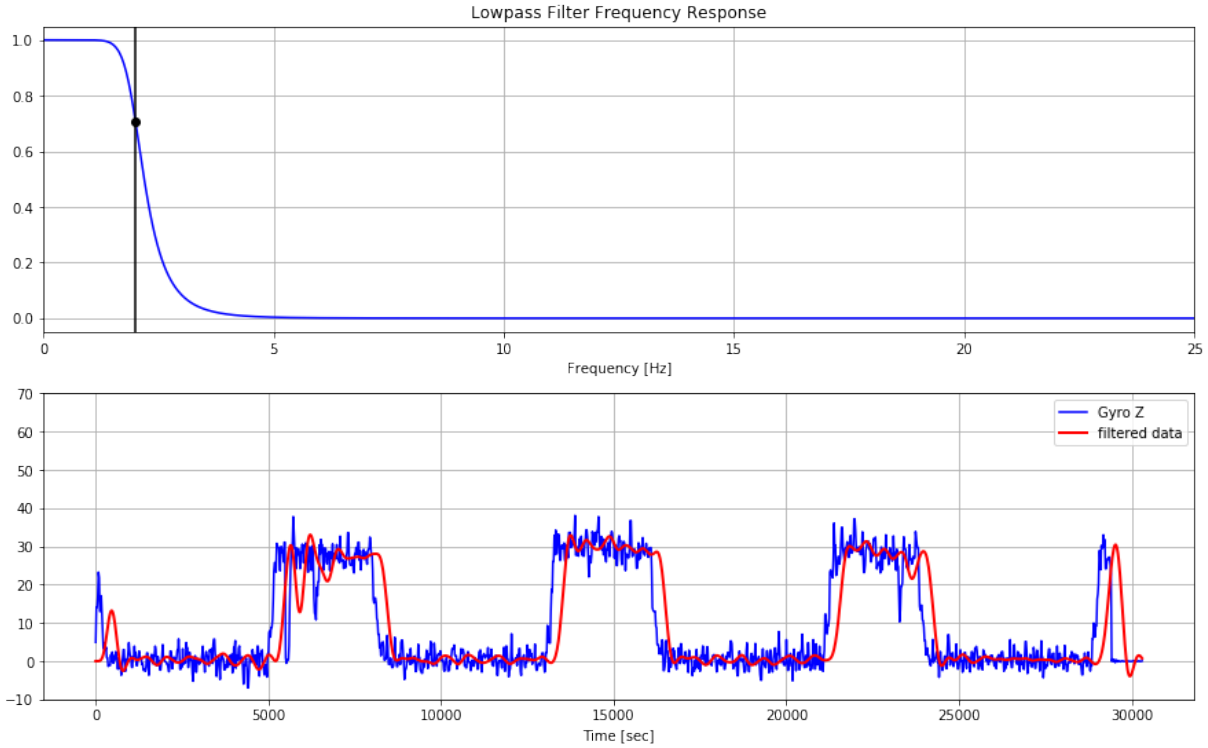


Figure 14: Filtering of IMU data using a Butterworth lowpass filter

Compared to the angular velocity, the forward velocity cannot be taken directly from the measurements, as the IMU measures translational acceleration. To obtain the velocities, the measurements thus first need to be numerically integrated (see Appendix [D]). Unfortunately, in addition to the regular problems with noise, the noise in the IMU measurements was skewed to the upside. This led to the errors being added up during integration, rather than positive and negative noise approximately cancelling out, which

finally resulted in improper representation of the translational velocities.
Furthermore, the orientation of the IMU relative to the PushBot had to be taken into account.

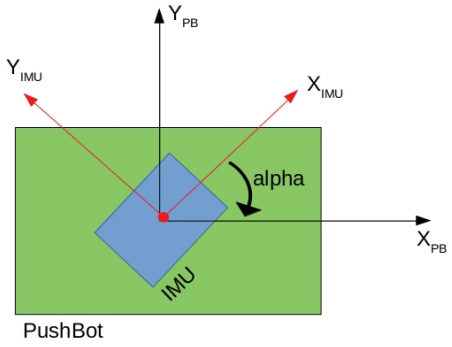


Figure 15: Angle correction for the IMU to compute forward velocity

Figure 15 shows a sketch of the PusBot where the IMU's axes are rotated by an angle alpha relative to the PushBot. In our case, unfortunately, alpha is not given. This is not a problem though since we know that $\dot{y}_{PB} = 0$ because the PushBot cannot have a velocity component perpendicular to the direction of its unturnable tracks. This means that $\vec{\dot{y}}_{IMU} + \vec{\dot{x}}_{IMU} = \vec{\dot{x}}_{PB}$. Additionally, since we are never moving backwards, we only need to determine the absolute value of the forward velocity $\dot{x_{PB}}$, which is given by

$$|\dot{x}_{PB}| = \sqrt{\dot{x}_{IMU}^2 + \dot{y}_{IMU}^2} \tag{14}$$

It should be explicitly noted that the disregarding of the sign for the forward velocity doesn't increase the effects of the aforementioned problems with the IMU noise. This is due to the fact that the computation of (14) is performed only after the integration of the IMU measurements $\ddot{x}_{IMU}$ and $\ddot{y}_{IMU}$.

Below are all the plots of the ideal and measured data of the four trajectories taken by the PushBot.

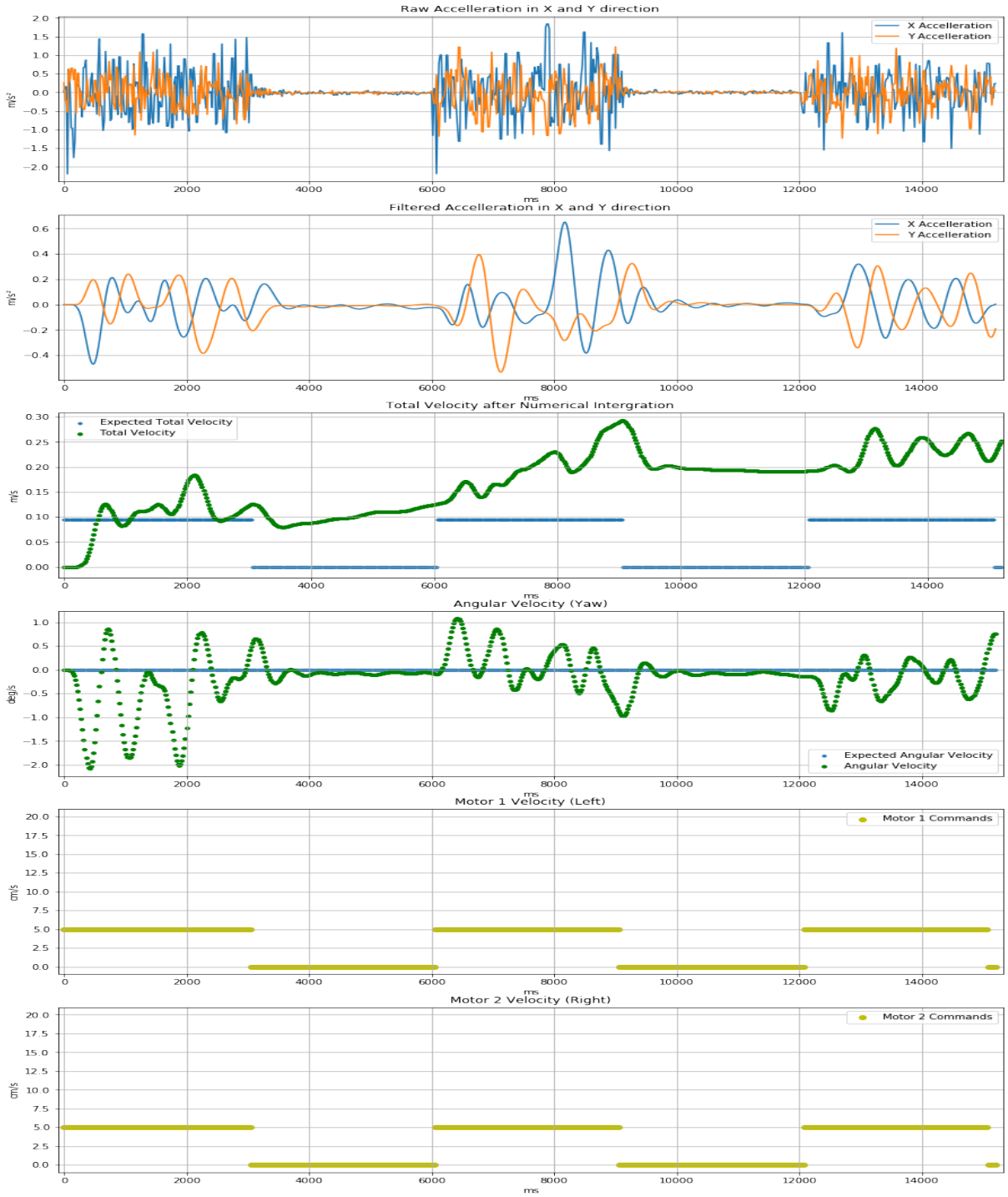


Figure 16: "Stop and Go Route" - Measured and Ideal Values

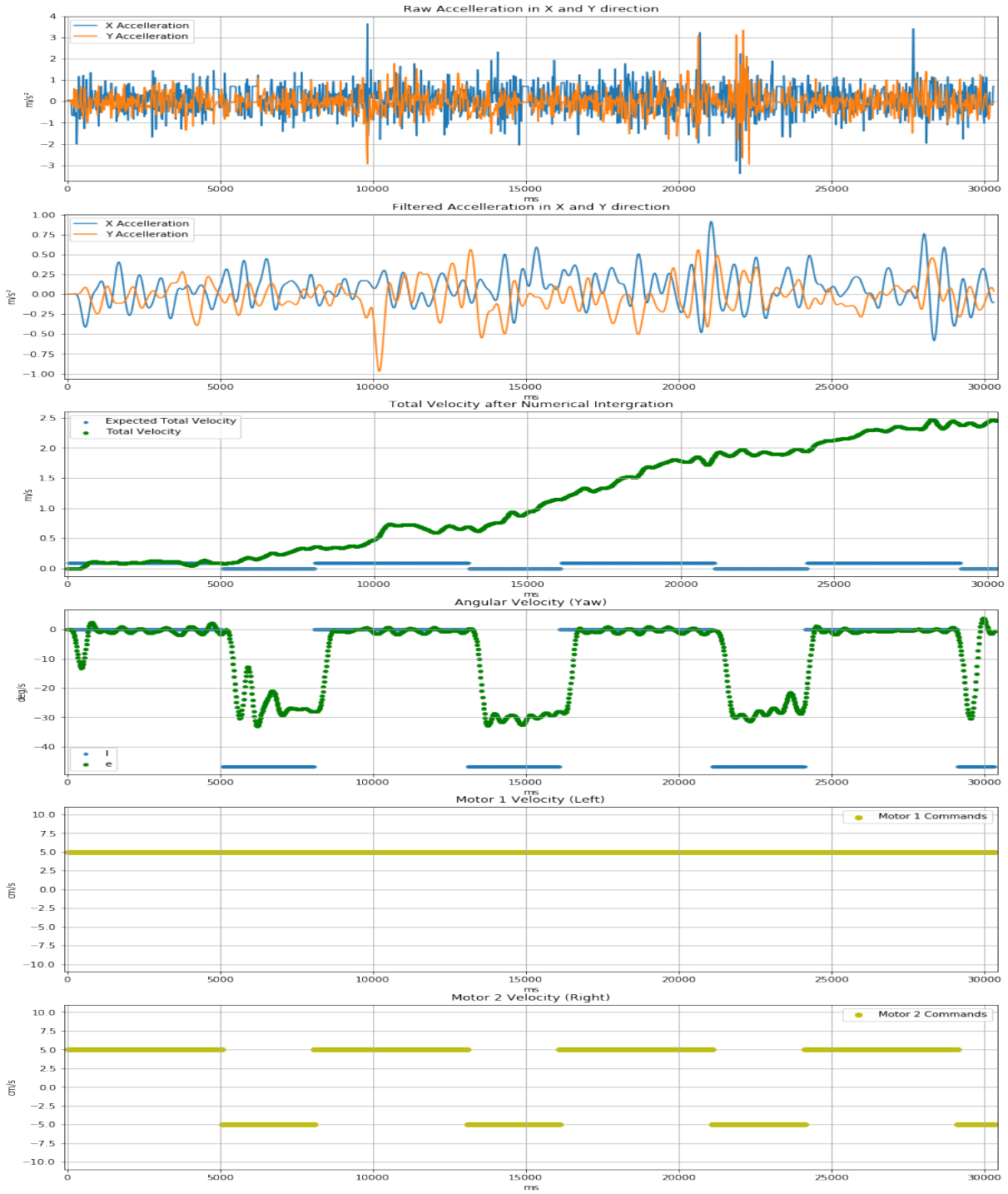


Figure 17: "Square" Route - Measured and Ideal Values

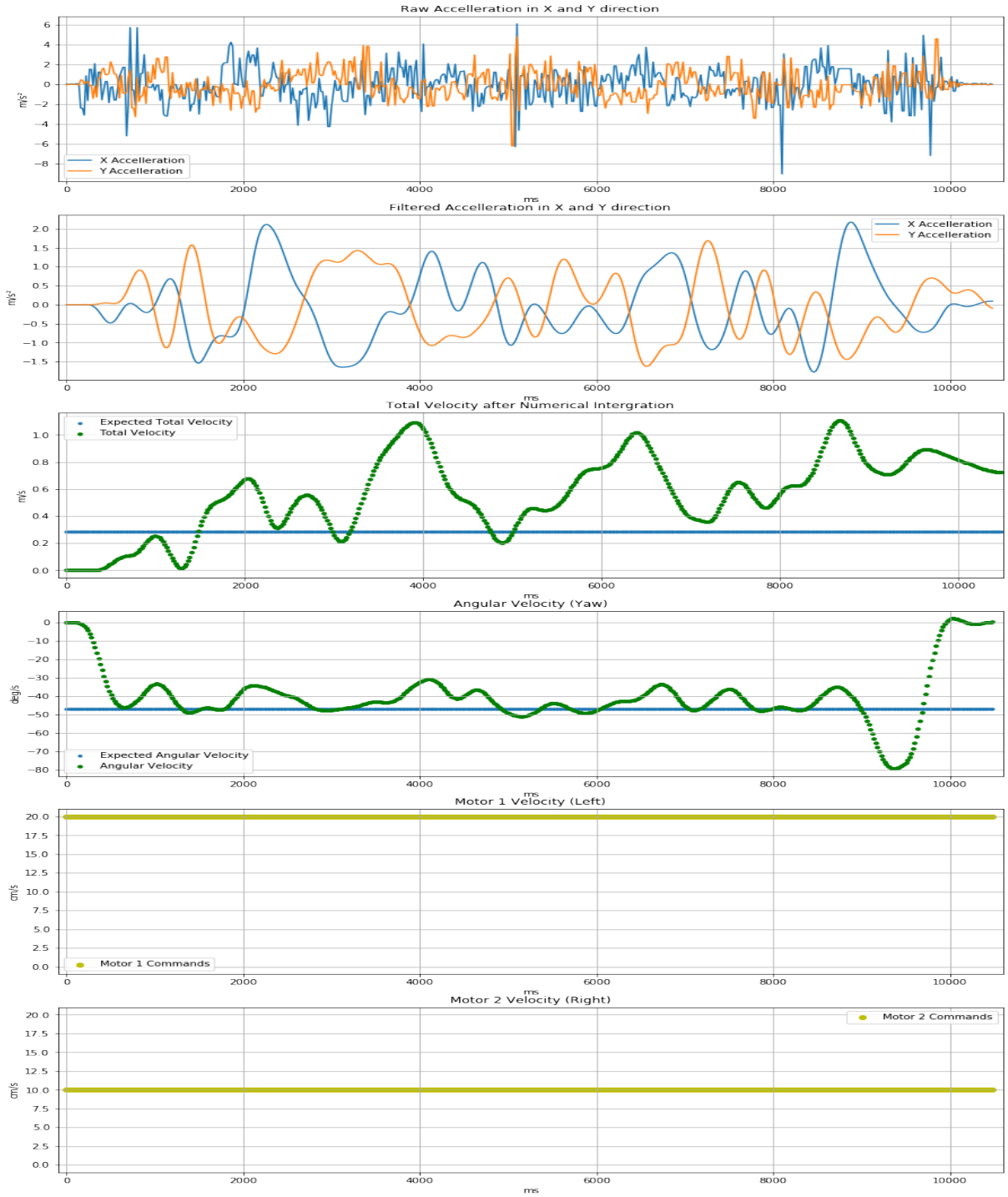


Figure 18: "Circle" Route - Measured and Ideal Values

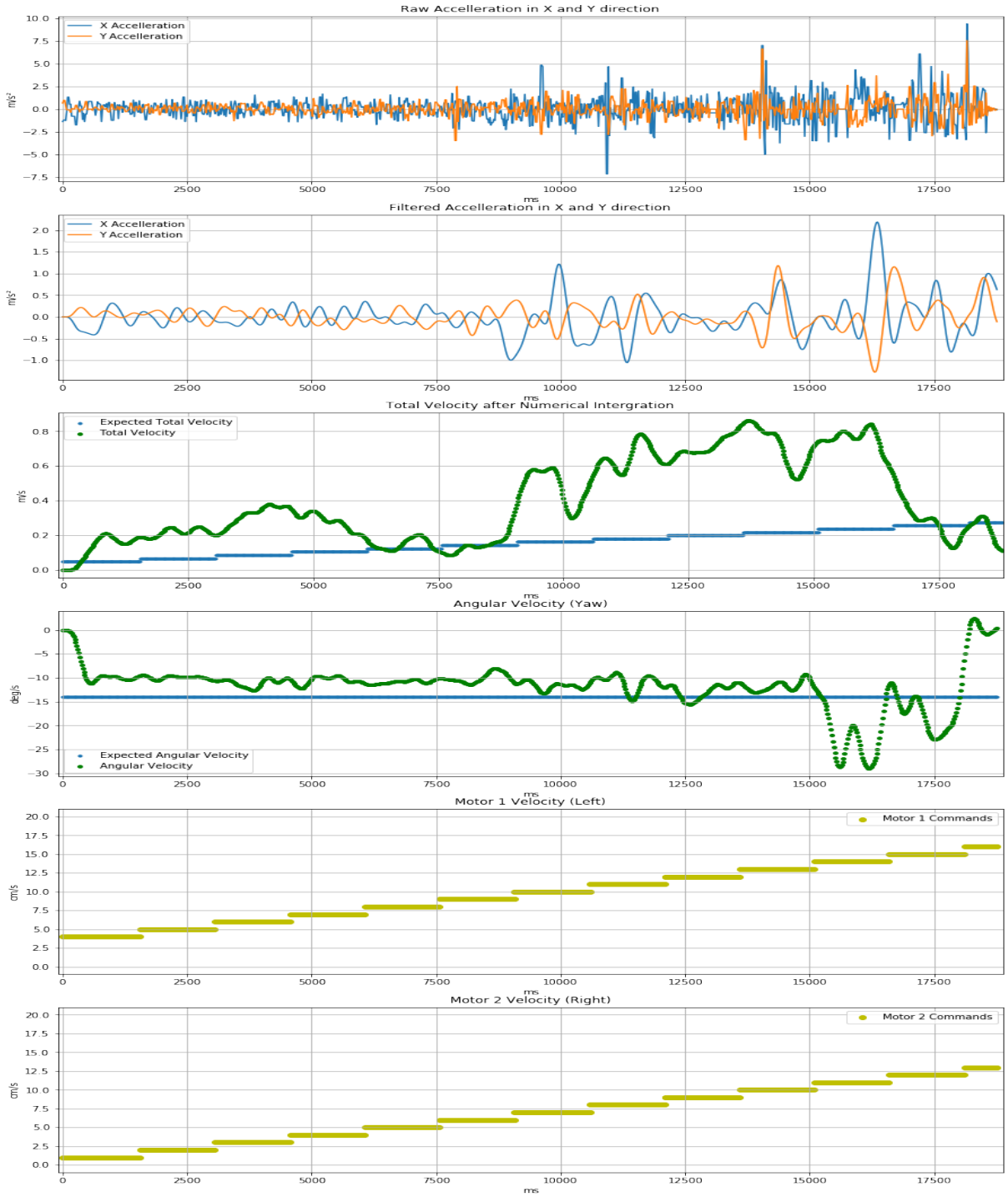


Figure 19: "Spiral" Route - Measured and Ideal Values

6.1.3 Prior Error in Data Measurement

As can be seen in the Figures above and below, the IMU data does not match the expected values at all. There are multiple reasons for this behaviour:
Firstly, the numerical integration of the accelerometer data leads to further mistakes being added up, as already mentioned. Secondly, because the PushBot moves at a cm/s rate, noise in the IMU becomes more significant. And last but not least, the latency between motor commands and actual movement of the PushBot can lead to further inaccuracies.

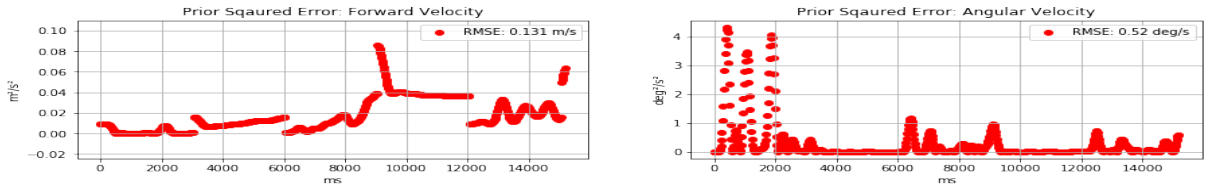


Figure 20: "Stop and Go Route" - Prior Error of Measurements.

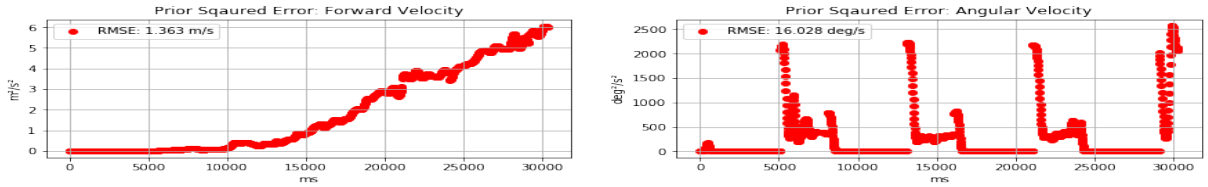


Figure 21: "Square" Route - Prior Error of Measurements.

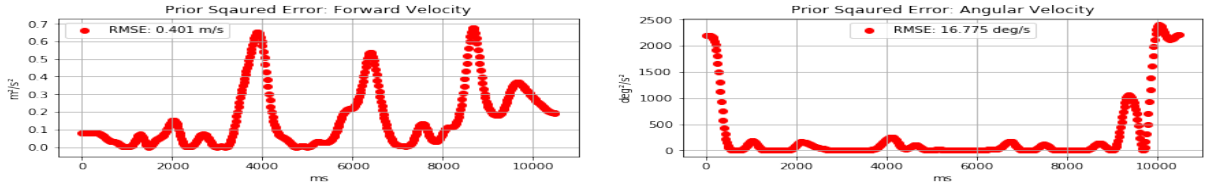


Figure 22: "Circle" Route - Prior Error of Measurements.

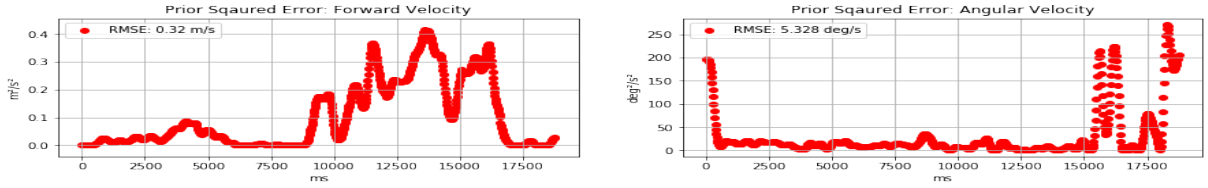


Figure 23: "Spiral" Route - Prior Error of Measurements.

6.2 SNN Architecture

In section 5.1, the general approach and structure of the network have been outlined. The exact parameters used to train the network on the measured data and obtain the following plots are:

- Learning Rule: $\Delta w = 2^{-4} * (x1 - 1) * (127 - w) * y0 - 2^{-2} * x0$
- Network Dimensions (Input and Output): 4 x 4
- weight$_{max}$: 150
- weight$_{min}$; $-0.9 * weight_{max}$ = -135
- Standard Deviation of 2D Gaussian synapse connection weight kernel: $1I$
- Self Excitation: None

6.3 Results

We used the ideal and recorded data to train the network described above on the forward and inverse mappings between IMU data and motor commands. In this section, however, we will only look at the inverse mapping, as this relation would be more important for a feedforward robot controller. The plots for the forward mappings can be found in the Appendix [E].
Training and testing were conducted in the same way as in section 4.2.2: In the training phase, we injected spikes based on the binned IMU data into the input WTA and spikes based on the binned motor commands into the output WTA. After a break of 1000 timesteps, we started the testing phase, where the same spikes were only injected in the input WTA. We then observed how well the resulting spikes in the output WTA matched the spikes previously injected into the output WTA in the training phase.
The graphs below show the input and output firing rates and population vectors. Furthermore, in the testing phase the expected output population vector is plotted next to the actual one and last but not least a plot of the error, i.e. the Euclidean distance between the two-dimensional expected and actual population vectors, is provided.

6.3.1 Ideal Data

As input for the spiking neural network in this section we used the ideal input data generated with the equations 10 and 11. This models the case where the measured data from the IMU as well as the integration for forwarding velocity are perfect. Additionally, these equations do not account for time delays between sent motor commands and the PushBots reaction.

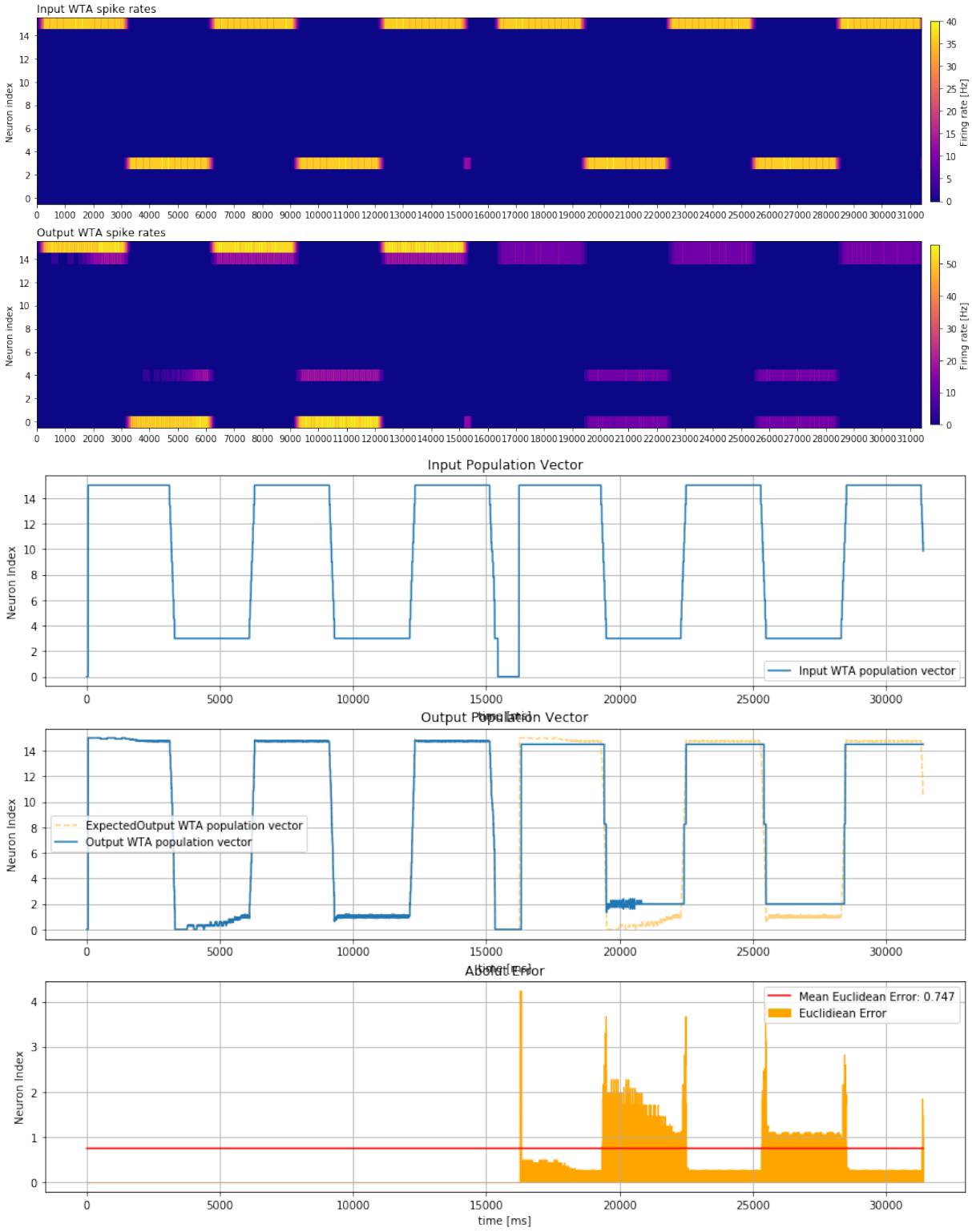


Figure 24: "Stop and Go" Trajectory - Spiking Neural Network Results (ideal data)

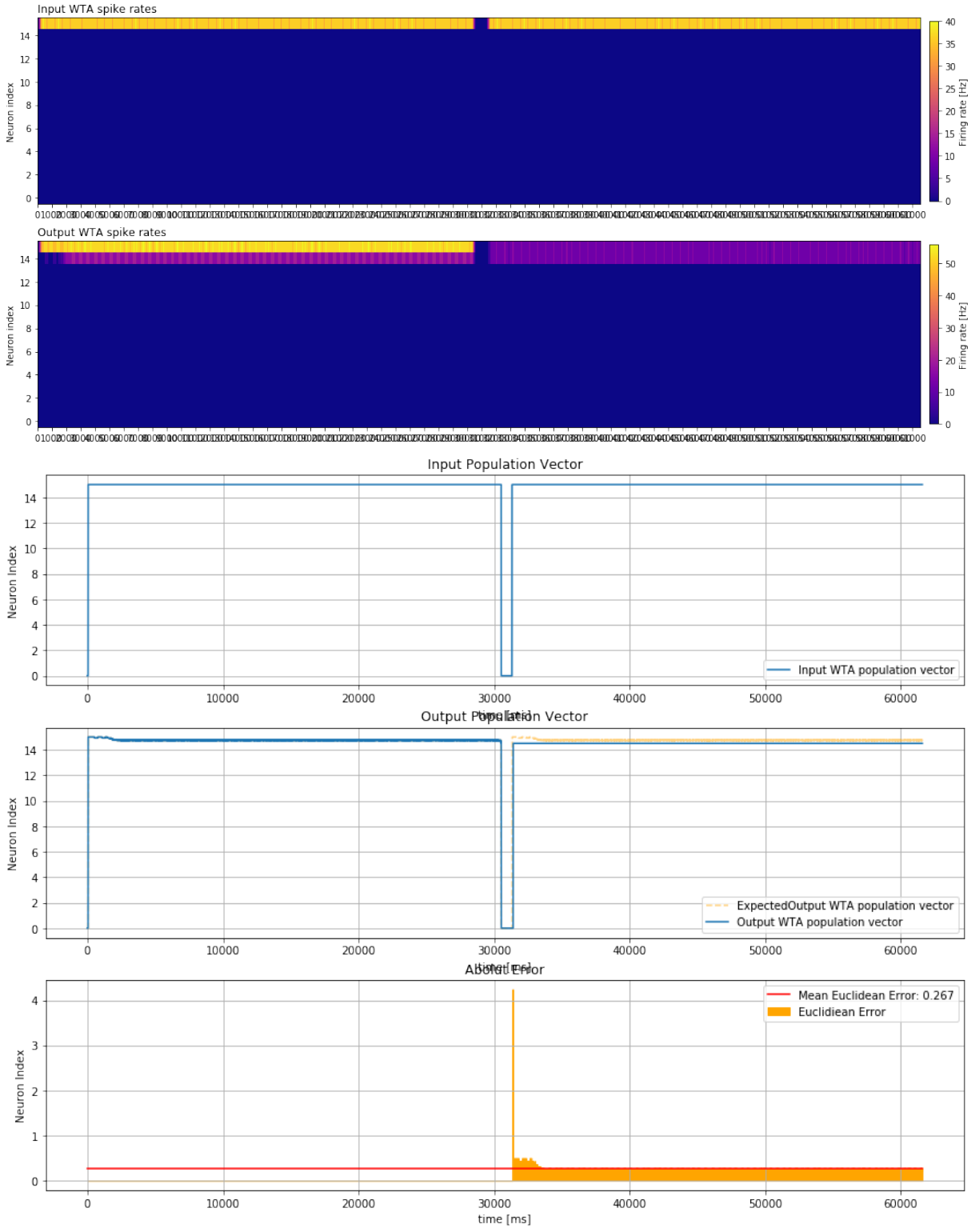


Figure 25: "Square" Trajectory - Spiking Neural Network Results (ideal data)

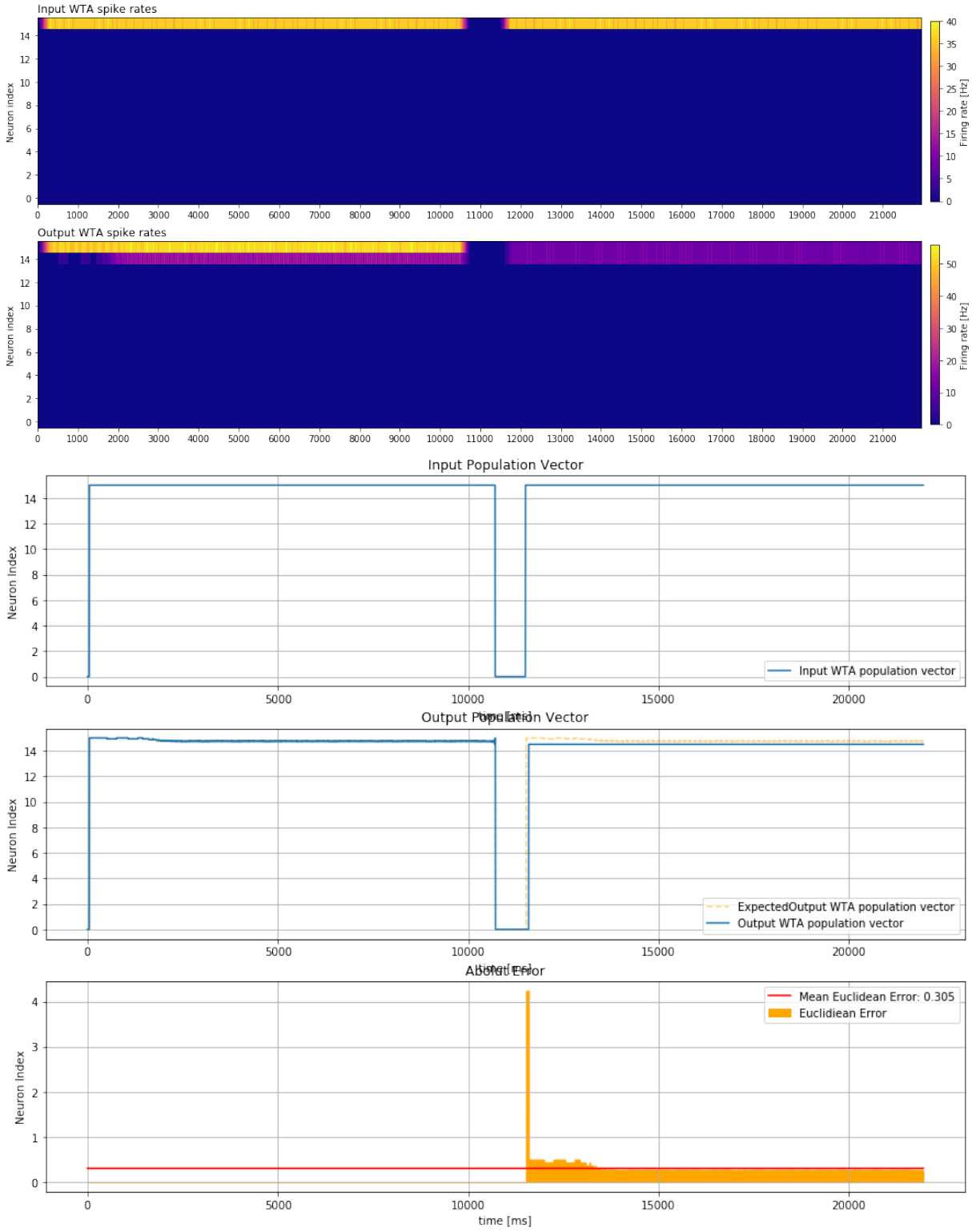


Figure 26: "Circle" Trajectory - Spiking Neural Network Results (ideal data)

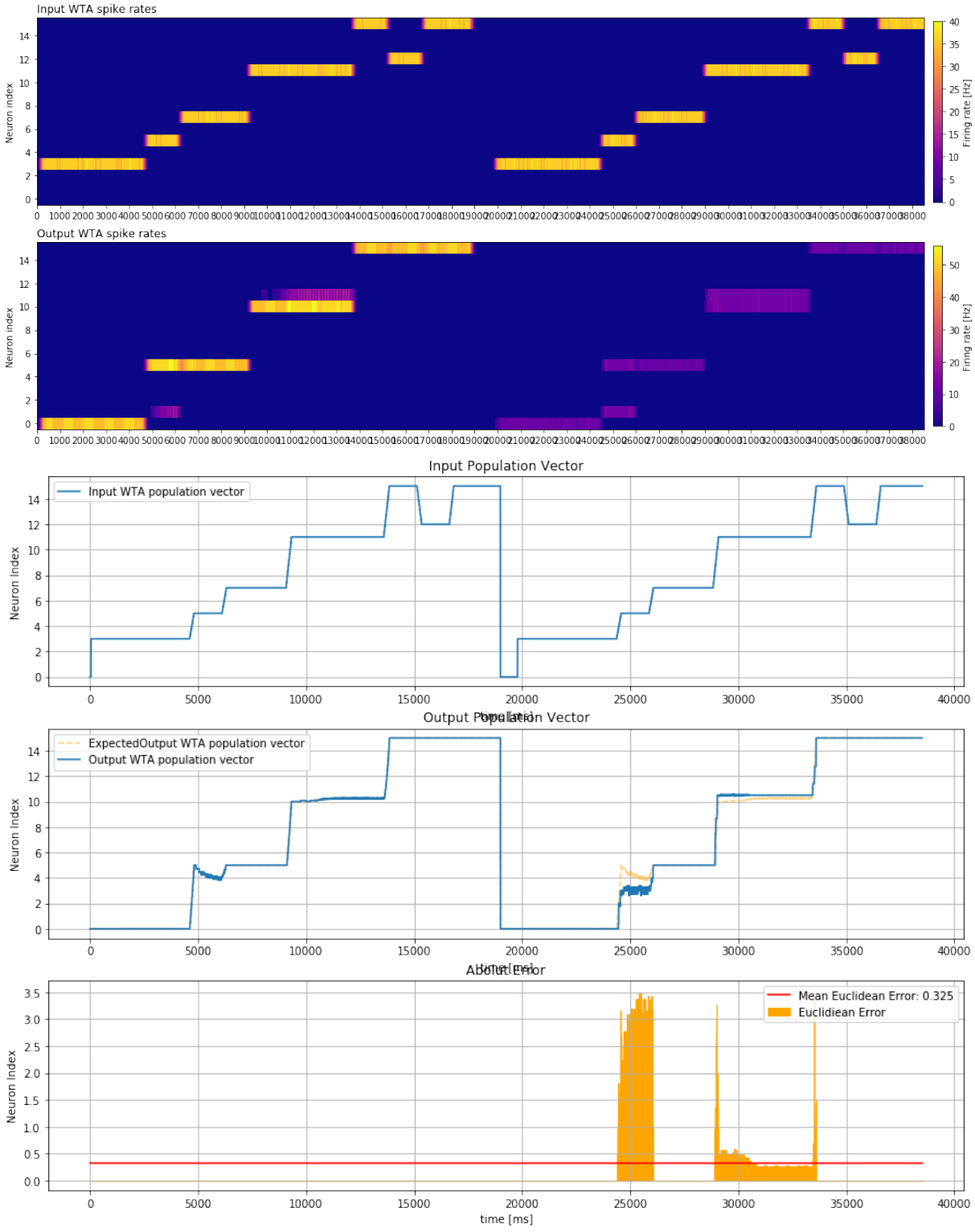


Figure 27: "Spiral" Trajectory - Spiking Neural Network Results (ideal data)

6.3.2 Measured Data

The following graphs have been generated using real IMU data collected from the PushBot as the inputs and outputs of the spiking neural network. This is a much more realistic approach, as the real world is seldom perfect.

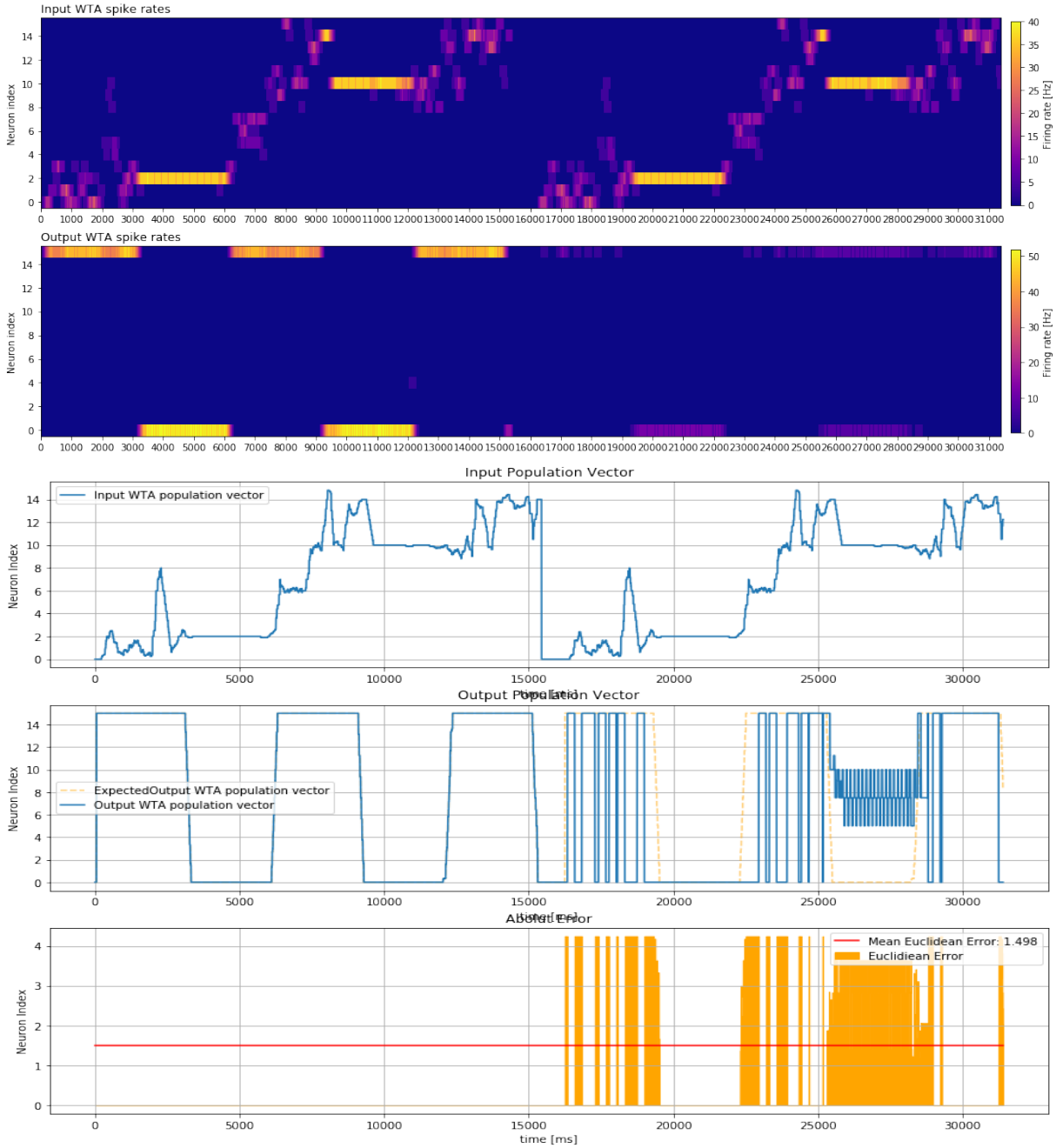


Figure 28: "Stop and Go" Trajectory - Spiking Neural Network Results (recorded data)

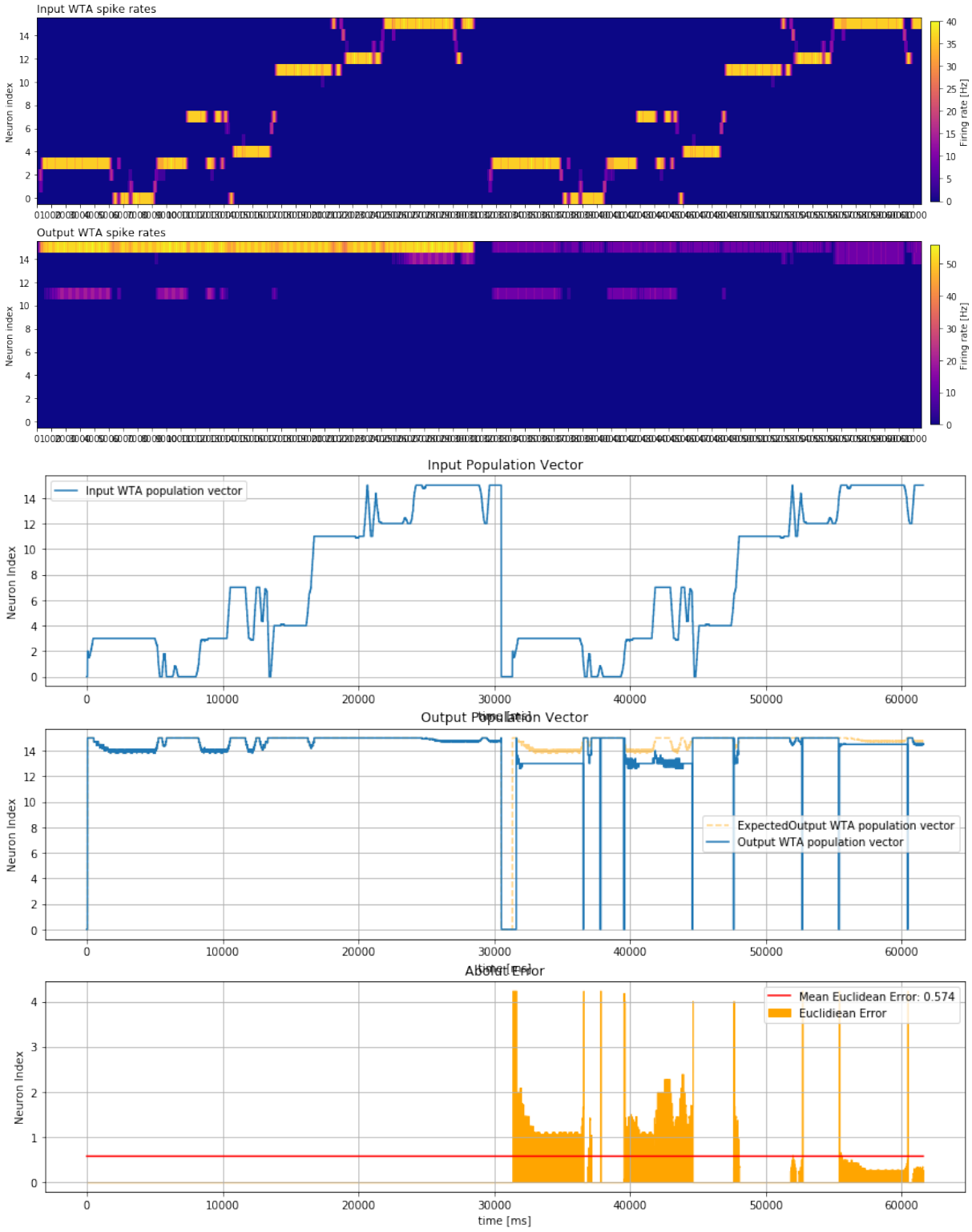


Figure 29: ”Square” Trajectory - Spiking Neural Network Results (recorded data)

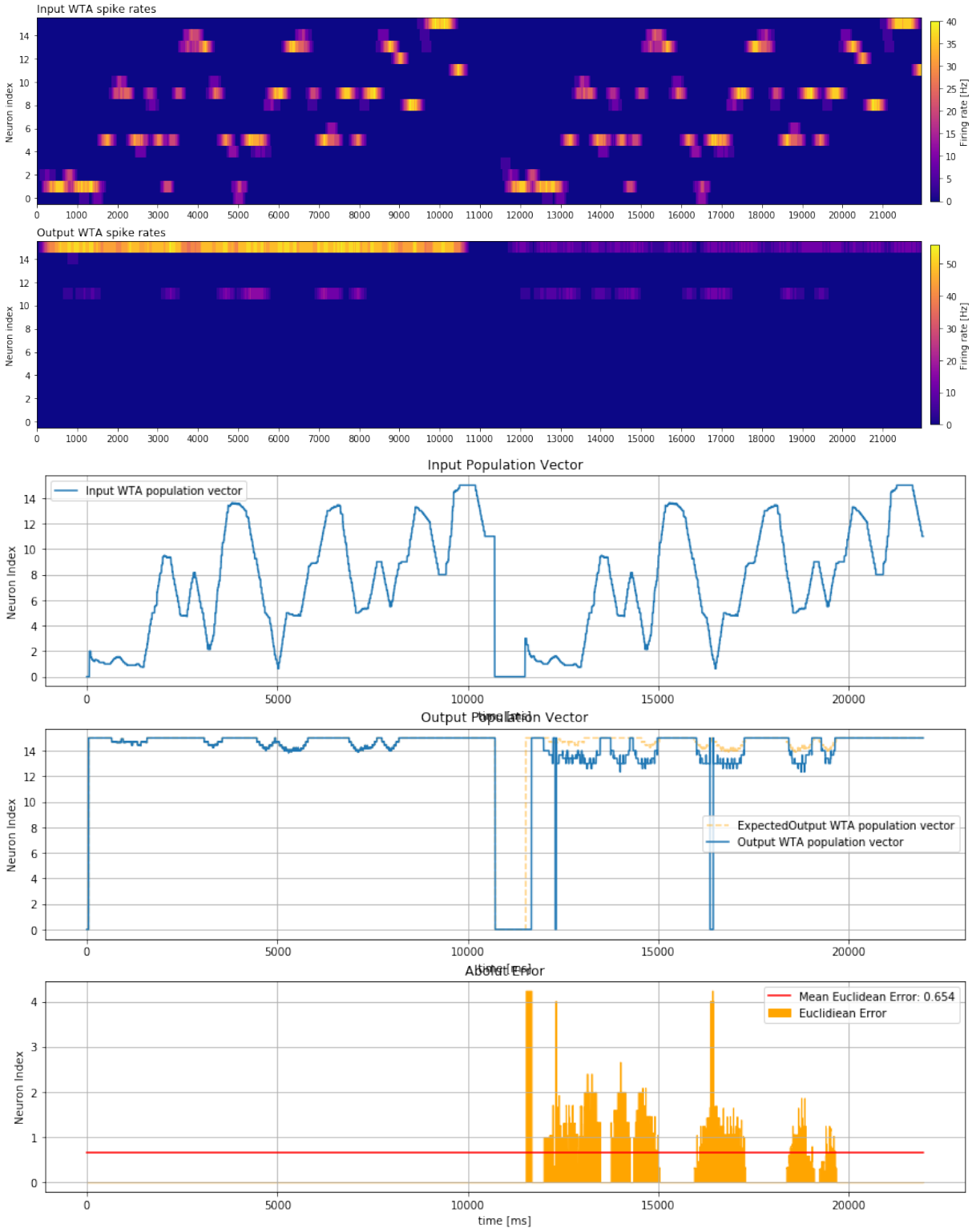


Figure 30: "Circle" Trajectory - Spiking Neural Network Results (recorded data)

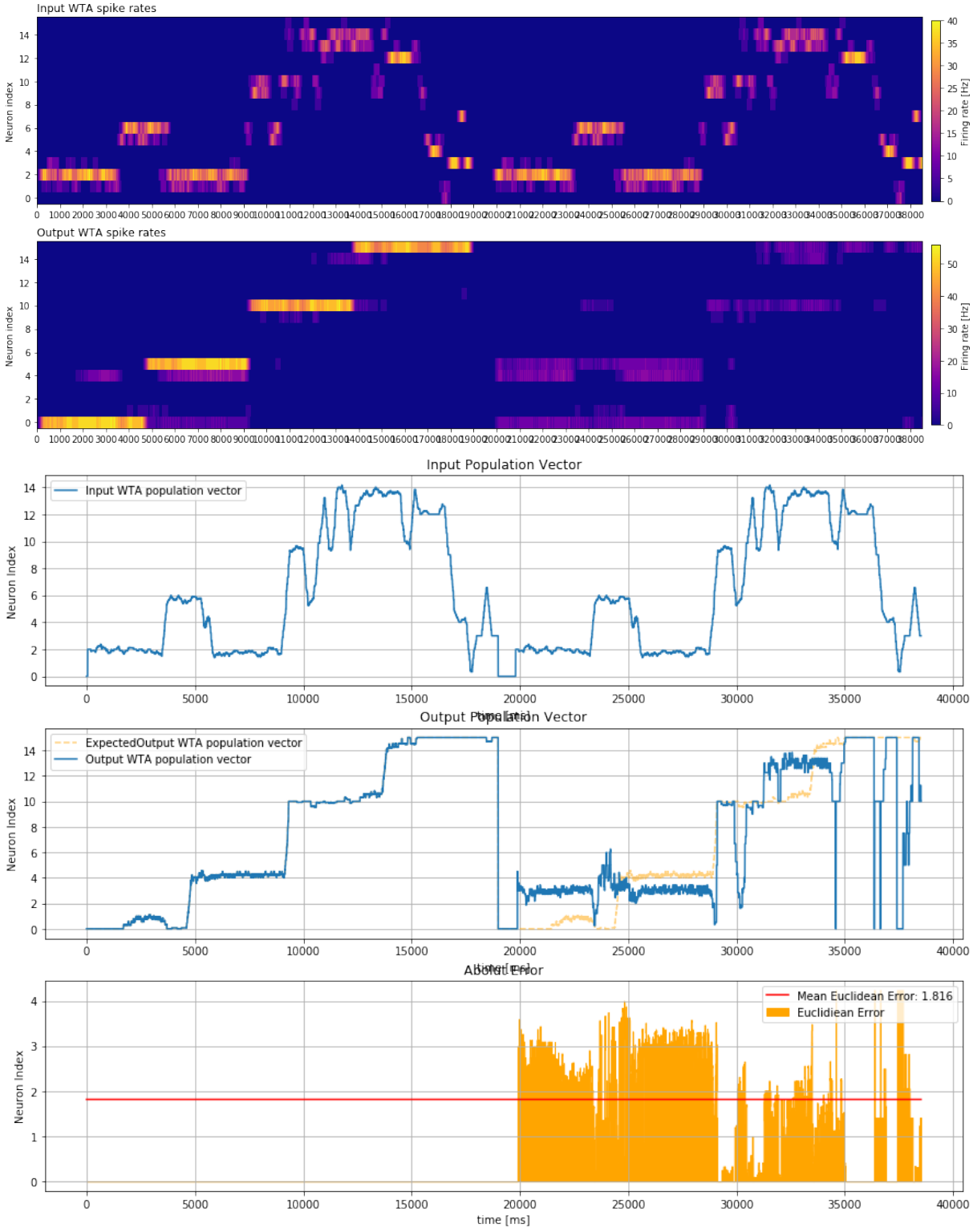


Figure 31: "Spiral" Trajectory - Spiking Neural Network Results (recorded data)

6.4 Technical Problems

There were various technical problems that occurred during the conduction of the experiments, which could lead to corrupted or inaccurate results.

6.4.1 Noisy Data

The IMU data was very inaccurate, especially the accelerometer measurements. Due to the slightly positive bias of the accelerometer noise, the translational velocity values are continuously increasing, as errors are being added up in the numerical integration.
A possible remedy would be to calculate forward and angular velocity, by using the data of the wheel encoders. This needs to be taken with care as well, though, as rolling wheels might not necessarily mean that the robot is moving, such as when grinding against an obstacle.

6.4.2 IMU Data Latency

Initially, the data that was being collected from the IMU did not make much sense, and we couldn't explain why we weren't seeing anything close to the expected values. We first thought that we could stop data recording once we saw that the PushBot had finished moving, which also matched the recorded timestamps. After several trials, though, we figured out that the data sent from the PushBot to the local computer had a relatively large delay. Thus, instead of terminating the scripts immediately after the movements were completed, we continued recording the data for an additional minute or so. With this approach, we were then also able to clearly see in the recordings when the PushBot had stopped moving. Consequently, we were able to cut out the insignificant data at the beginning and end of the measurements.

6.4.3 Loihi Training

Training a spiking neural network on the Loihi device can be done fairly quickly. Nonetheless, if one wishes to obtain information about the network, especially if a graphical representation of e.g. the spikes are needed, probing becomes necessary, which will increase the execution time substantially. For instance, a simple two-dimensional spiking neural network consisting of 5 by 5 neurons can be trained in barely a second; however, even if only spike probing is turned on, the whole procedure will take up to approximately 3 minutes.
Subsequently, because of the long training time, parameter optimization could only be performed in a very limited manner, leading to non-optimal network predictions after training.
Additionally, because of the limited number of Loihi chips and multiple groups using it, the accessibility for the training of our spiking neural networks was decreased.

7 Conclusion

7.1 Discussion

In the first part of the project, we expanded the WTA architectures built by [3] into two dimensions and tested them with respect to their abilities to filter incoming spikes and also to approximate the identity function. We were able to achieve great selectivity and promising learning abilities with 4×4 networks.
Subsequently, we recorded and processed IMU data from the PushBot to feed through our newly built architecture in order to test it on real-world data. We wanted to examine if the network is able to learn the forward and inverse mappings from IMU data to motor commands for four example routes.
In general, it can be observed that the error is higher for the recorded data than for the ideal data, which is to be expected. Furthermore, the resulting plots show that not all trajectories performed equally well. This will be a problem if one ultimately wishes to use the same network architecture to learn all possible input and output combinations, not only four separate subsets of them. These issues may have various causes.
One factor could be that the teaching signals might not be constant enough, which could lead to the weights not adjusting quickly enough during learning. This would be supported by the fact that trajectories with a higher movement of peaks among the two-dimensional neural network performed rather poorly. An easy shortcut to overcome this problem would be to use multiple training passes. (We have also implemented the possibility to do this, see Appendix [B].)
As a whole, it should also be added that the choice of routes may not have been optimal. This is especially evident for the extreme case of the circle route: As both the forward and the angular velocities of the PushBot stay constant in a circular movement, only one possible combination of motor commands and IMU measurements is provided, which leads to only one neuronal peak appearing in the input and output WTAs respectively.
Furthermore, if only one or two input or output combinations are provided, a problem with binning arises: The data is being binned into neurons on the edges of the network, which are prone to distortions due to asymmetries caused by the cutoff of the Gaussian connection weights.
Another factor can be seen when regarding the square route: We can observe that the output WTA spikes look almost identical to the spikes of the circle route, i.e. only one peak occurs. This shouldn't be the case, as there are clearly two distinct combinations of motor commands provided (see Figure 29). This problem can be attributed to the global inhibition in the WTA that doesn't allow for the second combination of motor commands that would result from the right turn of the PushBot (see Figure 11) to override the previous peak from the linear movement. A decrease in the maximum inhibitory connection weights might help alleviate this problem.
In conclusion, there is still a certain amount of finetuning to be done to the current spiking neural network in order for it to be able to handle measured data of moving robots; especially for future online learning.

7.2 Outlook

There are various future projects that could be done building upon our work in order to reach the final goal of a fully autonomous robot controlled by an SNN. Here are some suggestions on our behalf:

- Improving robustness of the robot's data collection algorithm. This can be done, by using the wheel encoders instead of the IMU.
- Switching the input and/or output variables of the spiking neural network and observe the resulting performance of the network. Instead of using angular and forward velocity, as well as left and right motor commands, different variables could be used, e.g. acceleration, distance ρ from a centre point M for curves (see sketches in 5.2), etc.
- Compare the existing two-dimensional architecture with two one-dimensional SNNs on their performance (see section 8.1 below).
- Scale the size of the SNN to three dimensions, such that it could be used for a robot with a higher degree of freedom.
- Finally, implement online learning, such that the spiking neural network is connected with the robot interface and can learn directly from the robot's movements in a closed loop (as in [5]).

8 Reflection on the group project

8.1 About the topic

Spiking Neural Networks and neuromorphic hardware are promising to become a very important class of control architectures in the future of embedded systems and robotics. However, as of yet, the implementation and computation of SNNs for multidimensional problems is not as user- friendly as machine learning frameworks for artificial neural networks, such as Theano, Caffe and TensorFlow.
This project only tested a supervised learning approach, which is all right to test the abilities of a Spiking Neural Network with respect to real-world data. Nonetheless, for realizing the vision of fully autonomous drones, online learning, as well as unsupervised learning, should especially be considered, such as in [5].
Finally, it should be added that a multidimensional network approach might not be the most efficient solution in terms of scalability. This is due to the fact that the number of neurons scales with $\mathcal{O}(n^m)$ where m is the dimension and n the number of neurons per dimension. It's even worse for the number of synapses between two multidimensional layers, which scales with $\mathcal{O}(n^{2m})$. Thus if a robot has more than the two dimensions of the PushBot, we will quickly run into problems.
A more efficient approach would be to consider a 1-dimensional layer for each of the inputs and a one-dimensional layer for each of the outputs, with a bag of neurons for learning in between.

8.2 Lessons learned

Throughout the project, ideas and approaches turned out not to work the way we expected them to. For example, initially, we wanted to use the OmniBot for our experiments. However, the complexity of the higher dimensionality, as well as the fact that the PushBot already had a much more advanced data collection interface, implemented led us to switch to the latter. We thus learned that one must always be prepared to adapt to unexpected circumstances.
Another aspect that stood out throughout the project is the fact that one should never underestimate so-called meta-work[4], since we had multiple problems especially with the VPN, SSH and Kapoho bay itself[5].
In addition, to avoid some of the encountered problems, further literature research, as well as more research on previous projects that have experimented with the same hard- and software, could be done more thoroughly in future projects.

[4]Meta-work describes activities that do not directly contribute to the progress of the actual work (project) but are still necessary in order to enable more actual work, such as installing a certainly needed program.

[5]A major problem is that when multiple people try to access Kapoho at the same time, in particular, if something is already running on the Loihi chip and another thread tries to access it as well (with the net.run() command), the chip dies and the whole computer needs to be restarted. A possible fix would be to implement a queue. This could maybe also be listed as a project at INI.

Appendices

A Multivariate Gaussian Weight Connection Matrix

```
import numpy as np

def create_gaussian_weight_matrix_2d(size,
                                     sigma,
                                     normalized=False):
    '''
    Creates a gaussian 2d weight matrix
    Args: size (tuple, list), dimension of the neuron layer
          sigma (int, tuple), standart devieation of variables
          normalized (bool), normalized or not
    Returns: w_matrix (np.matrix), connection weights for ROW-MAJOR representation of 2D neuron layer
    '''

    size_x = size[0]
    size_y = size[1]

    # w matrix calculation using product of one variable gaussian distributions
    w_matrix = np.zeros((size_x*size_y, size_x*size_y))
    norm_x = np.exp(-(1/2)*(make_distance_mat(size_x)/sigma[0])**2)
    norm_y = np.exp(-(1/2)*(make_distance_mat(size_y)/sigma[1])**2)
    for i in range(size_x):
        for j in range(size_y):
            w_matrix[i*size_y+j] = np.reshape(norm_x[i] * norm_y[j],(1,-1),'C')

    return w_matrix

w_matrix = create_gaussian_weight_matrix_2d(size, sigma)
```

B Translation of IMU and motor command data to Spikes

```
# bin the data according to size of neurons
def bucket(vel,nbins):
    '''returns np array of ints with length len(vel) where each entry has been binned accordingly'''
    ledge = np.percentile(vel, 5)
    redge = np.percentile(vel, 95)
    incr = (redge - ledge)/nbins
    edges = [ledge + i*incr for i in range(1,nbins)]
    binned = np.zeros(len(vel), dtype = int)
    assigned = False
    for i in range(len(vel)):
        for j in range(nbins-1):
            if vel[i] < edges[j]:
                binned[i] = j
                assigned = True
                break
            elif assigned == True:
                assigned = False
        if not assigned:
            binned[i] = nbins-1
    return binned

#INPUT and TEACHING SIGNAL parameters
#Firing rate (in herz, at least 50)
firing_rate_input =  50#in hertz
firing_rate_teach = 50
firing_rate_test = 50

#INPUT SPIKES PARAMS
thresh_input = 0
rate_scale_input = firing_rate_input/50 #input rate of measurements is approx. 50 hertz
int_scale_input = int(rate_scale_input)
add_scale_input = rate_scale_input-int_scale_input
nodes_input = bucket(transvel, wta_dimensions[0])*wta_dimensions[1] + bucket(angvel, wta_dimensions[1])
spike_times_input = {}
#TEACHING SPIKES PARAMS
thresh_teach = 0
rate_scale_teach = firing_rate_teach/50
int_scale_teach = int(rate_scale_teach)
add_scale_teach = rate_scale_teach - int_scale_teach
nodes_teach = bucket(commands[:,1], wta_dimensions[0])*wta_dimensions[1] + bucket(commands[:,0], wta_dimensions[1])
spike_times_teach = {}

forwardmapping = False
if forwardmapping:
    temp = nodes_input
    nodes_input = nodes_teach
    nodes_teach = temp

for i in np.unique(nodes_input):
    spike_times_input[i]=[]
for i in np.unique(nodes_teach):
    spike_times_teach[i]=[]
```

```
start_time = 0
training_passes = 1
for p in range(training_passes):
    for i in range(len(timestamps)-1):
    #INPUT
        t_diff = timestamps[i+1]-timestamps[i]
        for j in range(int_scale_input):
            time = start_time + timestamps[i] + t_diff/int_scale_input*j
            spike_times_input[nodes_input[i]].append(int(time))
        thresh_input += add_scale_input
        if thresh_input >= 1:
            time = timestamps[i+1]-0.5/int_scale_input*t_diff
            spike_times_input[nodes_input[i]].append(int(time))
            thresh_input -= 1
    #TEACHING
        for j in range(int_scale_teach):
            time = start_time + timestamps[i] + t_diff/int_scale_teach*j
            spike_times_teach[nodes_teach[i]].append(int(time))
        thresh_teach += add_scale_teach
        if thresh_teach >= 1:
            time = timestamps[i+1]-0.5/int_scale_teach*t_diff
            spike_times_teach[nodes_teach[i]].append(int(time))
            thresh_teach -= 1
    start_time += timestamps[-1]

#ADD TEST SPIKES
thresh_test = 0
timestampstest = start_time + timestamps + 1000

for i in range(len(timestampstest)-1):
    t_diff = timestampstest[i+1]-timestampstest[i]
    for j in range(int_scale_input):
        time = timestampstest[i] + t_diff/int_scale_input*j
        spike_times_input[nodes_input[i]].append(int(time))
    thresh_test += add_scale_input
    if thresh_test >= 1:
        time = timestampstest[i+1]-0.5/int_scale_input*t_diff
        spike_times_input[nodes_input[i]].append(int(time))
        thresh_test -= 1

for node in spike_times_input:
    input_spike_gen.addSpikes(spikeInputPortNodeIds=node,
                              spikeTimes=spike_times_input[node])
for node in spike_times_teach:
    teaching_spike_gen.addSpikes(spikeInputPortNodeIds=node,
                              spikeTimes=spike_times_teach[node])
```

C Butterworth Low Pass Filter

```
import numpy as np
from scipy.signal import butter, lfilter

def butter_lowpass_filter(data, cutoff, fs, order=5):
    b, a = butter_lowpass(cutoff, fs, order=order)
    y = lfilter(b, a, data)
    return y

y = butter_lowpass_filter(data, cutoff, fs, order)
```

D Numerical Integration of Accelleration

```
import numpy as np

dt = 0.02

v_x = [0]
v_y = [0]

for i in range(1,acc_x.shape[0]):
    v_x.append(acc_x[i-1]*dt+v_x[i-1])
    v_y.append(acc_y[i-1]*dt+v_y[i-1])

velocity = (np.asarray(v_x)**2 + np.asarray(v_y)**2)**0.5
```

E Forward Mapping

E.1 Ideal Data

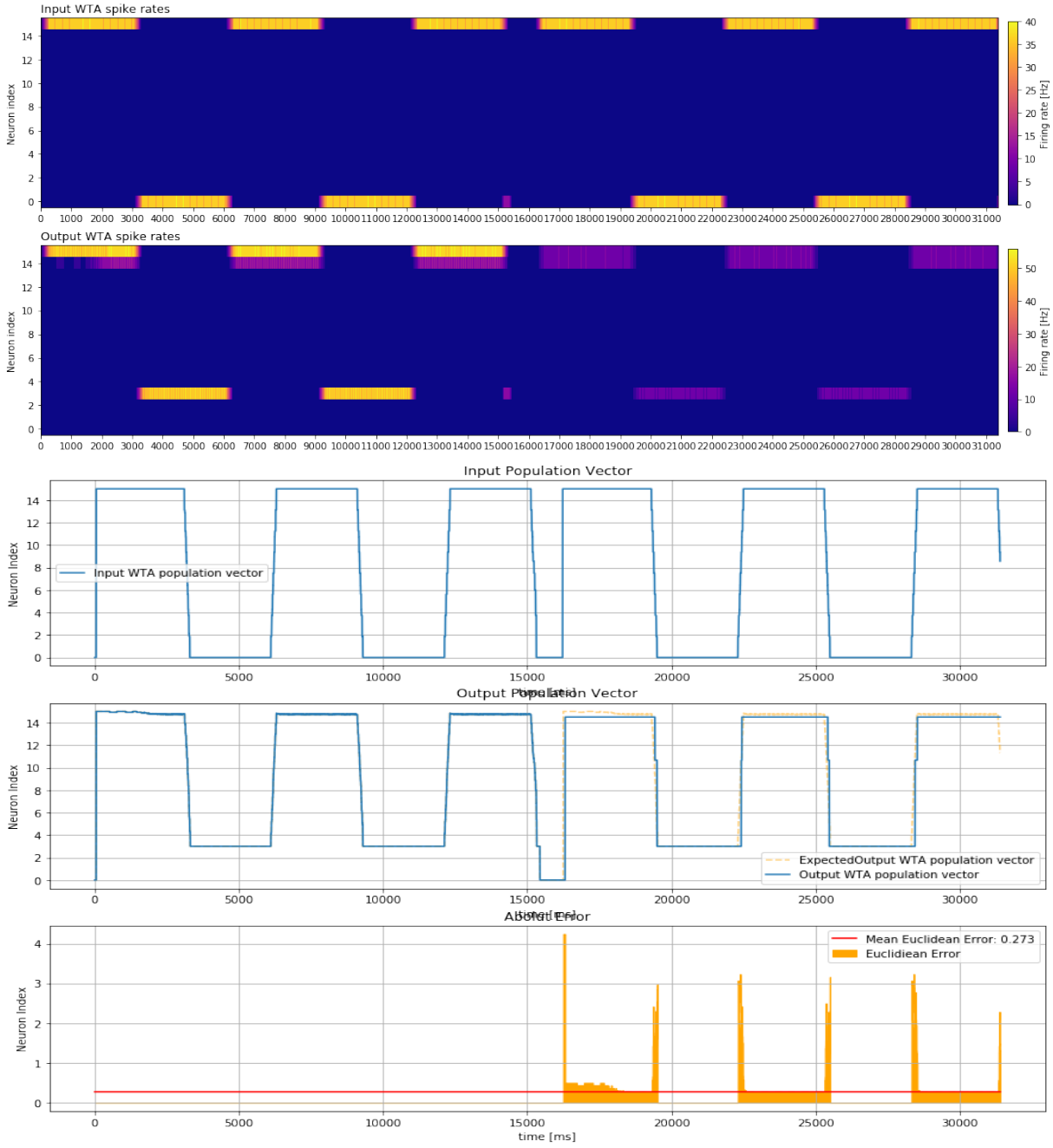


Figure 32: "Stop and Go" Trajectory Forward Mapping - Spiking Neural Network Results (Ideal data)

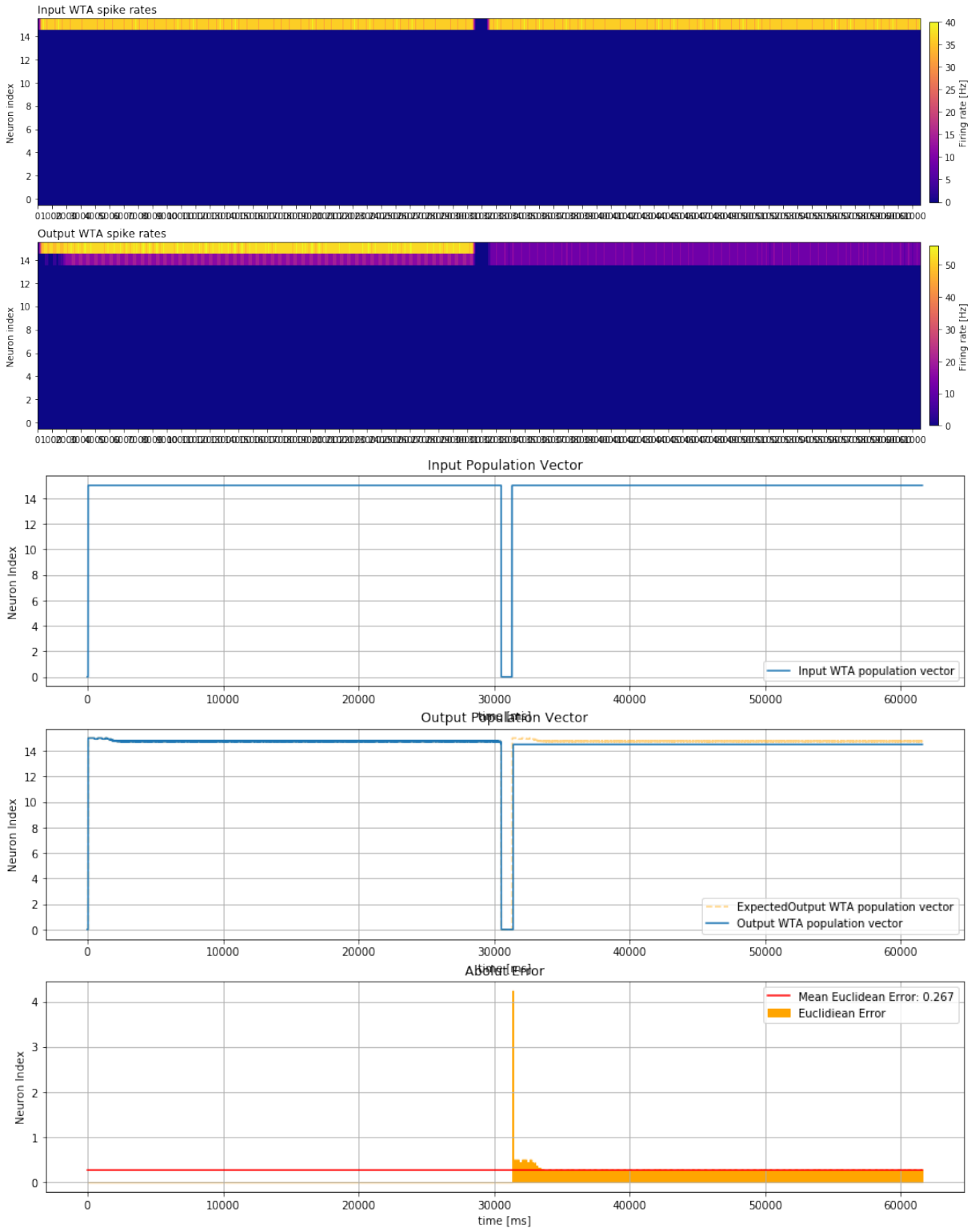


Figure 33: "Square" Trajectory Forward Mapping - Spiking Neural Network Results (Ideal data)

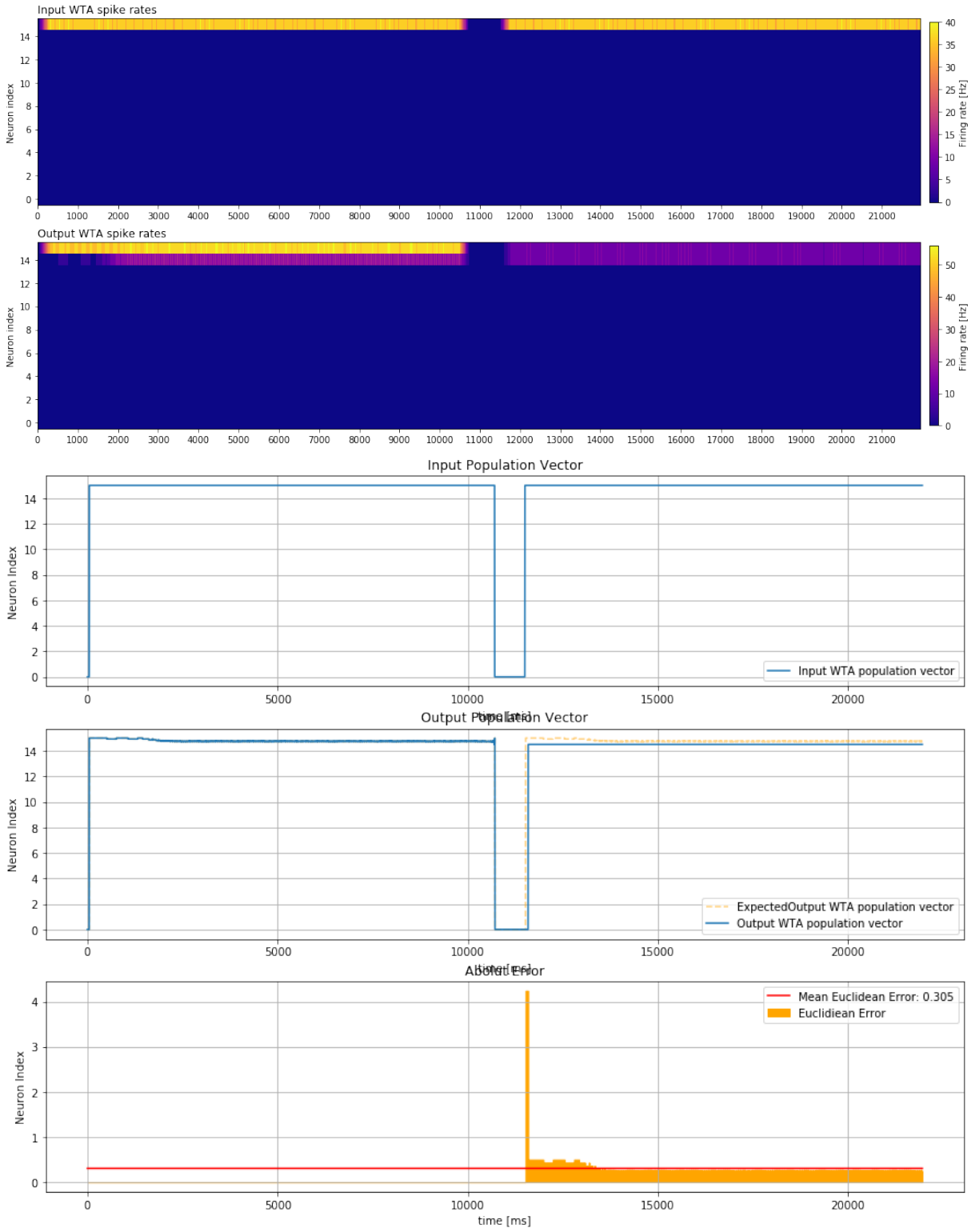


Figure 34: "Circle" Trajectory Forward Mapping - Spiking Neural Network Results (Ideal data)

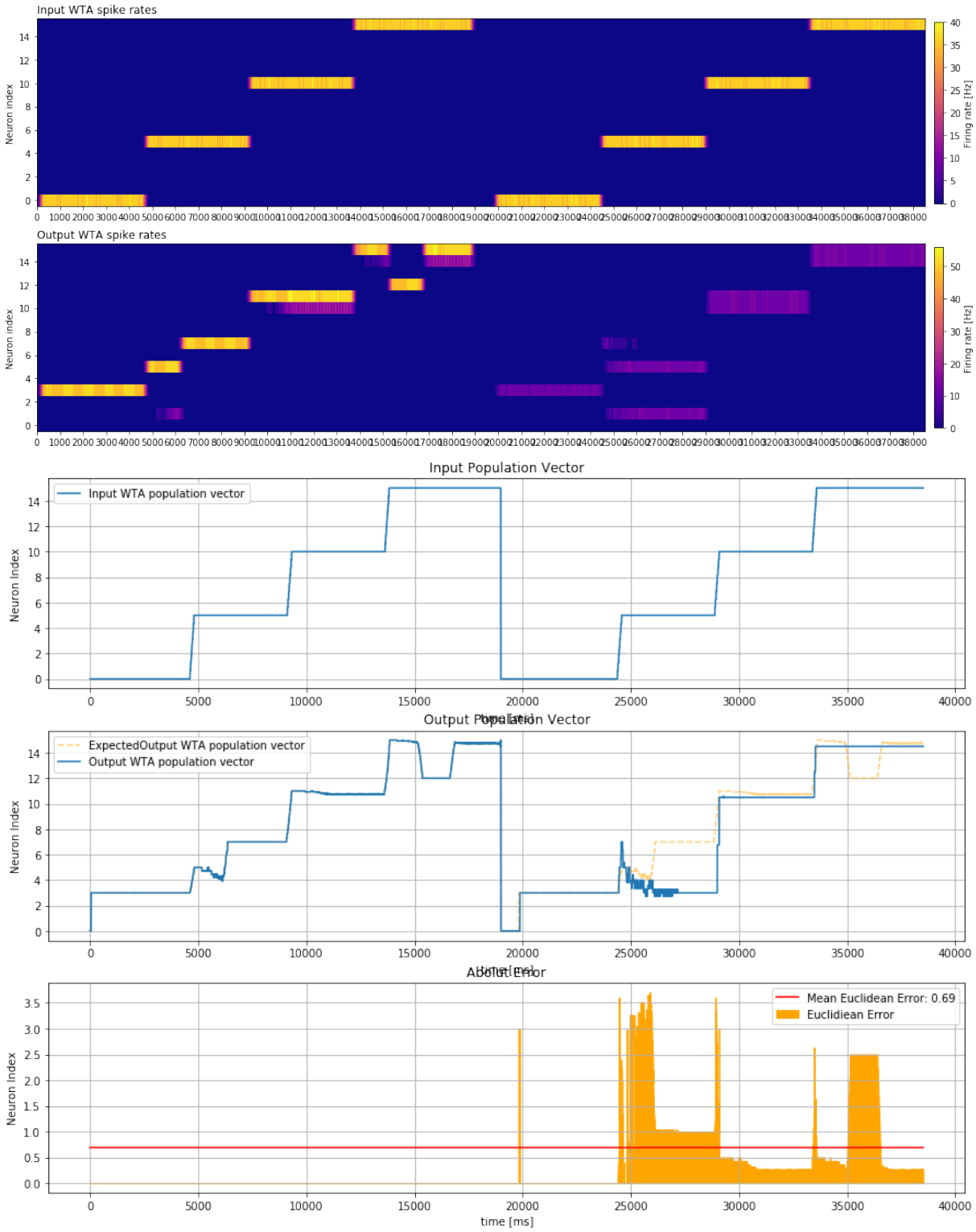


Figure 35: "Spiral" Trajectory Forward Mapping - Spiking Neural Network Results (Ideal data)

E.2 Measured Data

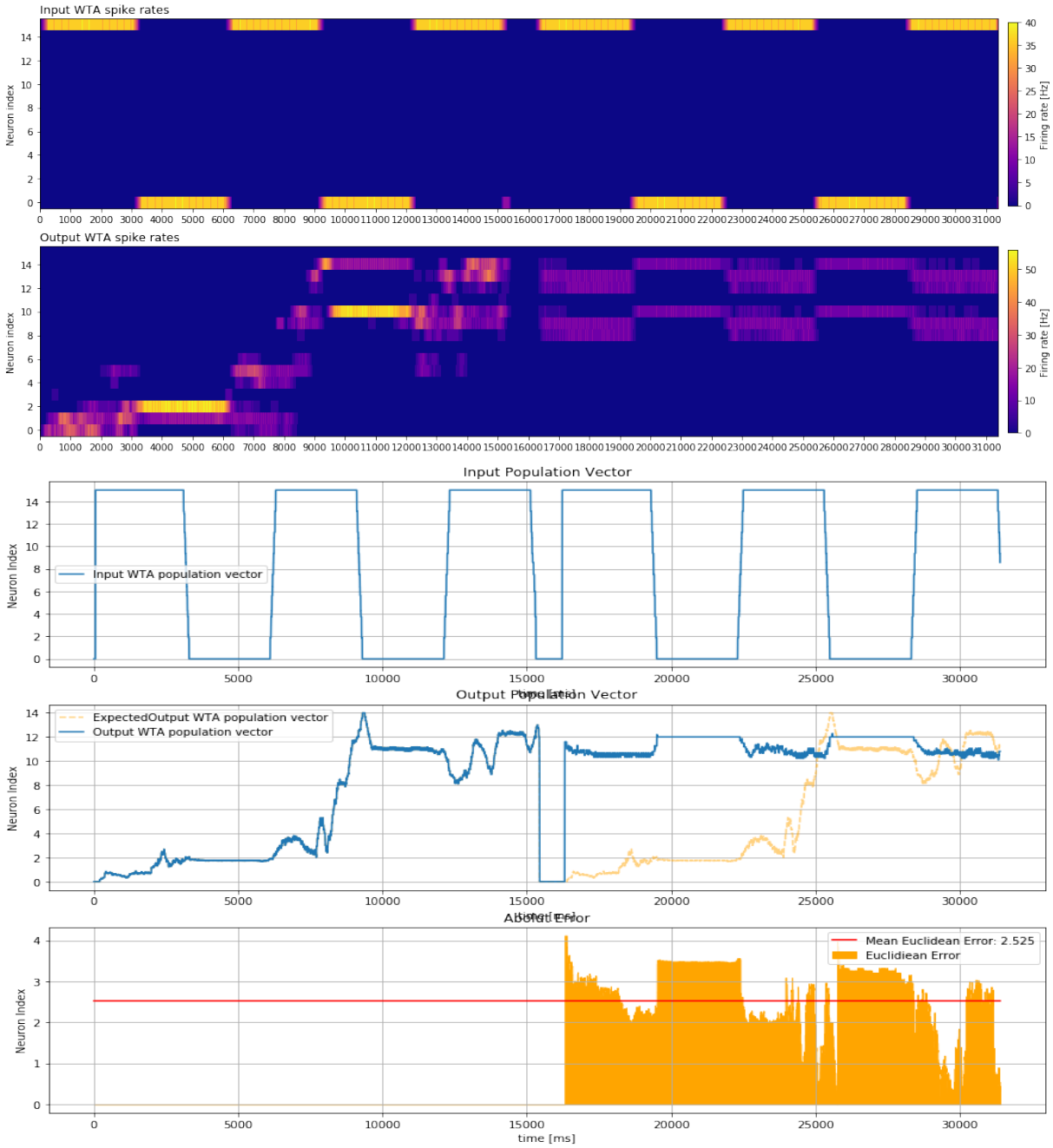


Figure 36: "Stop and Go" Trajectory - Spiking Neural Network Results (recorded data)

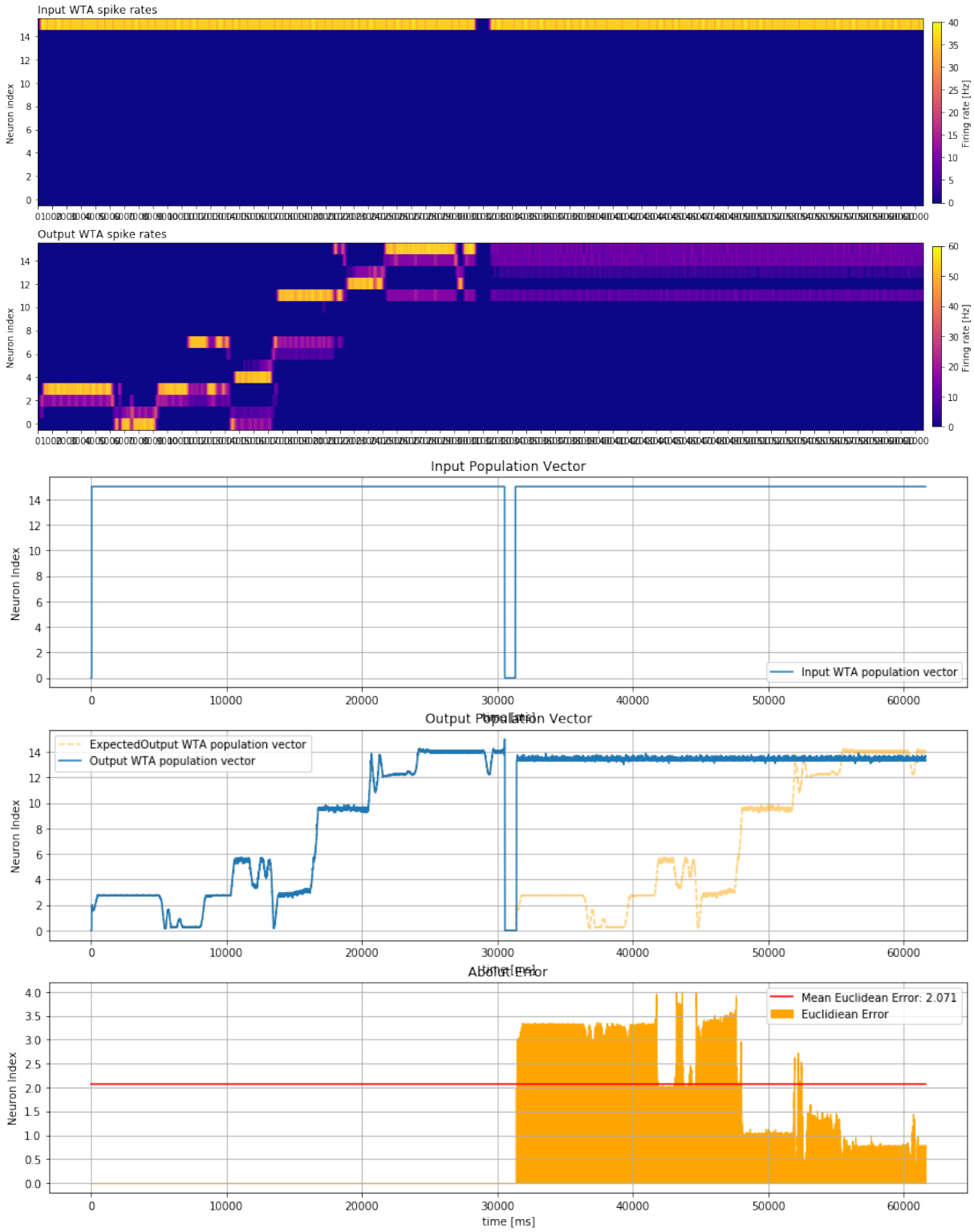


Figure 37: "Square" Trajectory Forward Mapping - Spiking Neural Network Results (recorded data)

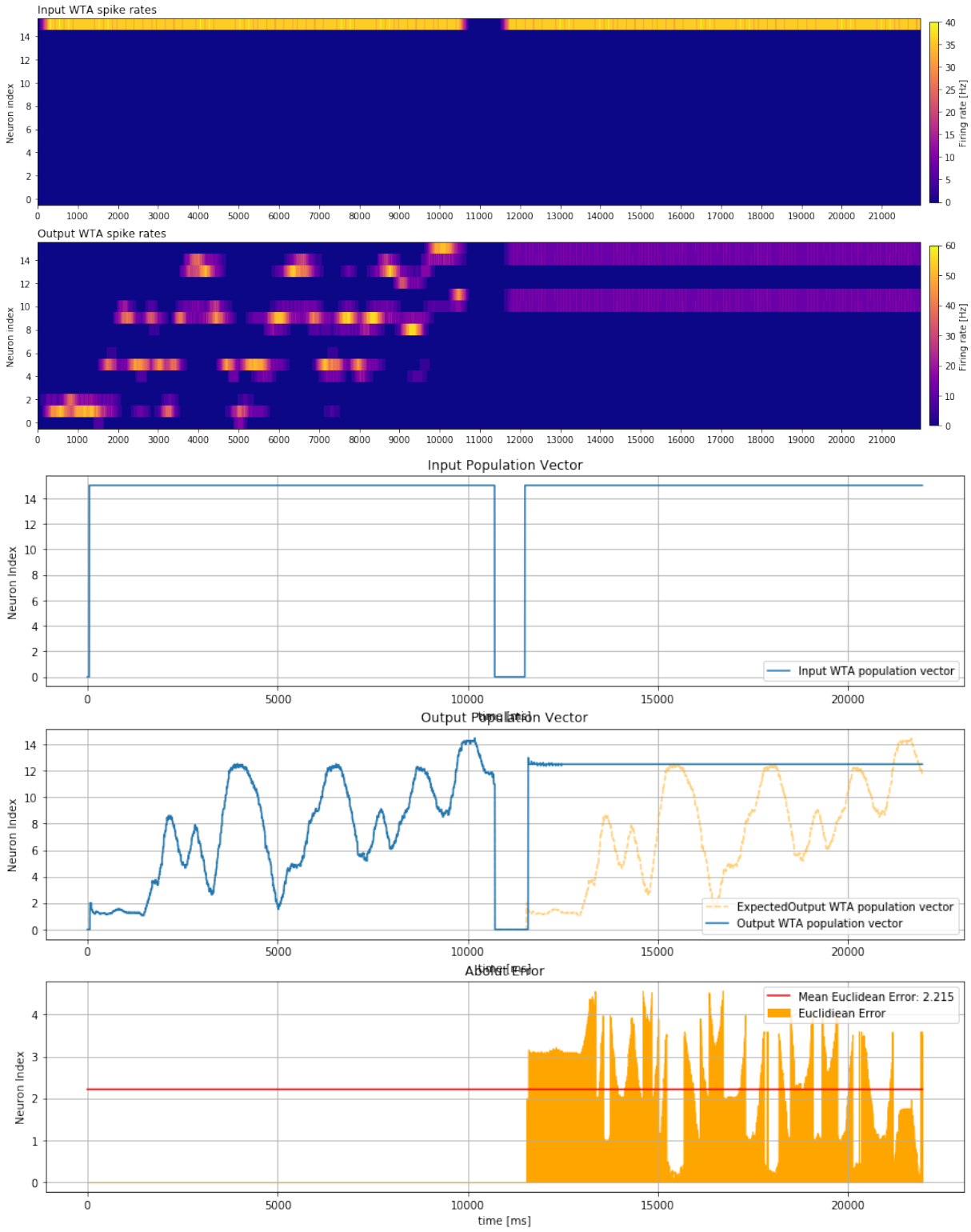


Figure 38: "Circle" Trajectory Forward Mapping - Spiking Neural Network Results (recorded data)

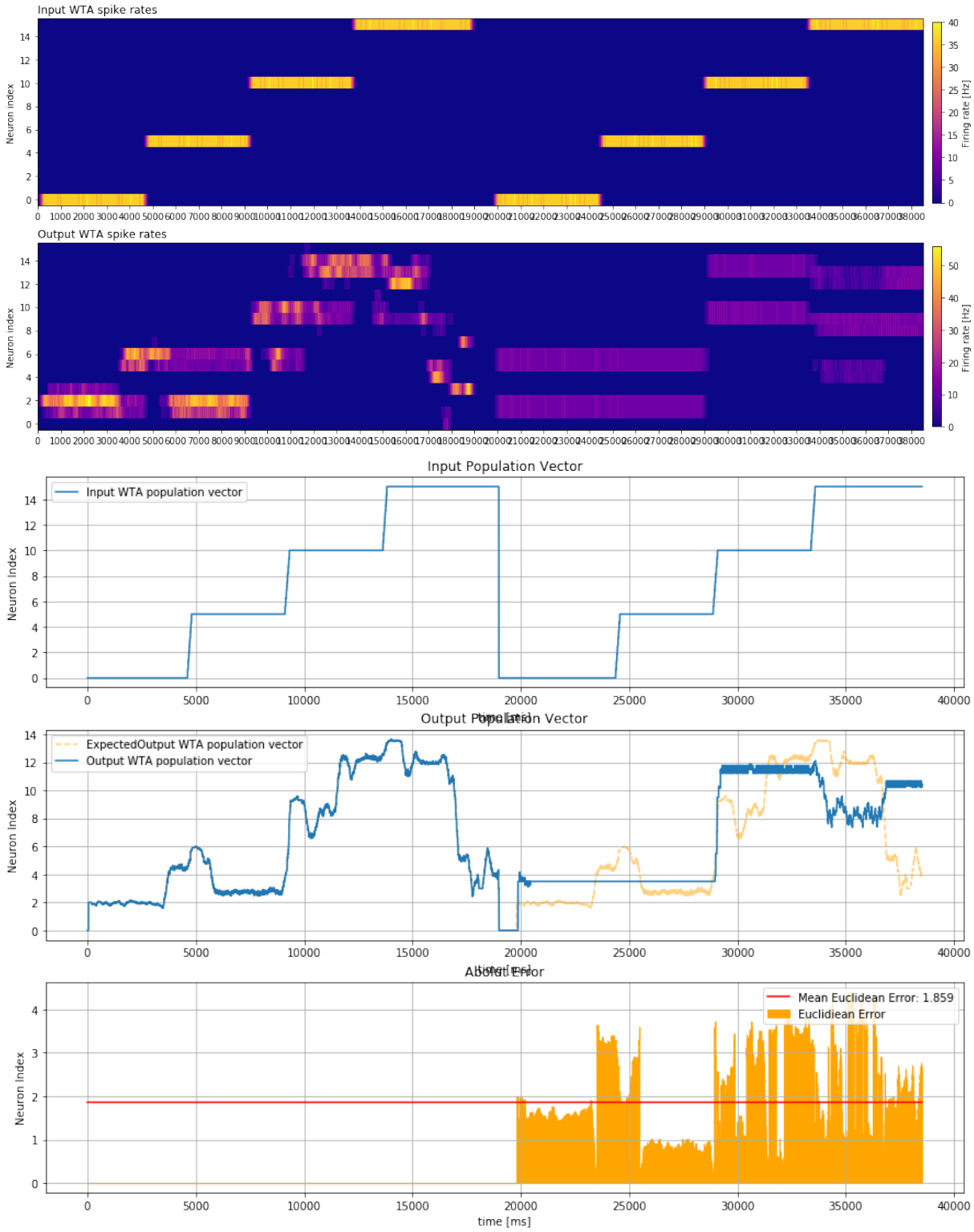


Figure 39: "Spiral" Trajectory Forward Mapping - Spiking Neural Network Results (recorded data)

References

[1] Inivation Pushbot User Guide
`https://inivation.com/support/hardware/edvs/` called on 03.06.2018

[2] Intel Loihi Wiki
`https://en.wikichip.org/wiki/intel/loihi` called on 05.06.2019

[3] Davide Plozza and Damiano Steiger. *Learning forward and inverse mappings for motor control of an omnidirectional robotic platform on a neuromorphic device (Intel Loihi).*

[4] J. Sjöström and W. Gerstner *Spike-timing dependent plasticity*

[5] Sebastian Glatz, Julien Martel, Raphaela Kreiser, Ning Qiao and Yulia Sandamirskaya. *Adaptive motor control and learning in a spiking neural network realised on a mixed-signal neuromorphic processor.*

[6] Dr Yulia Sandamirskaya and Dr Jörg Conradt *Learning Sensorimotor Transformations with Dynamic Neural Fields.*

[7] Viviane Yang, Balduin Dettling and Paul Joseph *Mapping and localisation using path integration and an event-based camera for assistance.*

[8] Ning Qiao, Hesham Mostafa, Federico Corradi, Marc Osswald, Dora Sumislawska, Giacomo Indiveri, and Giacomo Indiveri. *A Reconfigurable On-line Learning Spiking Neuromorphic Processor comprising 256 neurons and 128K synapses. Frontiers in neuroscience*