学位論文

Analyses of neural population dynamics generating distinct behaviors of *Drosophila* larvae

(ショウジョウバエ幼虫の様々な行動を生成する神経細胞集団活動ダイナミクスの解析)

平成２８年１２月博士（理学）申請

東京大学大学院理学系研究科
物理学専攻
尹　永択

Abstract

The way in which the central nervous system (CNS) governs animal movement is complex and almost impossible to solve solely from the movement pattern. We tackle this problem by observing the activity pattern of a large population of neurons in the CNS in Drosophila. *Drosophila* larvae show various behaviors, including forward locomotion, backward locomotion, and turning. We focused on these three behaviors and analyzed the neural activity of the larval CNS corresponding to these behaviors.

We recorded calcium imaging movie of isolated *Drosophila* larval CNS undergoing fictive locomotion using light-sheet microscopy which allows acquisition of neural activities in a large volume at a fast frame rate. After recording the movies, we executed preprocessing to eliminate artifacts. Since the size of the data was large and contained a lot of information, we compressed the data in an automated manner.

We then analyzed the neural activity of the CNS at a circuit level. The principal component analysis showed the circuit generates at least two distinct activity patterns. Also, by applying hidden Markov model to the activity of neural population, more detailed classification of the activity pattern was

made. Using information of the circuit state at each time, we found neurons which exhibit circuit state-specific activity. Also, we found neurons in the anterior CNS, which were active at the beginning of the fictive forward locomotion and thus were good candidates for triggers of forward locomotion.

Contents

List of Figures

Chapter 1

Introduction

1.1 Behavior and Nervous System

Animals actively respond to external stimuli and adapt well to the surrounding environment. The stimulus is converted into electrical signals by the sensory neurons, and the electric signals excite the motor neurons in a specific pattern via a complicated circuit composed of the interneurons. The muscles contract according to the specific pattern of motor neuronal activity, thereby eliciting reactions appropriate for the stimulation.

We define behavior as a series of muscle contractions. We further categorize behavior by its spatio-temporal pattern: different behaviors show distinct spatio-temporal patterns of muscle contraction. An animal uses one muscle in

various behaviors, which means a motor neuron innervating a particular muscle could be activated in various behaviors. Hence, the circuits corresponding to each behavior at least merge in the final layer (motor layer). Even if behaviors don't share motor neurons, the underlying circuits could be interconnected with each other, provided the behaviors are functionally related [23].

The nervous system of an animal which shows various behaviors contains multiple neural circuits. And the circuits are interconnected with each other. Thus, to truly understand the neural circuit dynamics, one has to observe the whole-brain activities.

1.2 Acqusition of Activity of Neural Circuit

Various techniques have been developed to investigate function of neural circuit. Electrophysiological methods which enable us to measure the activity of individual neurons, have the disadvantage that while it is possible to accurately measure membrane potential or ionic current crossing a cell's membrane, the number of cells that can be examined at the same time is extremely limited. Other neuroimaging methods like positron emission tomography (PET) and functional magnetic resonance imaging (fMRI) have been used to measure activity pattern of the whole nervous system in a non-invasive manner.

Although these techniques allow one to record neural activities in people and primates, also have a disadvantage of poor spatial resolution, so functional analysis at a cellular level is almost impossible. However, with the development of fluorescent probes and measuring instruments over the past decade, it has become possible to measure the activity dynamics of neural populations. Several studies using the technique of whole-brain functional imaging in verterbrates have been reported [1, 30, 25]. In this study, we performed the whole-brain functional imaging of *Drosophila* larvae and analyze the neural population dynamics of the circuit.

1.3 Calcium Imaging Technique

Calcium imaging is a method of indirectly measuring the activity of neurons using the phenomenon that intracellular calcium concentration rises when neurons are active. When a neuron excites, it releases a neurotransmitter at the end of the axon to convey the signal to the next cell. The signaling occurs as a result of electrical signal transmission through the axon to the cell terminal. As the voltage-dependent calcium channel opens, the extracellular calcium ion flows into the cell and the calcium concentration in the cell rises. If the calcium concentration could be measured, one would know that the neuron had been

active.

GCaMP [24] is a genetically encoded calcium indicator (GECI) made by fusion of circularly permuted green fluorescent protein (cpGFP), calmodulin (CaM), and M13. In the presence of calcium ions, CaM undergoes a conformational change and M13 binds to the hinge. This changes the structure of cpGFP and results in bright fluorescence.

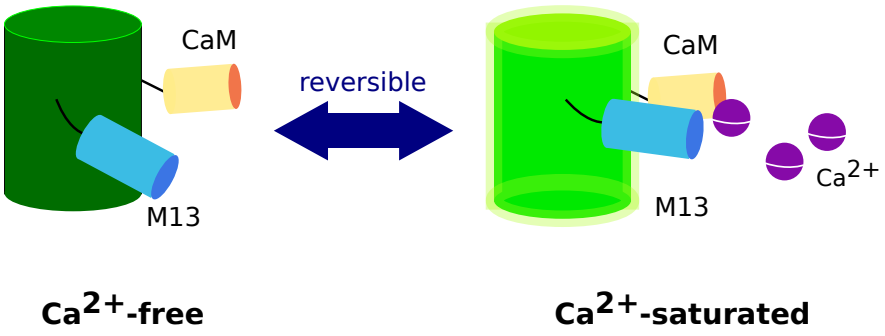


Figure 1.1: Mechanism of GCaMP. left: low calcium ion concentration. right: high calcium ion concentration.

Variants of GCaMP with improved performance such as GCaMP3 [28], GCaMP5 [2] have been developed so far. In this study, we use GCaMP6f which is one of the GCaMP6 [7] family (f for fast kinetics).

1.4 Model Animal: *Drosophila* larvae

1.4.1 Behaviors

Animals respond to changes in the environment by taking various behaviors. *Drosophila* larvae also show various behaviors [11, 19], including forward locomotion, backward locomotion, turning, rearing, and borrowing.

Drosophila larvae have 8 abdominal segments, and there are corresponding neuromeres in the CNS. During the forward locomotion (Figure 1.2a), the larvae contract the muscles of the abdominal segments of the body wall from the tail (A8) to the head (A1) [14]. During the backward locomotion, the direction of the propagation of the muscle contraction is opposite, from the segment A1 to segment A8. When the larvae change direction, they execute turning behavior (Figure 1.2b), contracting their muscles of only one side of the head.

1.4.2 Nervous System

The nervous system of a *Drosophila* larva is composed of thousands of cells, and the larval CNS is divided into two parts, the brain and the ventral nerve cord (VNC). The VNC, corresponding to the spinal cord in vertebrates, undertakes

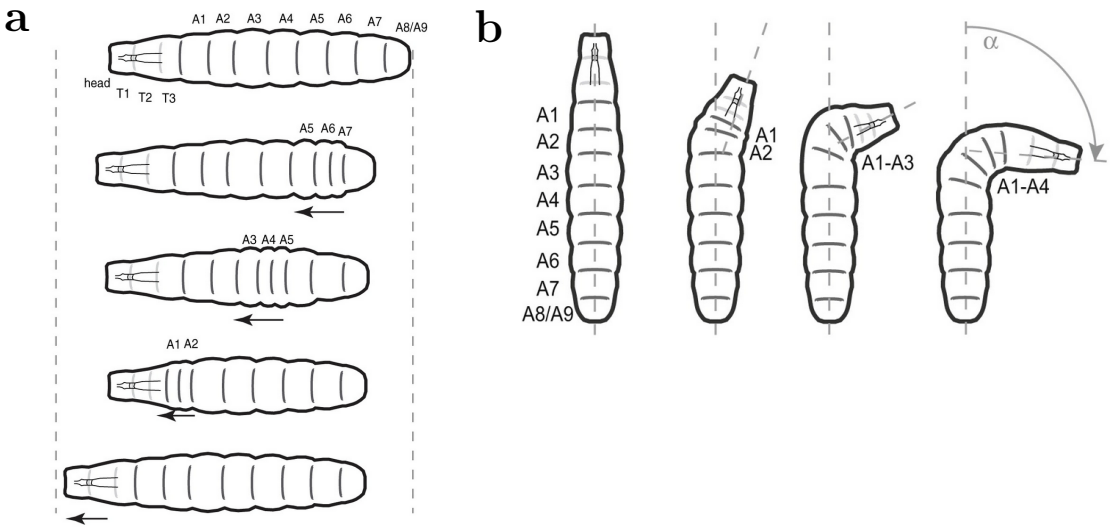


Figure 1.2: Behaviors of *Drosophila* larvae. **(a)** Forward locomotion. **(b)** Turning. Modified from Berni et al., 2012 [5].

various neural processing and sends the signals through motor nerves, resulting in movement of the larva.

Like the segmental structure of the body wall of the larva, the VNC also is divided into several neuromeres (Figure 1.3). Since motor neurons in each neuromere innervate muscles in the corresponding segment of the body wall, the patterns of neural activity of the motor neurons in the VNC reflect those of muscle contraction during the larval behaviors [9]. Therefore, it is possible to infer the actual larval behavior from the activity pattern of the motor neurons in the CNS.

Many of the functions of the neural circuit of *Drosophila* larvae have not

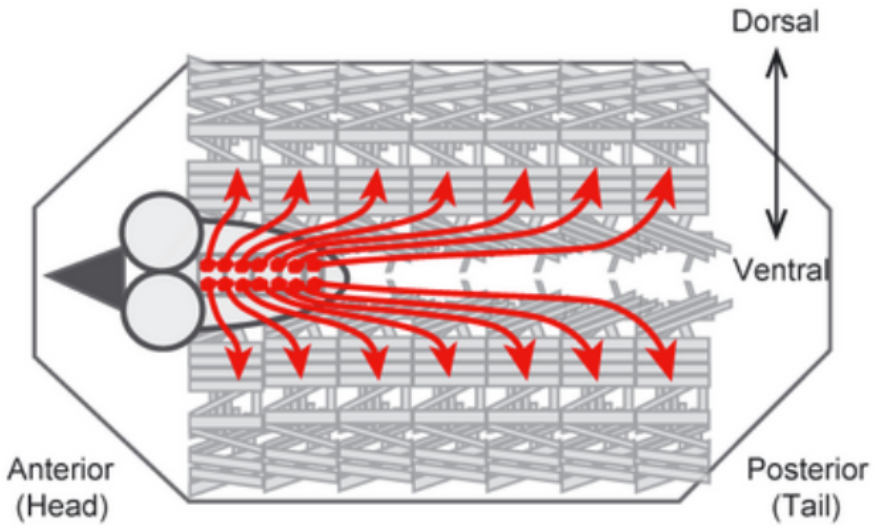


Figure 1.3: *Drosophila* larvae dissected along the dorsal midline and flatted to expose the muscles and the ventral nerve cord (VNC). Motor neurons (red circles), in each neuromere of the ventral nerve cord, innervate muscles in the corresponding segment of the body wall (red arrows). Modified from Kohsaka et al., 2012 [17].

been clarified. However, despite the complexity of the circuit, there are several reports on the motor circuit of *Drosophila* larvae at a module level. A class of segmentally arrayed local interneurons called PMSIs (period-positive median segmental interneurons) regulates the speed of locomotion in *Drosophila* larvae [18]. A class of glutamatergic interneurons called GVLIs (glutamatergic ventrolateral interneurons) is activated when the front of a forward motor wave reaches the second or third anterior segment [15]. Putative excitatory premotor interneurons, termed CLIs (cholinergic lateral interneurons) directly

activate motor neurons sequentially along the segments during larval locomotion [13]. The excitatory neurons called A27hs which are premotor and active only during forward locomotion, and the inhibitory neurons called GDLs which are necessary for both forward and backward locomotion [10]. However, the connection between the modules, where they are apparently controls the same motor neurons, remains unclear.

To analyze the neural circuit of the larvae at a circuit level, we measured neural activities of the CNS of *Drosophila* using the calcium imaging movies (Figure 1.4). Also, we developed both preprocessing program and analysis program so that we could handle large volumes of data objectively and efficiently.

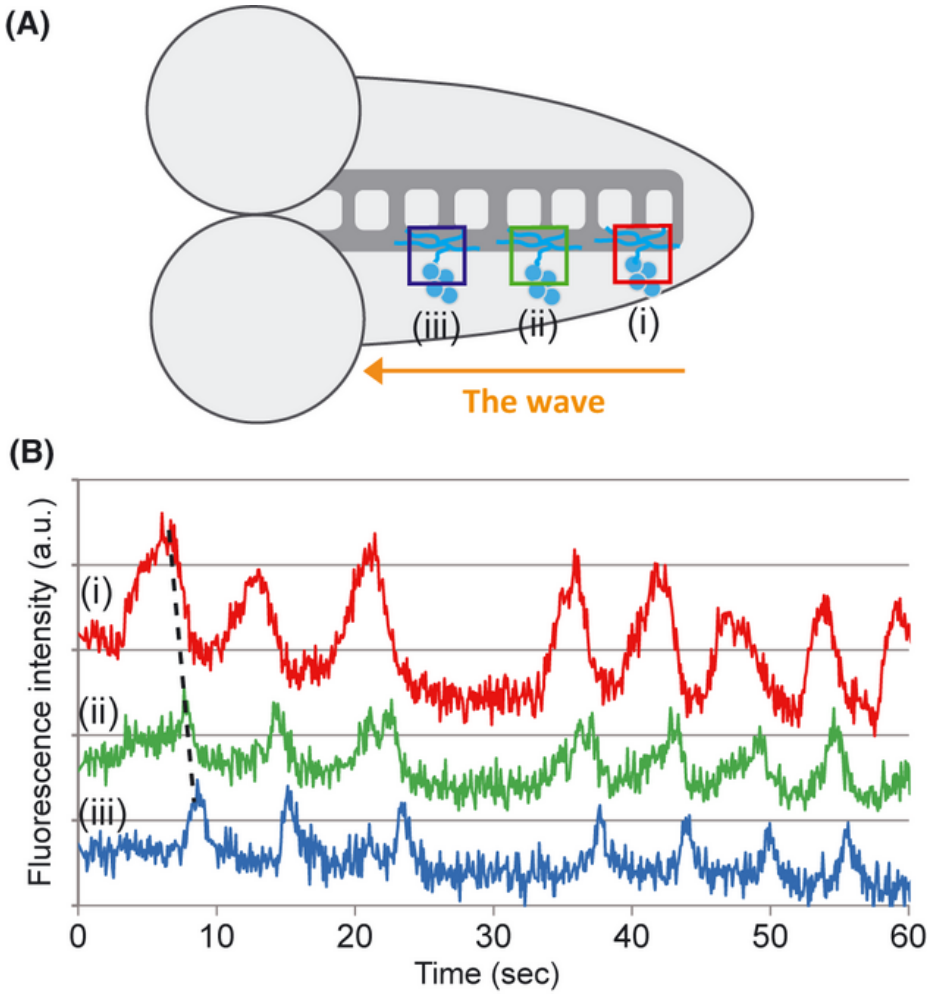


Figure 1.4: An example of calcium imaging. From the activity pattern of the motor neurons, one can infer the larval VNC is under the forward locomotion state. **(a)** Region of interests (ROIs) in the larval VNC. **(b)** The time series of the fluorescence intensity in the ROIs. Modified from Kohsaka et al., 2012 [17].

Chapter 2

Methods

2.1 *Drosophila* melanogaster strains

The Gal4-UAS system [6] is a genetic tool for *Drosophila*, which can be used to express specific genes to a certain group of cells. Suppose that there are flies of a Gal4 line to target a cell group and flies of a UAS line carrying a specific gene. By crossing these two lines, in the offspring GAL4 proteins expressed only in the specific cell group binds to UAS and drives the expression of the specific gene. In this way, by combining a Gal4 line and a UAS line, it is not necessary to prepare a gene construct for each group of cells to be expressed from the beginning.

In this study, the flies of *elav-Gal4, UAS-GCaMP6f, UAS-mCherry.nls*

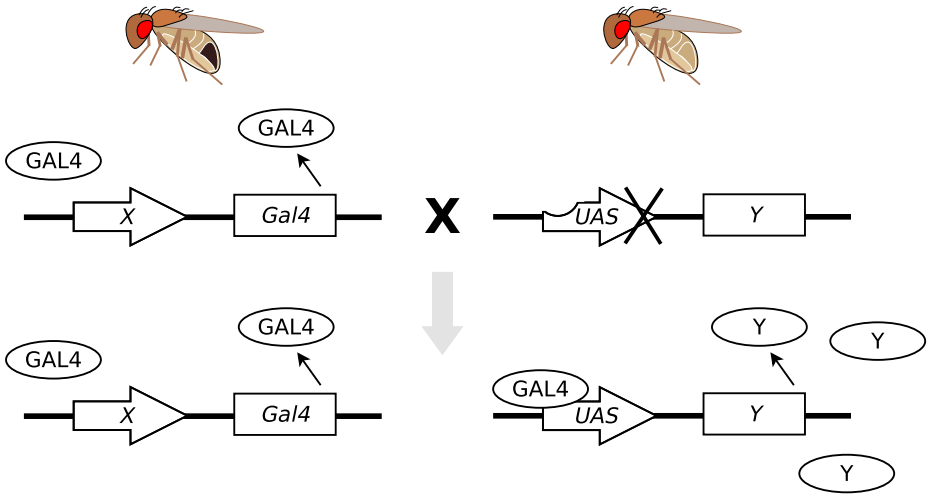


Figure 2.1: Gal4-UAS system. By crossing the flies of *X-Gal4* and *UAS-Y*, one can obtain the flies having both *X-Gal4* and *UAS-Y*, which express *Y* in tissue-specific manner.

are obtained by crossing two fly strains, one carrying *elav-Gal4* and *UAS-GCaMP6f*, and the other carrying *UAS-mCherry.nls*. The *elav-Gal4* line expresses the GAL4 protein in all the neurons, and the GAL4 protein binds to the UAS sequence, thereby expressing *GCaMP6f* and *mCherry.nls*.

2.2 Calcium Imaging

2.2.1 Preparation

We selected first instar larvae (21 $\sim$ 30 hours after egg laying), isolated the larval CNS (central nervous system) from the body so that motion by muscle contractions would not interfere calcium imaging. To prevent the isolated CNS from moving during the recording, the CNS was placed on the MAS (Matsunami adhesive silane)-coated slide glass (Matsunami, Osaka, Japan) and submerged in the TES buffer (Table 2.1).

Table 2.1: Composition of the buffers

	TES Buffer
TES	5 mM
NaCl	135 mM
KCl	5 mM
$CaCl_2$	2 mM
$MgCl_2$	4 mM
Sucrose	36 mM

2.2.2 Light-sheet Microscopy

Light-sheet microscopy is a method that captures fluorescence with a confocal microscope with illuminating a sample with sheeted light, thereby acquiring only fluorescence from a specified plane. We used ezDSLM [27], a digital

scanned light-sheet microscopy (DSLM) that utilized the illumination optics from a conventional confocal laser-scanning microscope (CLSM). Using the ezDSLM, we could record calcium imaging movie containing multiple focal planes ($\sim$ 30) in fast ($\sim$ 0.6 s/volume) frame rate.

2.2.3 Imaging Protocol

First, we recorded neural dynamics of the larval CNS (called the dynamics movie). In the dynamics movie, GCaMP6f fluorescence from fewer (32 planes), and sparser (2.485 μm/plane) focal planes are acquired so as to achieve high temporal resolution.

After recording the dynamics movie, we scanned the larval CNS for mCherry and GCaMP6f fluorescence to detect the positions of the neurons more precisely. We created two movies (an intermediate movie and a reference movie) from the scanning, which were recorded together using more (1000 planes), and denser (0.146 μm/plane) focal planes. These two movies were taken in a interleaved manner, using lasers of different wavelengths 488nm and 561nm for the intermediate movie and the reference movie respectively, so there is no need of spatial alignment between the intermediate movie and the reference movie.

The intermediate movies contain fluorescence of GCaMP6f whose feature is identical with that of the dynamics movies except for the spatial interval of the focal planes. The reference movies contain fluorescence of mCherry.nls which enables us to determine positions of the somas in the samples. The parameters used for the recordings are summarized in Table 2.2.

Table 2.2: Parameters of movie acquisition

Parameters	Dynamics	Intermediate	Reference
Dimension (normal)	32 planes	1000 planes	
Dimension (focal)	2048×1024 px^2		
Interval (normal)	2.485 μm/px	0.146 μm/px	
Interval (focal)	0.146×0.146 μm^2/px^2		
Temporal resolution	0.638 s	-	
Laser wavelength	488 nm		561 nm

2.3 Preprocessing

2.3.1 Drift Correction

Some of the acquired movies underwent drift in a focal plane. Since information about the position would be lost when the position of a sample shifts, we had to apply registration to the movies. However, the scenes of the movies change not only by the drift effect but also by the neural activity of the sample. Therefore, accurate registration was not possible with a simple intensity-based method

such as comparing the entire screen. Hence, we made a custom registration program which enables one to eliminate the translation of the movies.

2.3.2 Morphology Detection

To find the area of the CNS and the neuropil of samples, we used the reference movies and the intermediate movies, which have the following characteristics:

1. Intermediate movie: GCaMP6f proteins distribute in *whole cytoplasms* of all the neurons in the CNS.

2. Reference movie: mCherry proteins localize *in the nuclei* of the all neurons in the CNS.

Since in the intermediate movies fluorescence distributes in CNS, they can be used to detect the CNS. We estimated the intensity level of the fluorescence in the boundary region to remove the background and create a mask for only the CNS region. We used the Sobel-Feldman operator to estimate the boundary region.

After detecting the CNS region, we can determine the neuropil region by subtracting the bright region of the reference movie from the CNS region. Using the information of the neuropil region, we can obtain detailed information on the structure of VNC. Also, we can exclude the region from the cell

detection since somas would not exist in the neuropil.

Sobel-Feldman Operator The Sobel-Feldman operator [26, 8] creates an image which emphasizes the edges in the 2-dimensional input image. The operator uses 3×3 kernels, G_x and G_y:

$$G_x := \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad G_y := \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}. \tag{2.1}$$

Applying the above matrices for a 2-dimensional image f, we obtain images f_x, f_y which are approximations of the gradient of the input image along with the x- and y-axis respectively:

$$f_x = G_x * f, \qquad f_y = G_y * f. \tag{2.2}$$

Since gradient of the image has a high value in the boundary region (Figure 2.2), we can estimate the border of the sample in the image using the Sobel-Feldman operator.

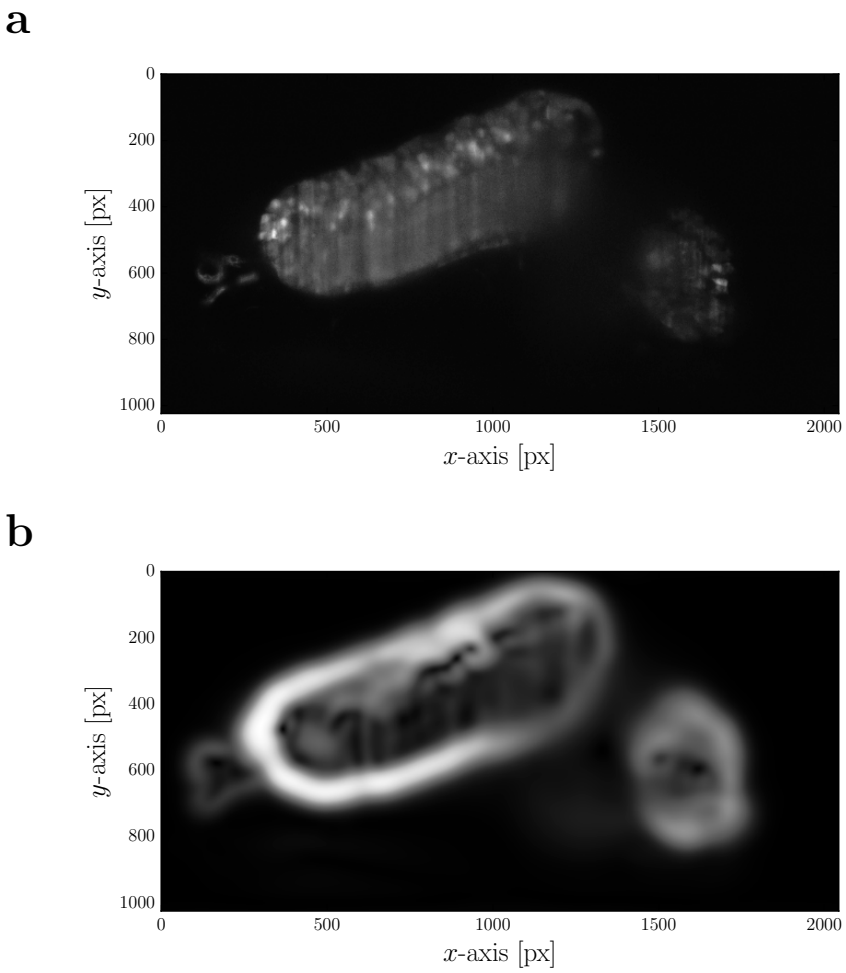


Figure 2.2: An example of the border detection with the Sobel-Feldman operator. **(a)** An image before the operation. **(b)** The image filtered by the Sobel-Feldman operator. The image indicates $f_x^2 + f_y^2$.

2.3.3 Cell Detection

Since the *Drosophila* larval CNS is composed of tens of thousand of neurons, manually identifying all neuron locations takes a lot of time and labor. In order to overcome this problem, we created a program to detect the position of cells from the images automatically.

The reference movie (imaging of the nuclei) were used to perform cell segmentation. Since lengths of all the edges of the voxel of the movie are identical (0.146 μm/px) by the settings of the recording, we could handle the image as an isotropic image. We applied LoG (Laplacian of Gaussian) on the reference movie and found local maxima of the result. Using the CNS mask and the neuropil mask calculated in the anatomy detection process, we excluded points which are outside of the CNS area or inside of the neuropil area.

In this section, we introduce a method for the cell detection by detecting the peaks of the fluorescence intensity in the images. In this method, we first detect the whole local maxima including the vertex by applying a method commonly used for edge detection of an object called LoG (Laplacian of Gaussian) [22]. After that, it is possible to detect only the peaks of the cells using an index [12] for distinguishing edges other than the vertex.

First of all, for the 3D image $f : \mathbb{R}^3 \to \mathbb{R}$, the LoG $\{f\}$ generates a 3D

image expressed as following defintion:

$$\mathrm{LoG}\{f\} := \nabla^2(G_\sigma * f) = \left\{ \frac{\partial^2}{\partial^2 x} + \frac{\partial^2}{\partial^2 y} + \frac{\partial^2}{\partial^2 z} \right\} (G_\sigma * f)\,, \tag{2.3}$$

which is applying Gaussian filter after blurring the image so derivatives on the image would be stable. G_σ is a Gaussian probability density function of standard deviation σ. Smoothing with G_σ results in reduced noise on the pixels of the scale below σ. So we can find pixels that take extremes calculating the Laplacian of the filtered image.

However, with this operation alone, not only the center of the cell we focus on but also the points corresponding to the edge where the change of pixel greatly changes would be detected. To exclude the points corresponding to the edges, we use information of Hessian $H(G_\sigma * f)$ of the 2D image defined as following equation:

$$H(G_\sigma * f) = \begin{bmatrix} \frac{\partial^2}{\partial^2 x} & \frac{\partial^2}{\partial x \partial y} \\ \frac{\partial^2}{\partial y \partial x} & \frac{\partial^2}{\partial^2 y} \end{bmatrix} (G_\sigma * f)\,. \tag{2.4}$$

Letting the first eigenvalue with Hessian obtained here be α and the second eigenvalue be β, these are proportional to the principal curvature of that point. If the ratio $r = \alpha/\beta$ is far from 1, the point corresponds to the edge, so we

could exclude the edges using the ratio r assigned to each pixel. For calculating r, we used the following relational expression:

$$\frac{\mathrm{Tr}(H)^2}{\mathrm{Det}(H)} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r+1)^2}{r} \tag{2.5}$$

2.3.4 Normalization

Fluorescence intensity in all the neurons can change by various influences besides the neural activity. Therefore, normalization of the fluorescence intensity was done to extract only neural activity as much as possible from the data.

Fluorescence time series of the cells are extracted from the dynamics movie, using the center of cells detected as described above. The time series data obtained from the calcium imaging movies change with various factors, and we want to extract neural activity which is one of these factors. The effect of the photobleaching, which is the main obstacle to the extraction of neural activity, causes a large change in fluorescence intensity. The change is often greater than that by neural activity, but the time scale of the phenomenon is much larger than that of neural activity. Therefore, we used a time window in a sufficiently small size to estimate the variation of the signal other than the neural activity by applying the Gaussian blur after taking the minimum value

of the window at each time. Assuming the result time series as the baseline, we obtained fluctuations in signal due to neural activity by dividing the original data by the baseline.

2.4 Circuit State Detection

Drosophila larvae show various behaviors including forward locomotion, backward locomotion, and turning [11, 19]. We focused on four kinds of activity patterns in the CNS corresponding to the forward locomotion, backward locomotion, left turning, and right turning.

To determine the circuit state at each time frame, we applied hidden Markov model (Section 2.4.1). We decided to use the hidden Markov model for the following reasons:

1. The model is relatively simple and clear.

2. From the physics, the state of the circuit at the next moment is supposed to be determined by the state at the moment just before it.

3. Each state does not necessarily produce the same result (neural activities of all neurons in the CNS) because of lack of information.

2.4.1 Hidden Markov Model

Hidden Markov model is a model which obeys the Markov process and the states are not observable. A brief explanation of the model is as follows.

Mathematical Description Let $\mathbb{S}$ be a state space composed of all the possible (hidden) states, and let $\mathbb{O}$ be a observation space composed of all the possible observations. Let a transition probability function $\phi : \mathbb{S} \times \mathbb{S} \to [0, 1]$ describe the probability of transition between the states, so $\phi(s_i, s_j)$ is the probability of transition from state s_i to state s_j. Finally, let a emission probability function $\theta : \mathbb{S} \times \mathbb{O} \to [0, 1]$ describe the probability of emission of observation, so $\theta(s, o)$ is the probability of emission of observation o for state s.

To determine the hidden states given the parameters, we implemented the Viterbi algorithm and applied it to the fluorescence time-series.

Viterbi Path Let T denote a set of time frames of the data. Given the initial probability function $\pi : \mathbb{S} \to [0, 1]$ ($\pi(s)$ is the probability of being in state s at the beginning), and the sequence of the observations $O : T \to \mathbb{O}$, we estimate the sequence of the states $S : T \to \mathbb{S}$ so that it maximizes the

likelihood:

$$L(S; O, \phi, \theta, \pi) := p(O|S, \phi, \theta, \pi)\,. \tag{2.6}$$

The Viterbi algorithm [29] is a dynamic programming algorithm which finds the sequence of hidden states, *viterbi path*, satisfying Equation (2.6). For descriptions of concrete procedure, refer to appendix B.2, the implementation in the Python 3. We can describe the procedure as the following pseudo-code:

Algorithm 1 Viterbi algorithm

function VITERBI(ϕ, θ, π, O)
  prepare $A : T \times \mathbb{S} \to \mathbb{R}$
  prepare $B : T \times \mathbb{S} \to \mathbb{S}$
  for $s \in \mathbb{S}$ **do**
    $A[T_{\text{first}}, s] \leftarrow \pi(s) \cdot \theta(s, O(T_{\text{first}}))$
  end for
  for $t \in T \setminus \{T_{\text{first}}\}$ sequentially **do**
    for $s \in \mathbb{S}$ **do**
      $B[t, s] \leftarrow \text{argmax}_{s' \in \mathbb{S}}\{A[t-1, s'] \cdot \phi(s', s)\}$
      $A[t, s] \leftarrow \theta(s, O(t)) \cdot \max_{s' \in \mathbb{S}}\{A[t-1, s'] \cdot \phi(s', s)\}$
    end for
  end for
  prepare $S : T \to \mathbb{S}$
  $S[T_{\text{last}}] \leftarrow \text{argmax}_{s \in \mathbb{S}}\{A[T_{\text{last}}, s]\}$
  for $t \in T \setminus \{T_{\text{last}}\}$ reversally **do**
    $S[t] \leftarrow B[t+1, S[t+1]]$
  end for
  return S
end function

We used Baum-Welch algorithm on the data from the samples so we can fix the parameters used in the Viterbi algorithm.

Baum-Welch algorithm The Baum-Welch algorithm [4] is EM (expectation-maximization) algorithm which maximizes the likelihood:

$$L(\phi, \theta, \pi; O) := p(O|\phi, \theta, \pi)\,. \tag{2.7}$$

For descriptions of concrete procedure, refer to appendix B.2, the implementation with the python 3. For the seed of the algorithm, see appendix A.1.

2.5 Other Numerical Methods

2.5.1 K-means Clustering

Let $f_i(t)$ be the fluorescence intensity of the t th time frame of the ith neuron ($1 \leq i \leq N$, $1 \leq t \leq T$). This time series data could be handled as N points in the T-dimensional space. Then, each point represents the characteristic of the whole time series data of each neuron. Clustering in this T-dimensional space makes it possible to divide a number of neurons into clusters showing several similar activity patterns.

In the following, we will explain K-means clustering. Given a set of S in the n-dimensional space, suppose that the set is divided into k groups. The first step is to randomly determine k points using the k-means++ algorithm [3]. Let them be $\{\mathbf{m}_1^{(0)}, \cdots, \mathbf{m}_k^{(0)}\}$.

1. Next, determine $S_i^{(n+1)}$ from each $\mathbf{m}_i^{(n)}$:

$$S_i^{(n+1)} = \{\mathbf{r} \in S \,|\, d_n(\mathbf{r}, \mathbf{m}_i^{(n)}) \leq d_n(\mathbf{r}, \mathbf{m}_j^{(n)}) \quad \text{for } j = 1, \cdots, k\} \tag{2.8}$$

$$\text{where} \qquad d_n(\mathbf{p}, \mathbf{q}) = \sum_{j=1}^{n} (p_j - q_j)^2.$$

2. Find the average $\mathbf{m}_i^{(n+1)}$ of elements of $S_i^{(n+1)}$:

$$\mathbf{m}_i^{(n+1)} = \frac{1}{|S_i^{(n+1)}|} \sum_{\mathbf{r} \in S_i^{(n+1)}} \mathbf{r}. \tag{2.9}$$

According to this procedure, (2.8) and (2.9) are iteratively calculated, and when $S_i^{(n^*+1)} = S_i^{(n^*)}$ is satisfied with some n^*, the calculation is terminated.

Chapter 3

Results

3.1 Distribution of Neurons in the CNS

In this study, calcium imaging was performed using *Drosophila* larvae of *elav-Gal4, UAS-GCaMP6f, UAS-mCherry.nls*, which enables to measure signals from all neurons in the CNS.

The distribution of neuronal cell body contained in the system was confirmed by immunostaining (anti-GFP) of the larval VNC (Figure 3.1). The population in the ventral side (Figure 3.1b) was found to be denser than that in the dorsal side (Figure 3.1a). We, therefore, decided to record the calcium imaging with the ventral side facing the camera (ventral side up).

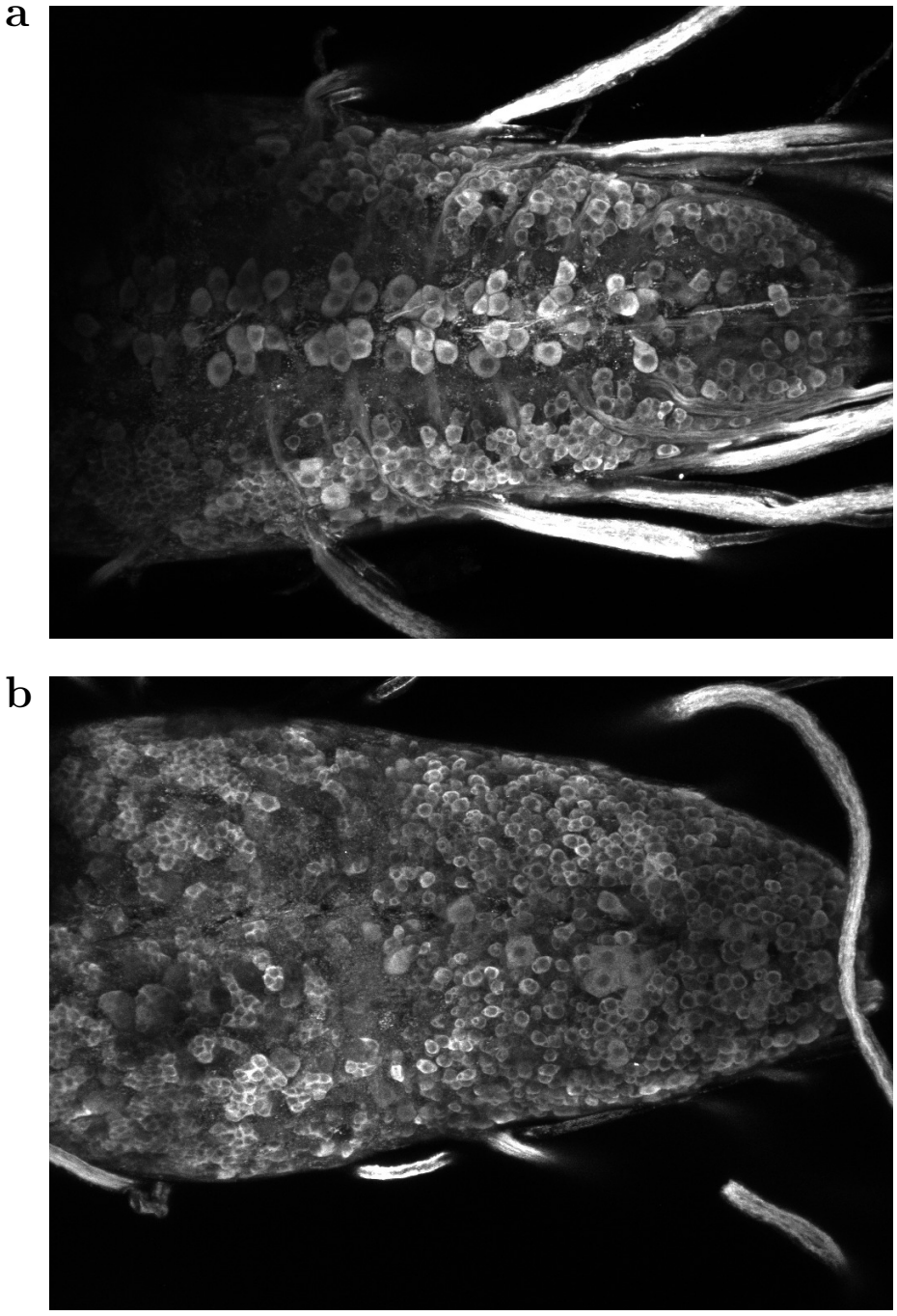


Figure 3.1: Immunostaining of the larval VNC of *elav > GCaMP6f* (anti-GFP). The left side of the figure is the head and the right side is the tail. **(a)** Dorsal side view. **(b)** Ventral side view.

3.2 Calcium Imaging Movie

We recorded the dynamics movie, the intermediate movie, and the reference movie for each sample. The summary of the characteristic of the movies is as follows:

1. Dynamics movie: high temporal resolution (0.638 s/volume), low spatial resolution (interval of z-stack: 2.485 μm), fluorescence of GCaMP6f

2. Intermediate movie: low temporal resolution, high spatial resolution (interval of z-stack: 0.146 μm), fluorescence of GCaMP6f

3. Reference movie: low temporal resolution, high spatial resolution (interval of z-stack: 0.146 μm), fluorescence of mCherry

3.2.1 Dynamics Movie

We recorded the dynamics movie to observe neural activities from the sample. We set the temporal resolution of the movie high enough ($\sim$ 0.6 s/volume) to detect the activity patterns during the larval behaviors.

After recording the movie, we confirmed the activity patterns were visible (Figure 3.2). To visualize the activities more clearly, we applied the Gaussian blur along the temporal axis and the spatial axis on the movie to obtain the

luminance in steady state. We then calculated the change from the steady state using the Gaussian blurred movie. From the resulting movie, we could observe the forward waves (Figure 3.2a) which propagate from the tail to the head, and the backward waves (Figure 3.2b) which propagate from the head to the tail.

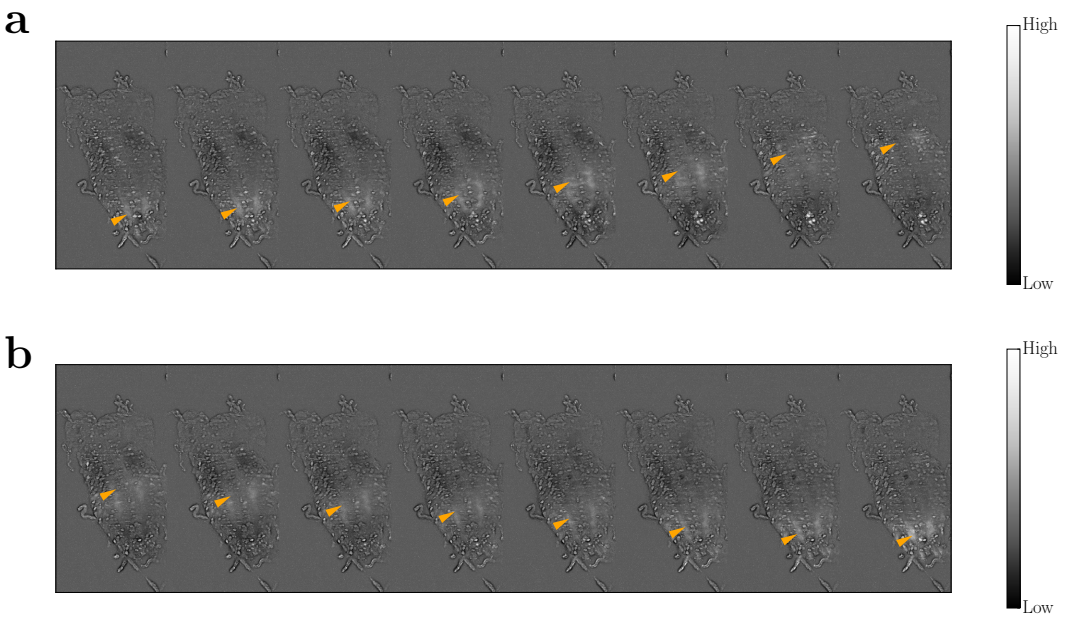


Figure 3.2: Activity patterns in the movie. The images of the movie in each time frame were spliced into one figure. In each image, the direction of the sample is anterior side up. **(a)** The forward waves propagate from the tail to the head. **(b)** The backward waves propagate form the head to the tail.

3.2.2 Intermediate Movie and Reference Movie

Neurons of *Drosophila* first instar larva consists of cell bodies of $\sim$ 1.5 μm radius and neurites of several hundred nanometers thick. Since the laser wavelength of the microscope was 488 nm or 561 nm, it was almost impossible to distinguish adjacent neurites of different neurons. Therefore we had to locate somas in the larval CNS so that we could extract the fluorescence intensity from their position.

For this reason, we recorded the reference movie to observe the morphology of the sample, especially the positions of the nuclei of the neurons. We set the voxel size in z-axis (normal to the focal plane) to the same value as that of the focal plane so that we could treat the image as an isotropic image.

However, registration of the two images of different features is not trivial. As shown in Figure 3.3a, the image of the reference movie which contains fluorescence of mCherry is quite different from the dynamics movie which contains fluorescence of GCaMP6f (similar with Figure 3.3b), since the latter visualizes the entire cell bodys whereas the former only visualizes the nuclei. Since we wanted to use the information obtained from the reference movie in the dynamics movie, we had to solve this registration problem.

In order to avoid the problem which is quite difficult to handle, we intro-

duced the intermediate movie. The intermediate movie (Figure 3.3b) contains fluorescence of GCaMP6f, so the registration between the dynamics movie and the intermediate movie is trivial (except for the difference of the spatial resolution along the z-axis). Also, the misalignment between the intermediate movie and the reference movie could be almost ignored from the imaging protocol. Therefore, by introducing the intermediate movie, the problem turned into a registration problem of two images with different spatial resolution (the intermediate movie and the dynamics movie).

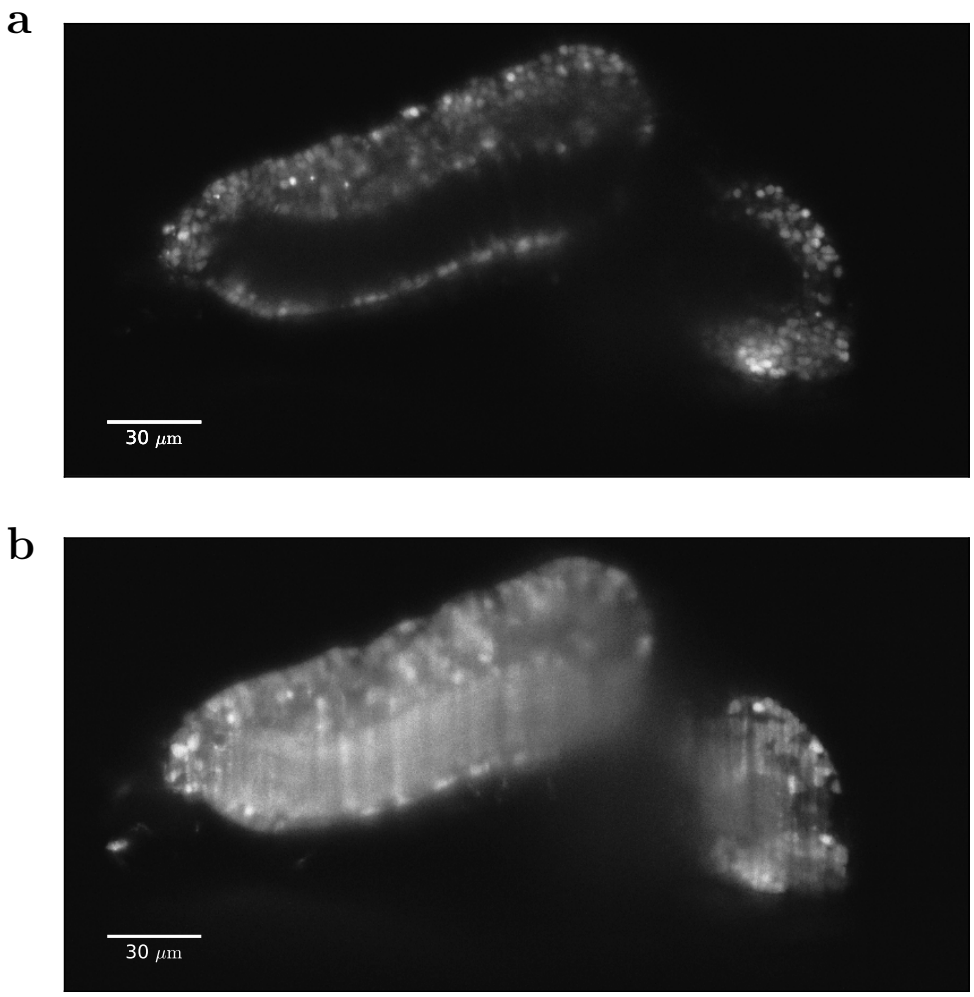


Figure 3.3: Comparison of the reference movie and the intermediate movie. We took the images from the same focal plane. **(a)** The reference movie. **(b)** The intermediate movie.

3.3 Drift Correction

Some of the acquired movies (the dynamics movies, the intermediate movies, and the reference movies) showed drift in the focal plane. To extract the signals from the neuron using the information of the positions, the signals would be mixed without proper registration. In fact, we confirmed that all the dynamic movies were translated by a larger amount ($\sim$ 50 px) than the scale of cell radius ($\sim$ 10 px).

We made a custom registration program which enables one to eliminate the translation of the movie and thus to obtain the correct signal from each cell. The acquired movies, especially the dynamics movies, had the following characteristics:

1. There is almost no rotation compared to the translation.

2. The intensity changes sporadically and greatly at various places according to the activity of the CNS.

3. The speed of translation caused by the drift is sufficiently slower than the frame rate of the movie.

4. Due to the photobleaching of the calcium indicator, the intensity of the movie is decreasing.

Considering the above characteristics, we decided to prepare the algorithm which satisfies the following conditions:

1. Parameters representing the movement of the sample can be narrowed down to a very small number representing only the translation.

2. We detect many feature points in the image and track the position of the feature points. Then we estimate the movement of the sample from feature points which shows the same behavior.

3. Referring to the previous feature points, we can reduce the computational cost since the deviation from the previous feature points is limited to a certain range.

4. Comparing the features, we use normalized cross-correlation (NCC) [21] rather than root-mean-square error (RMS), thereby reducing the effect of the photobleaching.

The flow of the algorithm is as follows.

Data Compression

Since the movement of the sample lies in the focal plane, we reduce the computational cost and the noise of the image by stacking the images of all the focal planes (z-stacks) at each time.

Feature Detection

To calculate the overall motion of the sample, we detect many feature points throughout an image (*template image*, drawn from the stacked images of the movie). First, by generating lattice-like points on the template image (Figure 3.4), we reduce the computational cost involved in the evaluation of feature points in the next step and avoid concentration of feature points. Next, an image around a point (256×256 px^2) is cut out (Figure 3.4) and template matching using normalized cross-correlation with the original large image was performed. From the distribution of the distances, we calculated the accuracy of matching and select a certain number (256, in this case) of feature points in descending order (from the smallest to the largest) of the value (Figure 3.4).

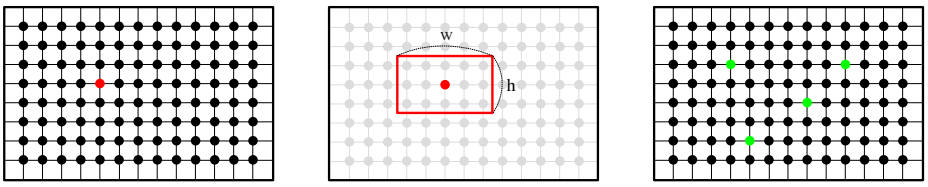


Figure 3.4: The process of the feature detection. left: the index (the accuracy of matching) is calculated for each lattice point. middle: For each feature point, we used the image of the area surrounded by a rectangle (width $w_f = 256$ px and height $h_f = 256$ px) centered on the point. right: Selected certain number of feature points.

Feature Tracking

First, we searched for all the feature points determined in the previous step in the image at the first time frame (Figure 3.5). We determined offsets from the correspondence of each feature point and create a histogram of those offset (Figure 3.6). Choosing the value of the highest density, we obtained the motion of the sample. For the search of feature points for the image at the next time frame, the information of the sample motion was used (Figure 3.7) to reduce the computational cost. By repeating this process, all feature points were detected in the images throughout the recording.

Figure 3.5: Searching for the feature point. Execute template matching using the image of each feature point, in all the area of the new image.

Movement Estimation

In order to suppress the vibration of the corrected moving image, we made a matrix that connects the histogram at each time frame created by the same method as the previous step (If it is an N-dimensional correction, the ma-

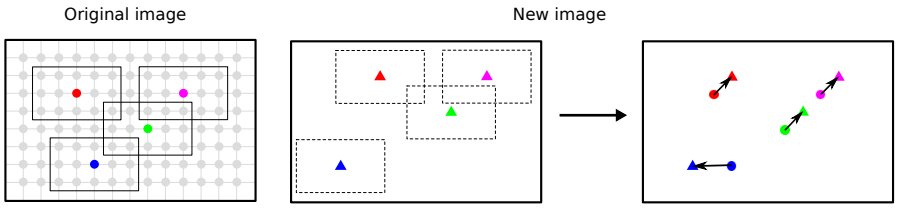


Figure 3.6: Estimation of the offset. Create the offset using the feature points in the template image and the corresponding points found in the new image. Estimation of the offset as a whole is done by ignoring few offsets (in the figure, the blue one) of different value.

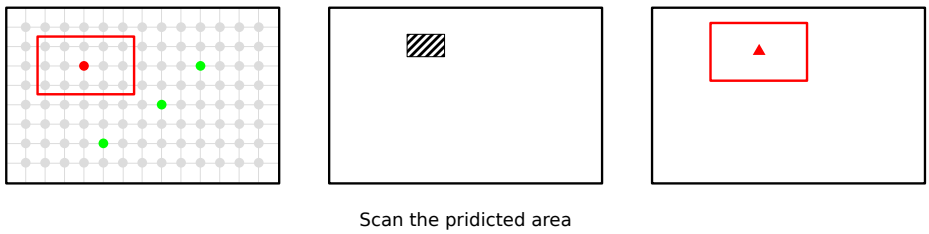


Figure 3.7: Efficient searching. Using the offset previously obtained, the search is done in a small area in a new image, which reduces computational cost significantly.

trix would become an (N + 1)-dimensional one). Since the temporal resolution was sufficiently higher than the speed of the movement of the sample ($\leq$ 1 px/frame), it is expected that more accurate estimation can be made by using the information of the adjacent time frame. Here we decided the movement of the sample by applying the Gaussian blur to the matrix and performing the same calculation used in the previous step at each time frame.

3.3.1 Application

Acquired movies, especially the dynamics movies underwent drift in the focal plane. In the case of dynamic movies, signals would be mixed on the signal extraction process without a correction. We applied the correction (Section 2.3.1) on all the movies, so we could safely use spatial information of the neurons to extract the signals of them.

Figure 3.8 shows an example of the result of the correction. We confirmed that in a typical dynamic movie, the amount of translation caused by the drift is large enough to distort the extracted signal since the offsets could be larger than the scale of the cell radius ($\sim$ 10 px).

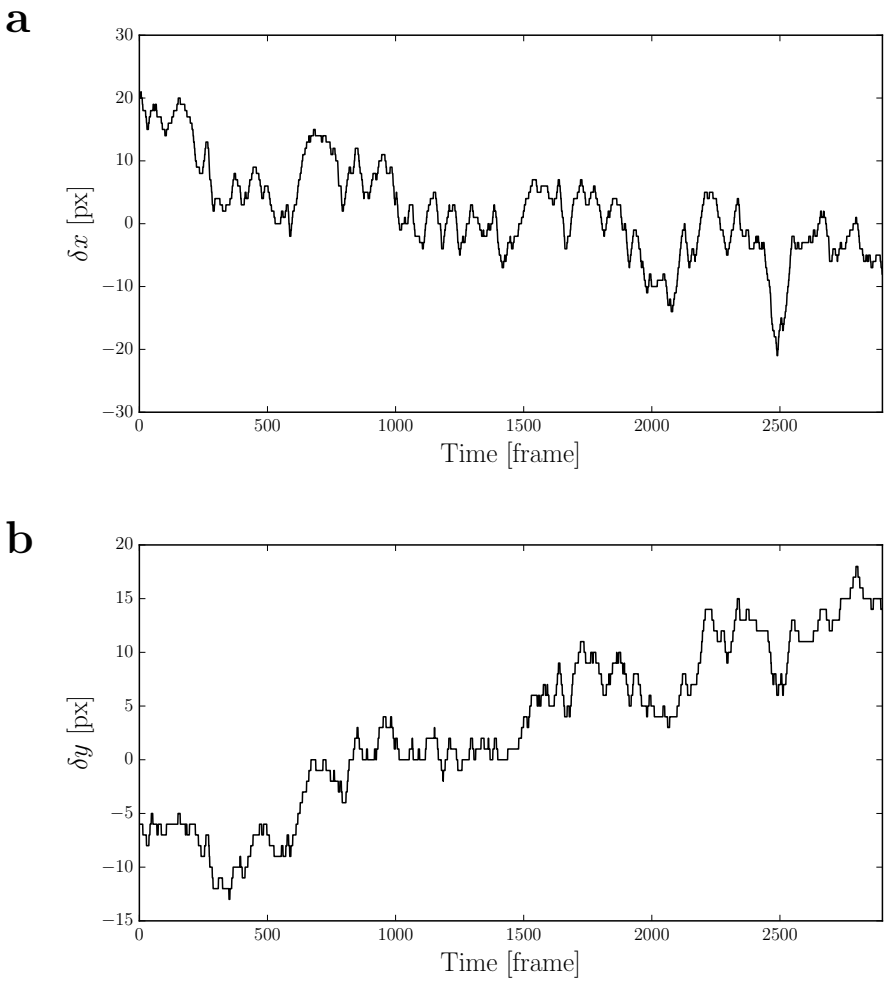


Figure 3.8: Drift correction on a sample movie. δx and δy indicate the offset from the template image in the x- and y-axis, respectively. **(a)** The offsets along with the x-axis. **(b)** The offsets along with the y-axis.

Also, we calculated the velocity of the movement of the sample in the movie, to test if check the speed of translation caused by the drift is sufficiently slower than the frame rate of the movie:

$$v_x(t) := \delta x(t+1) - \delta x(t)\,, \tag{3.1}$$

$$v_y(t) := \delta y(t+1) - \delta y(t)\,. \tag{3.2}$$

We confirmed the velocity (Figure 3.9) was sufficently slow (≤ 1 px/frame).

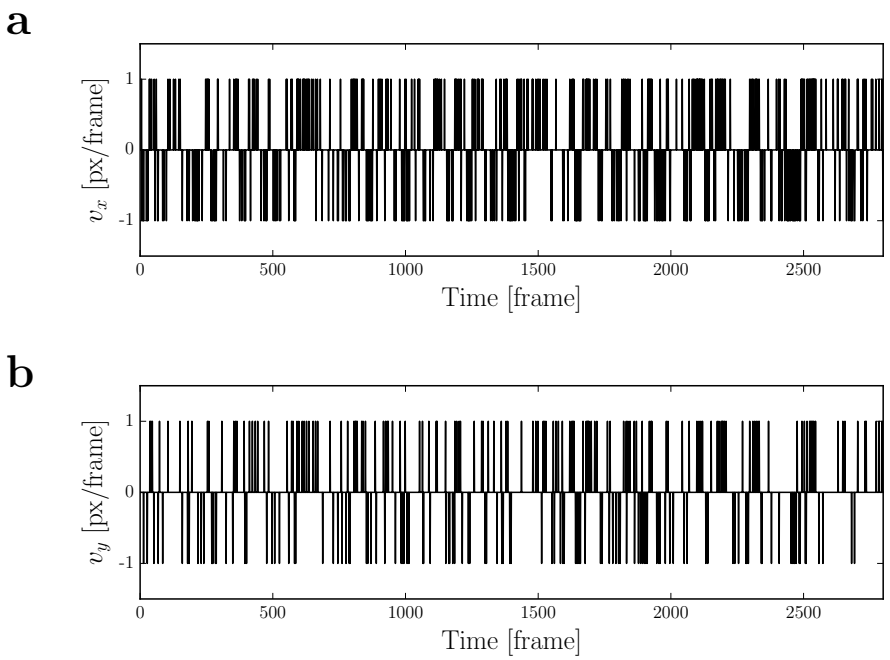


Figure 3.9: The velocity of the estimated drift of the sample movie. **(a)** The velocity of the drift along with the x-axis. **(b)** The velocity of the drift along with the y-axis.

Since there was no simple cost function which enables us to confirm that the correction was done properly, we created stacked images (Figure 3.10), averaging the movies along the temporal axis. If the correction is correct, the stacked image of the corrected movie (Figure 3.10c) would have high contrast and similar to a single frame of the movie (Figure 3.10a). Apparently, we found that the stack image made using the corrected image is clear, indicating the correction worked well.

Also, if the correction to the original movie was necessary, the stacked image of the original movie (Figure 3.10b) should be a blurred image. As expected, it was confirmed that the stacked image of the original movie was blurred significantly compared with the other images (Figure 3.10a, c).

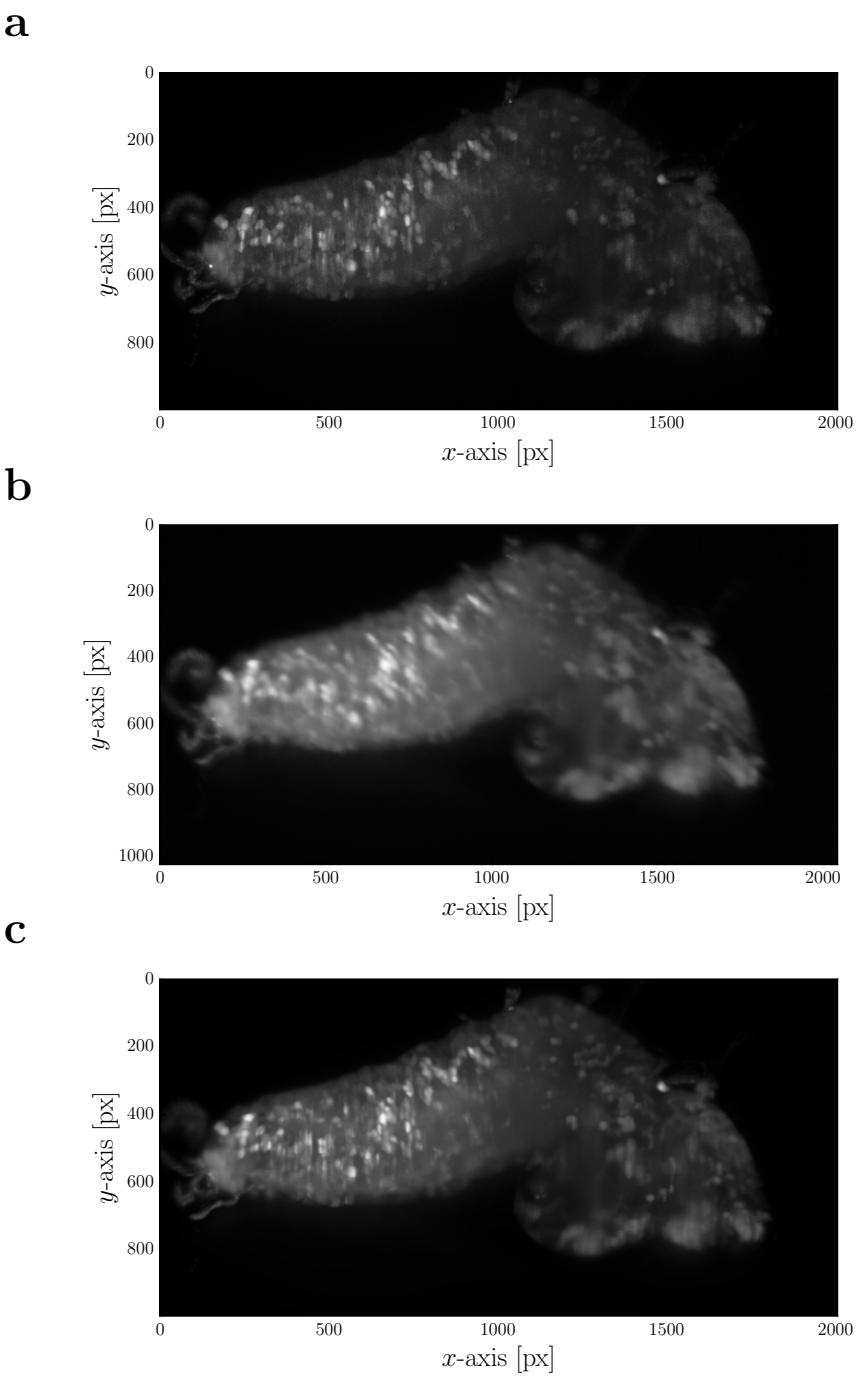


Figure 3.10: Drift correction on a sample movie. **(a)** An image of a single frame of a movie. **(b)** A mean-stacked image of the movie before the correction. **(c)** A mean-stacked image of the movie after the correction.

3.3.2 Validation

In order to evaluate the accuracy of the drift correction algorithm, the same algorithm was applied to artificial data. As the artificial data, we used the drift-corrected movies which were offset by a different amount for each time frame after the correction. Then, we compared the detected offsets against the simulated offsets in various conditions.

First, we applied the algorithm to movies translated by offsets having a form of a linear function:

$$\delta x_{\mathrm{sim}}(t) = s_x t\,, \quad \delta y_{\mathrm{sim}}(t) = s_y t\,, \tag{3.3}$$

where s_x and s_y are the rate of shift along with x- and y-axis respectively. For $s_x = 80/2800$ px/frame and $s_y = 40/2800$ px/frame, we evaluated the difference between the offsets of the simulated value and that of the estimate (Figure 3.11).

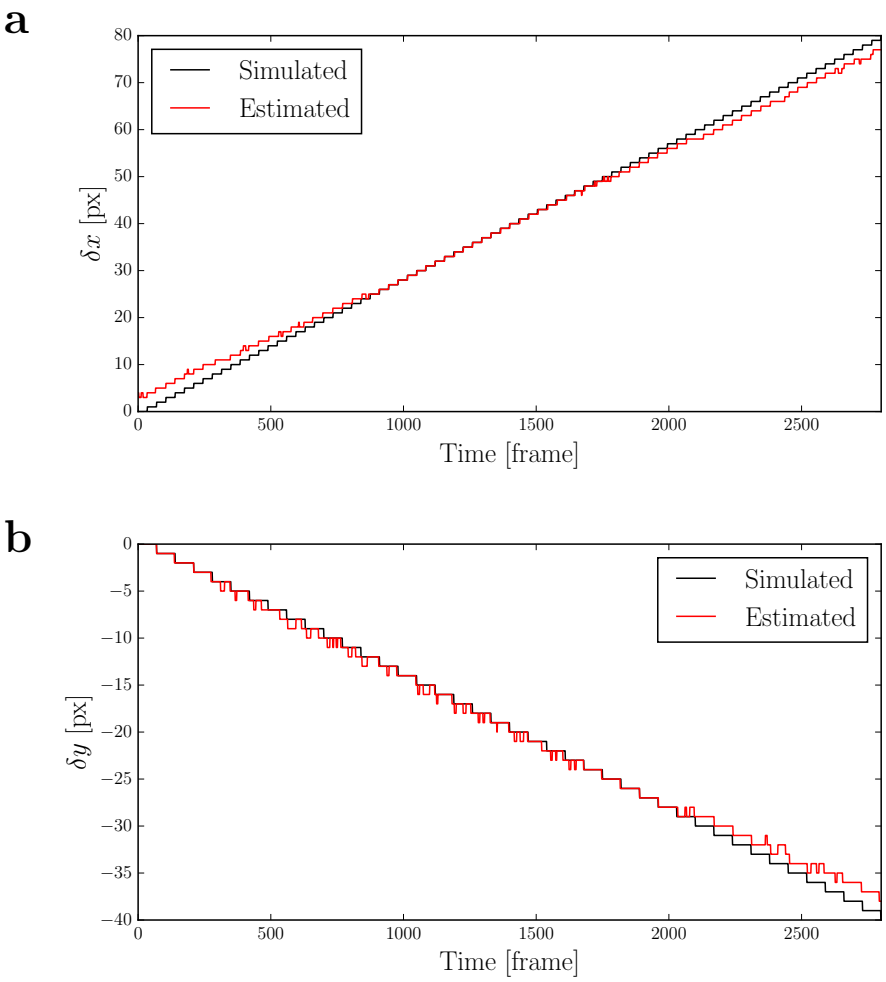


Figure 3.11: Drift correction on the artificial data (linear drift: $s_x = 80/2800$ px/frame, $s_y = 40/2800$ px/frame). The red line indicates estimated offsets by the correction and the black line indicates the simulated offsets. **(a)** The offsets along with the x-axis. **(b)** The offsets along with the y-axis.

Since we would align the movie using the estimated offsets, it is natural to decide the worst-case error as the following way:

$$E_x^{\mathrm{worst}} := \max_t \Delta x(t) - \min_t \Delta x(t)\,, \tag{3.4}$$

$$E_y^{\mathrm{worst}} := \max_t \Delta y(t) - \min_t \Delta y(t)\,, \tag{3.5}$$

where $\Delta x(t) := \delta x_{\mathrm{est}}(t) - \delta x_{\mathrm{sim}}(t)$ and $\Delta y(t) := \delta y_{\mathrm{est}}(t) - \delta y_{\mathrm{sim}}(t)$.

In the case of Figure 3.11, E_x^{worst} was 7 px and E_y^{worst} was 3 px, which was small compared to the cell radius ($\sim$ 10 px).

Also, we calculated the velocity of the movement of the sample in the simulated movie, to confirm the speed of translation caused by the simulated drifts:

$$v_x(t) := \delta x_{\mathrm{sim}}(t+1) - \delta x_{\mathrm{sim}}(t)\,, \tag{3.6}$$

$$v_y(t) := \delta y_{\mathrm{sim}}(t+1) - \delta y_{\mathrm{sim}}(t)\,. \tag{3.7}$$

In the case of Figure 3.11, we confirmed the velocity (Figure 3.12) was sufficiently slower ($\leq$ 1 px/frame) than the frame rate of the movie.

Also, we applied the algorithm to movies translated by offsets having a

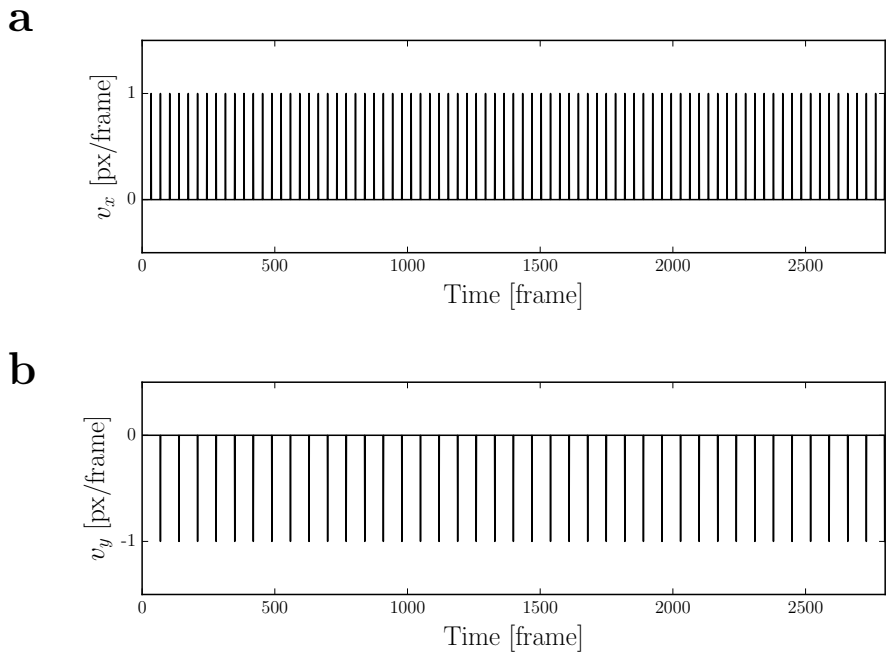


Figure 3.12: The velocity of the simulated drift and estimated drift of the artificial data (linear drift: $s_x = 80/2800$ px/frame, $s_y = 40/2800$ px/frame). **(a)** The velocity of the drift along with the x-axis. **(b)** The velocity of the drift along with the y-axis.

form of a sine function:

$$\delta x_{\text{sim}} = a_x \sin w_x t\,, \quad \delta y_{\text{sim}} = a_y \sin w_y t\,, \tag{3.8}$$

where w_x and w_y are the frequencies along with x- and y-axis respectively. With different values of w_x and w_y, we evaluated the differences between the offsets of the simulated value and that of the estimate (Figures 3.13 and 3.15).

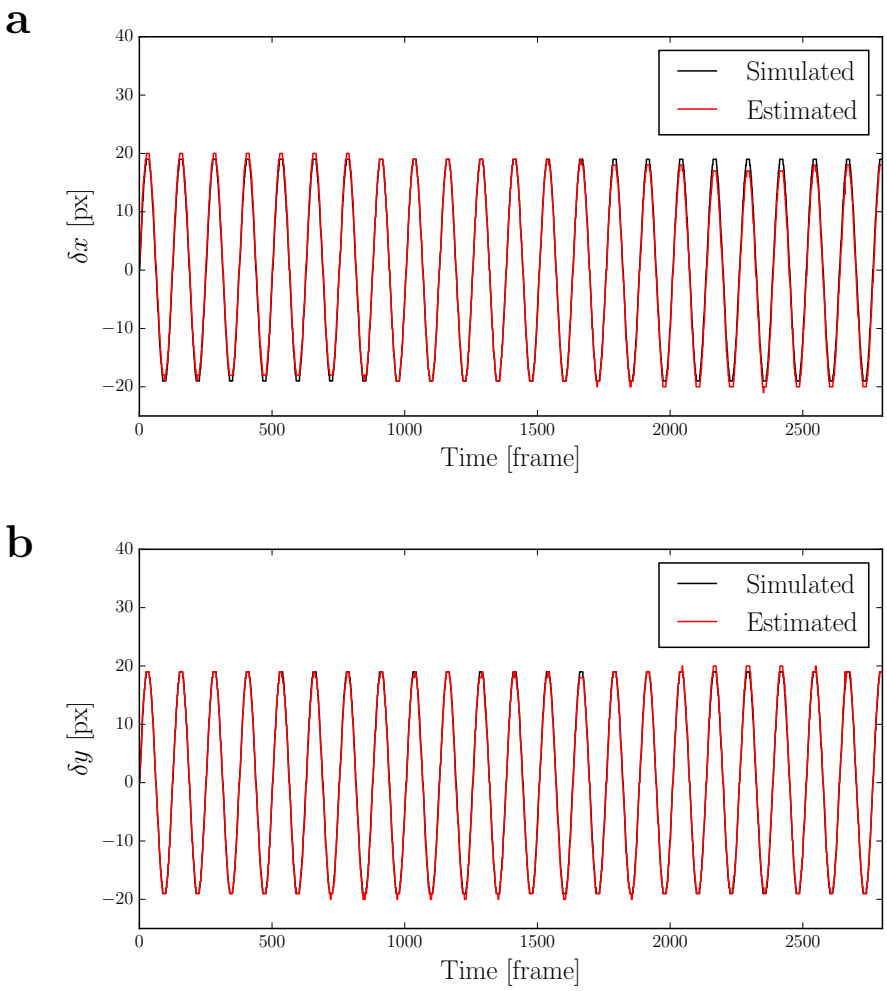


Figure 3.13: Drift correction on the artificial data (sine drift: $a_x = a_y = 20$ px, $w_x = w_y = 0.05$ rad/frame). The red line indicates estimated offsets by the correction and the black line indicates the simulated offsets. **(a)** The offsets along with the x-axis. **(b)** The offsets along with the y-axis.

We calculated the worst-case error in the first case (Figure 3.13), E_x^{worst} was 3 px and E_y^{worst} was 2 px, which are small compared to the cell radius. Also, the speed of the movement caused by the simulated drift in this case (Figure 3.14) was slow (≤ 1 px/frame).

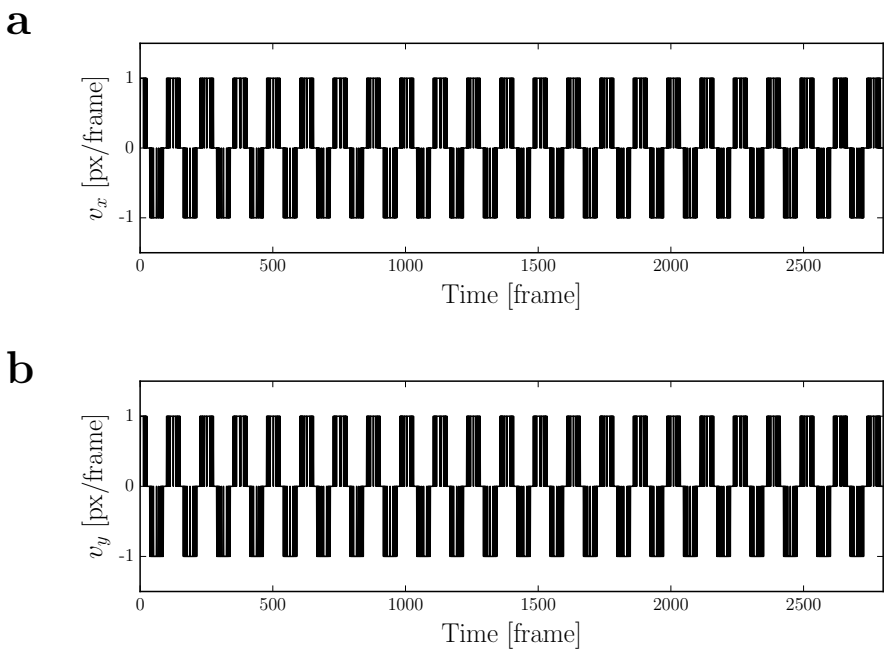


Figure 3.14: The velocity of the simulated drift and estimated drift of the artificial data (sine drift: $a_x = a_y = 20$ px, $w_x = w_y = 0.05$ rad/frame). **(a)** The velocity of the drift along with the x-axis. **(b)** The velocity of the drift along with the y-axis.

For the fastly vibrating movie (Figure 3.15), on the other hand, the worst-case error was relatively large, E_x^{worst} was 7 px and E_y^{worst} was 5 px, which are much larger than the previous case of sine draft. Since the range of the translation caused by the sine drift was same with the previous case (Figure 3.13), it can be inferred that the deterioration of the error is due to the speed of the translation (Figure 3.16), which was 2 fold of the previous case (Figure 3.14).

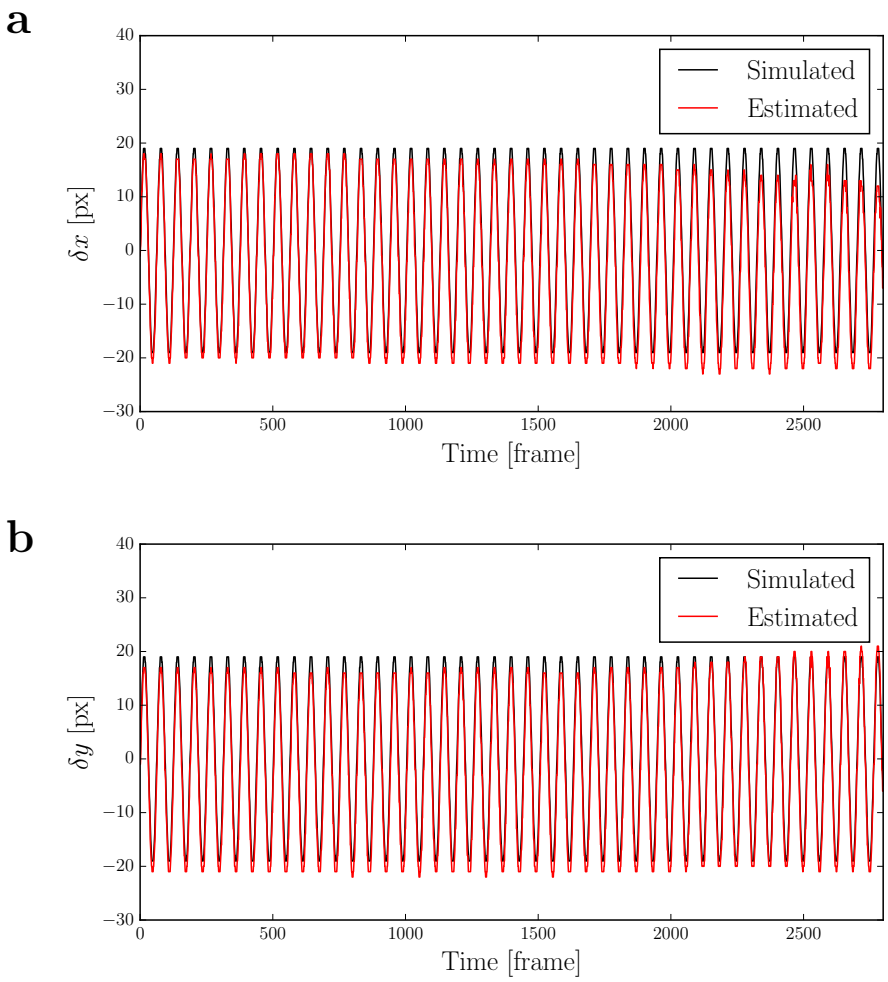


Figure 3.15: Drift correction on the artificial data (sine drift: $a_x = a_y = 20$ px, $w_x = w_y = 0.1$ rad/frame). The red line indicates estimated offsets by the correction and the black line indicates the simulated offsets. **(a)** The offsets along with the x-axis. **(b)** The offsets along with the y-axis.

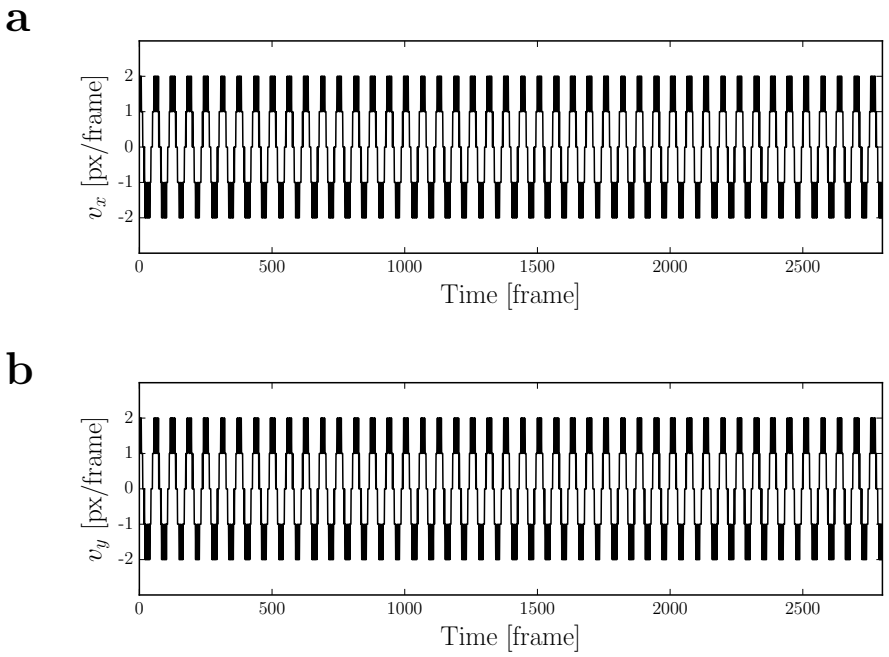


Figure 3.16: The velocity of the simulated drift and estimated drift of the artificial data (sine drift: $a_x = a_y = 20$ px, $w_x = w_y = 0.1$ rad/frame). **(a)** The velocity of the drift along with the x-axis. **(b)** The velocity of the drift along with the y-axis.

3.4 Normalization

Since it was able to calculate the positions of all cells, we extracted time series of fluorescence (Figure 3.17a) by averaging the values in the volume belonging to the location of each cell in the dynamics movie (from now on, we will call it *signals*). The signals obtained from the calcium imaging movies change with various factors, and we need to extract neural activity from them.

We first eliminated the effect of the photobleaching. The intensity of the calcium imaging movie decreases with the lapse of time. However, the time scale of the photobleaching is much larger that of the neural activity. To estimate the intensity by the other factors than the neural activity, we calculated local minima of the signal in a small area (radius of ~ 7 time frames or 4.5 s which is the time scale of the neural activity), and applied the Gaussian blur with a small standard deviation (~ 7 time frames) on the result (Figure 3.17a). Then we use the resulting time series as the baseline of the signal.

Using the baseline, we could obtain the change in the intensity during the neural activity (Figure 3.17b). However, the baseline was lower than the real baseline, since we set the baseline as the minimum of each local region. Therefore, we estimated the statistics of the noise assuming the distribution of the noise is the Gaussian distribution and obtained the center of the distribution.

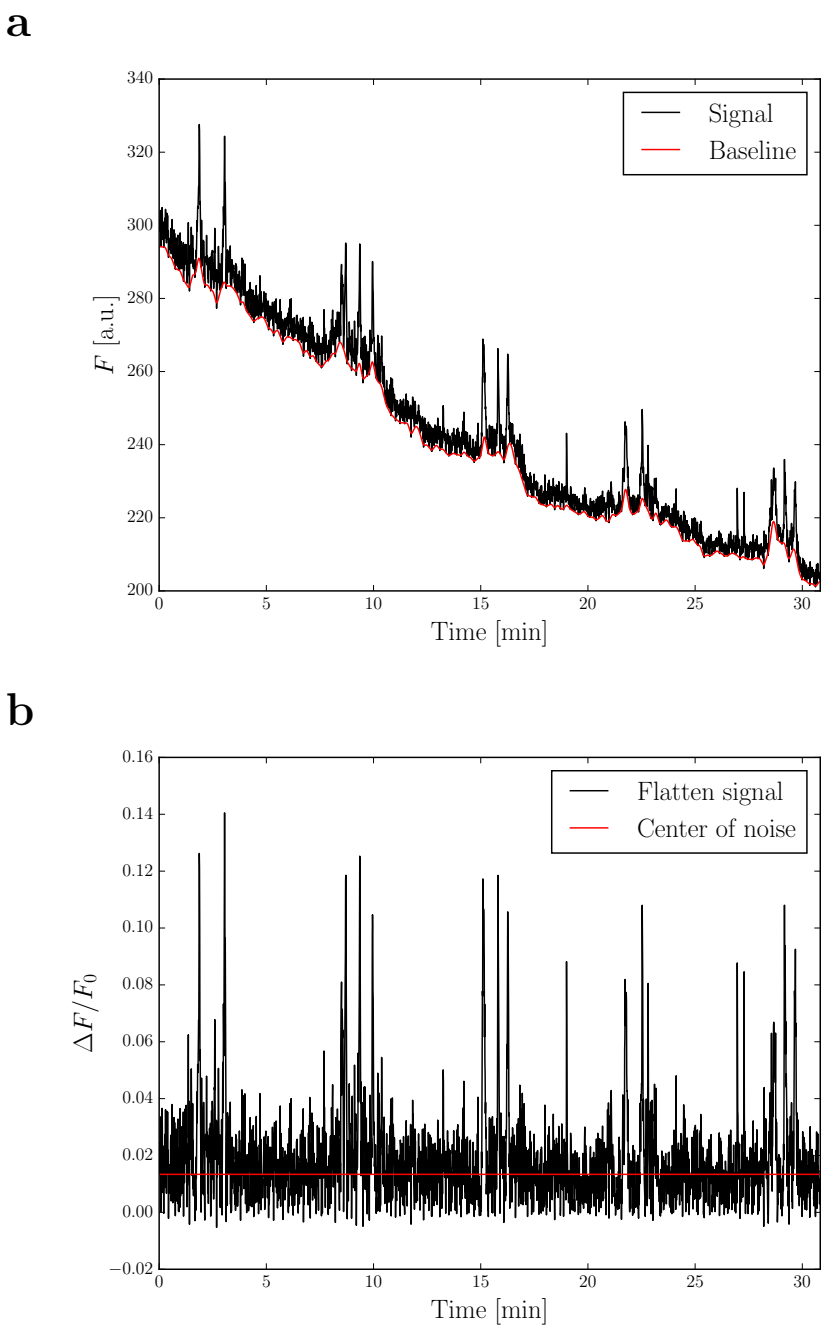


Figure 3.17: The normalization process consists of the two steps. **(a)** Estimating the baseline of the signal. The red line indicates the estimated baseline. **(b)** Estimating the center of the noise. After the signal had been flattened, we shifted the flatten signal by the center of the noise.

In the following analyses, the normalized signal refers to the modified signal obtained through this process (Figure 3.18).

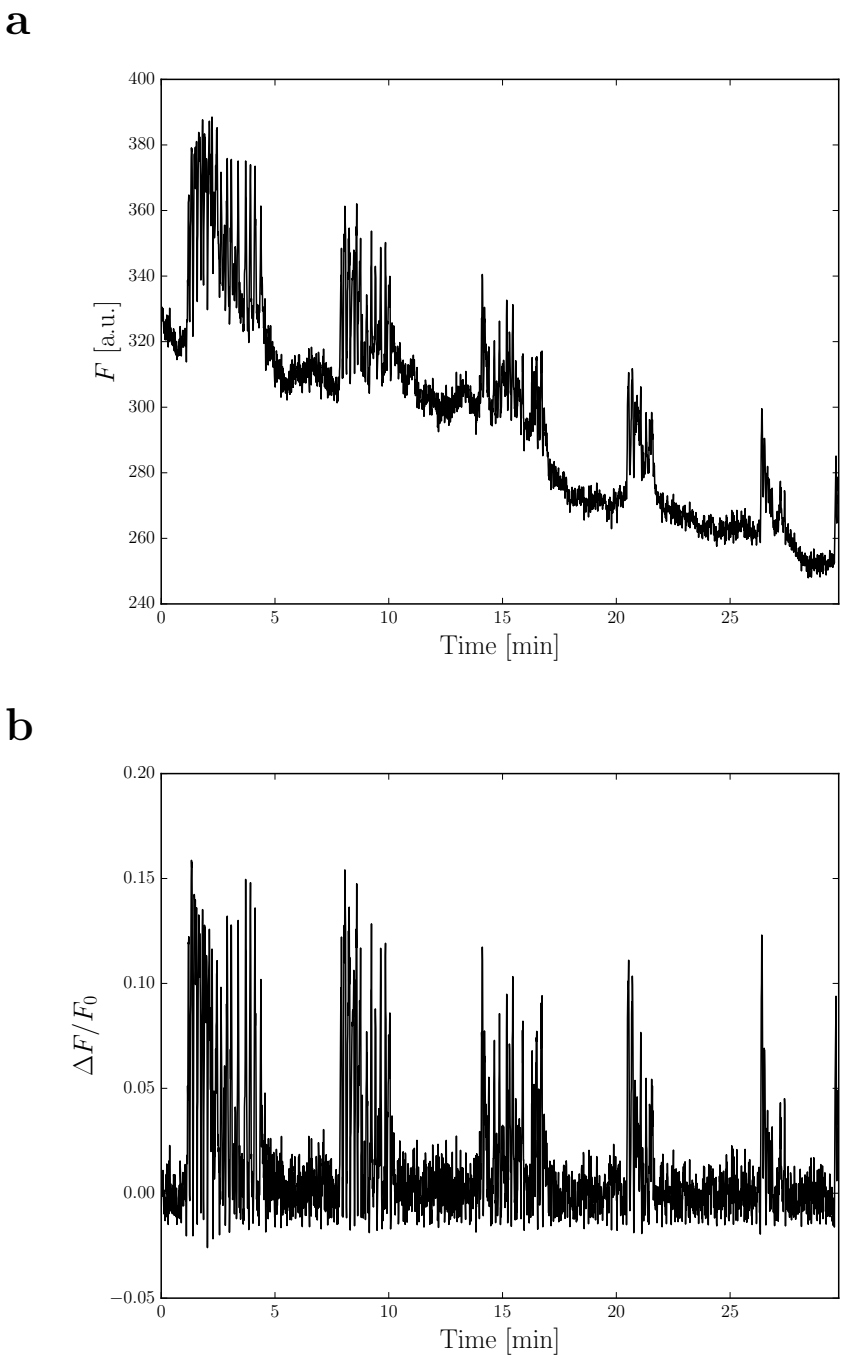


Figure 3.18: An example of the normalization. **(a)** A raw signal. **(b)** The normalized signal.

3.5 Circuit State Detection

Based on several previous reports and our visual inspection, *Drosophila* larvae show various behaviors including forward locomotion, backward locomotion, and turning. We focused on four kinds of activity patterns in the CNS corresponding to the forward locomotion, backward locomotion, left turning, and right turning.

Drosophila larvae have eight abdominal segments and there are corresponding neuromeres in the CNS. During the forward locomotion, the larvae contract the muscles of the abdominal segments of the body wall from the tail (A8) to the head (A1). Since the contraction pattern of the muscles is generated by the CNS and the motor neurons in each neuromere of the CNS innervate muscles in the corresponding segment of the body wall, the CNS activity pattern is similar to the body muscle contraction pattern at the macroscopic level [9].

For this reason, we named the activity pattern which occurs during the forward locomotion as *forward wave* and defined 8 active states (F0 $\sim$ F7) and one idle state (FQ) for the forward wave. Each state in the 8 active states corresponds to the activity concentrated on each neuromere of the CNS. For example, the first state of the forward wave F0 indicates the activity state corresponding the muscle contraction in the segment A8 of the body wall,

and the last state F7 indicates the activity state corresponding the muscle contraction in the segment A1.

In the case of the backward locomotion, we defined 8 active states (B0 $\sim$ B7) and one idle state (BQ) for *backward wave*, as we did for the forward locomotion. The difference in forward locomotion is, the first state of the backward wave B0 corresponds to the contraction of the muscles in the segment A1 of the body wall because the contraction starts from the tail in the backward locomotion.

The direction of the turning of the larva could be left or right. Therefore we defined two activity patterns, TL and TR. TL, which indicates *turning left*, consists of 3 active states (TL0 $\sim$ TL2) and 1 idle state (TLQ). The 3 active states correspond to the activity patterns which cause the contraction of the muscle in the anterior left side (A1 $\sim$ A3). TR, *turning right*, consists of 4 states (TR0 $\sim$ TR2, and TRQ) like the TL, but is related to turning in the opposite (right) direction.

Finally, we defined a steady idle state (SQ) which is not directly related to the motor outputs that generate behaviors. Thus, activity states of the CNS are SQ, FQ, F0 $\sim$ F7, BQ, B0 $\sim$ B7, TLQ, TL0 $\sim$ TL2, TRQ, TR0 $\sim$ TR2, totaling 27 states.

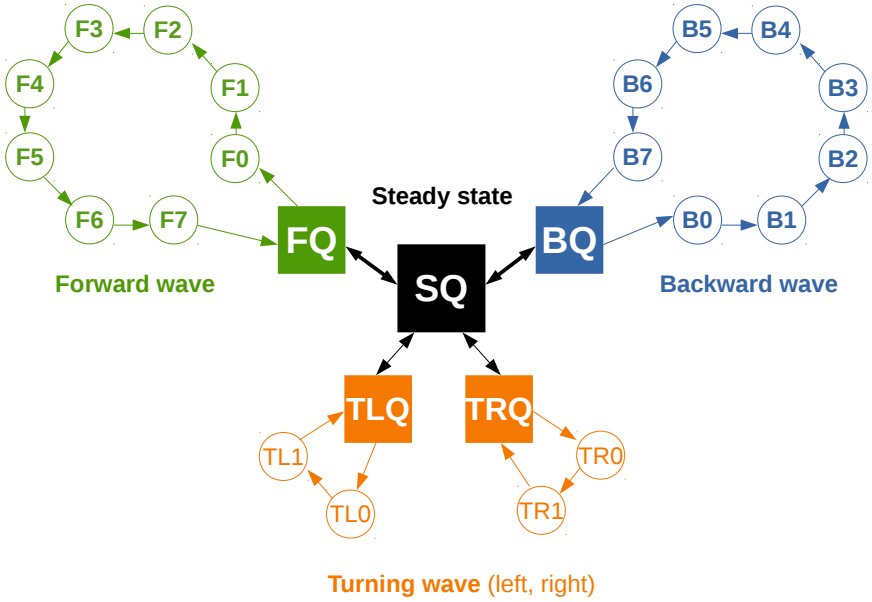


Figure 3.19: The states of the CNS related to the motor outputs.

To determine the circuit state at each time frame, we applied hidden Markov model (Section 2.4.1) whose hidden states are the above 27 states. The information about the locomotions we have is the pattern of the muscle contractions. Hence we extracted the signals of all pairs (left and right portion) in neuromeres A1 $\sim$ A8. And we binarized the 16 signals so each observation would fall into $0 \sim 2^{16}$ $(= 65536)$.

3.5.1 Application

In order to extract activity dynamics of the circuit as a whole, we applied principal component analysis (PCA) to the normalized signals of the neurons. In the trajectory in a 3-dimensional space obtained by PCA, we found two different circular trajectories. It suggests that there are at least two different periodic activity patterns in the circuit. We confirmed the two trajectories correspond to the forward wave and the backward wave by visual inspection of the movie.

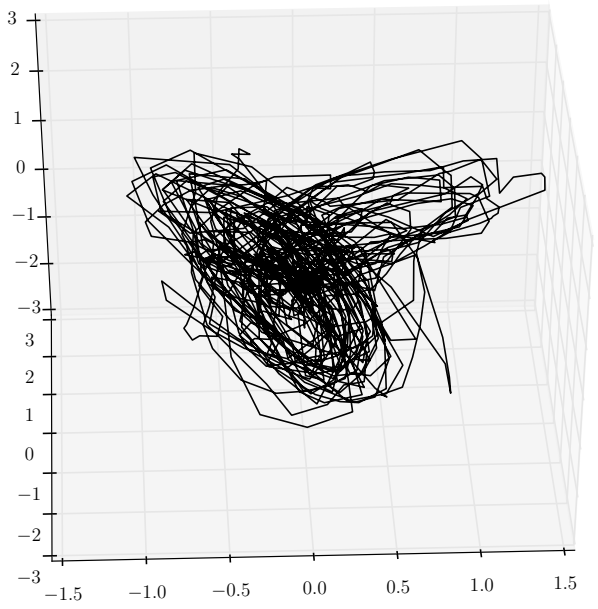


Figure 3.20: The result of principal component analysis (PCA) on the normalized signals. Two different circular trajectories are visible.

In order to investigate the change of the state of the circuit in detail, the hidden Markov model was applied to the normalized signals. First, we obtained the parameters (the transition matrix and the emission matrix, refer to Section 2.4.1) reflecting the calcium imaging movie from the initial parameters using the Baum-Welch algorithm. By calculating the Viterbi path (Figure 3.21) from the obtained parameters, we were able to obtain the state of the circuit at each time.

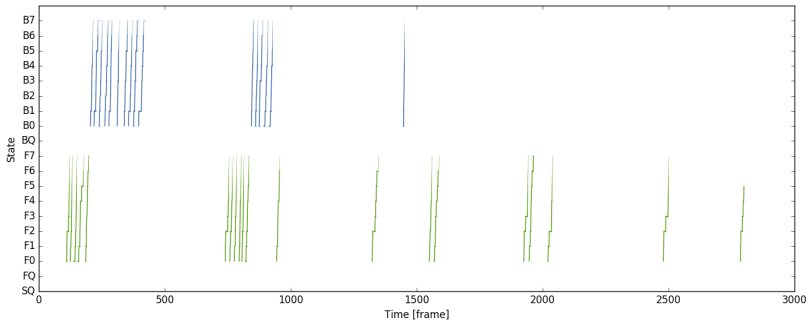


Figure 3.21: The Viterbi path of the normalized signals. The green lines indicate the forward waves, and the blue lines indicate the backward waves.

3.6 Activity Profiles of Cells

3.6.1 Motor Activity-dependent Neurons

Since we were able to detect the state of the circuit at each time, we used that information to investigate the activity profiles of the cells. The states defined

in this study (Section 3.5) related to the larval movements, so we could find cells whose activity depends on the state of the circuit.

First, we focused on the two frequent movement states, forward wave and backward wave. Since almost all muscles are used during the forward locomotion and the backward locomotion, many motor neurons are active at both the forward locomotion and the backward locomotion. However, the activity of the interneurons could be different between the two locomotions, and we wanted to find those interneurons.

To measure the difference in the activities of a neuron during the forward wave and the backward wave, we created a mask m_{F-B} (Figure 3.22a) as the following definition:

$$m_{F-B}(t) = \begin{cases} +1 & \text{for } t \in \bigcup_i F_i \\ -1 & \text{for } t \in \bigcup_i B_i \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

where F_i is a set containing the time frames of the state Fi, and B_i is a set containing the time frames of the state Bi. Then we calculate a score $s_{F-B}(i)$ of neuron i by dotting (inner product) the normalized signal of the neuron and the mask. However, the magnitude of the changes in intensity of the

neurons are all different, so we created lots of shuffled signals (Figure 3.22c) to each signal (Figure 3.22b) of neuron and the score was normalized using the distribution of scores of the shuffled signals.

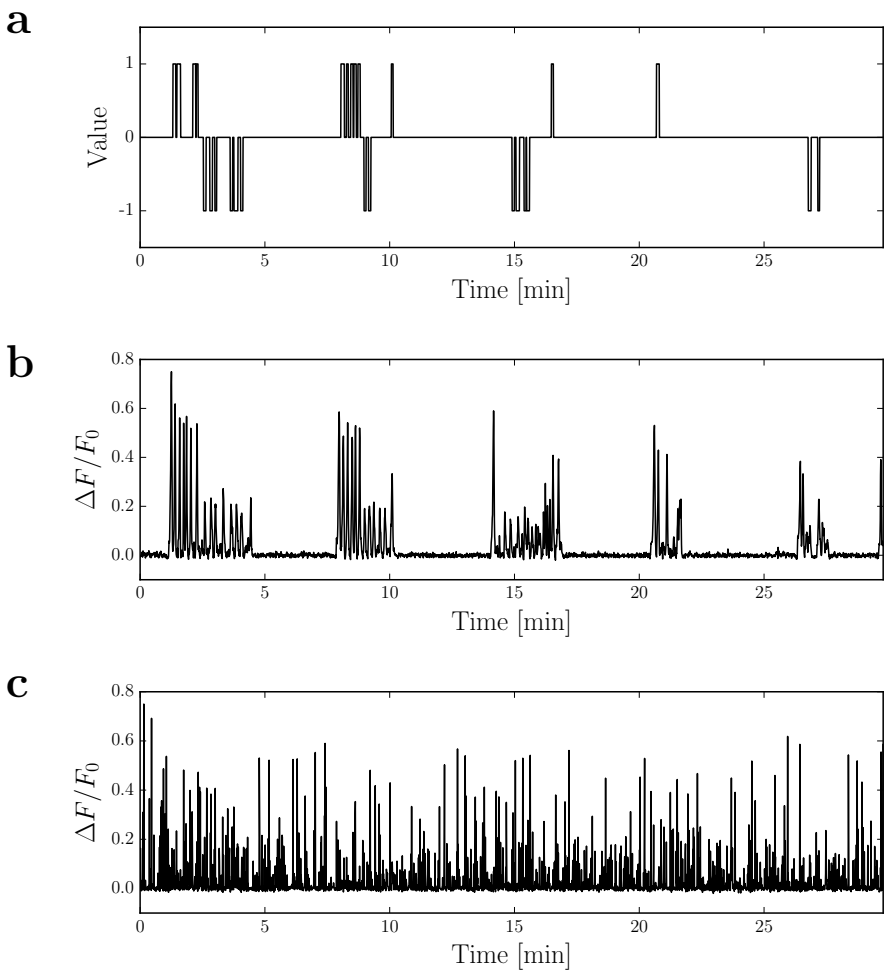


Figure 3.22: Calculating the score (forward wave - backward wave) on each neuron. **(a)** The mask created as Equation (3.9). **(b)** A signal of a neuron. **(c)** An example of the shuffled signal from the signal of the neuron.

Then, we obtained the distribution of the scores (Figure 3.23) for all the neurons. We found that the distribution is not a Gaussian distribution since it has large tails on both sides. Hence, the distribution is not a form of Figure 3.24a, which has one center of the Gaussian curve. We concluded the distribution of the scores was a form of Figure 3.24c, having the parameter $m_0 \approx 0$. Since the heights of the curves at m_F and m_B would be small, the population of the activity specific neurons also would be a small fraction.

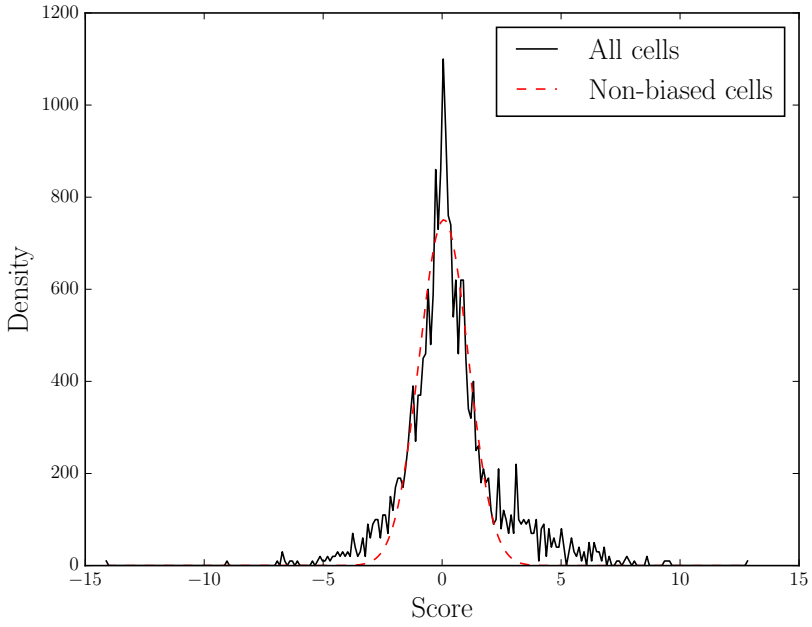


Figure 3.23: The distribution of the scores. The red line is the estimated gaussian distribution of the non-biased cells. The two large tails are visible, which correspond to the populations of the activity dependent cells.

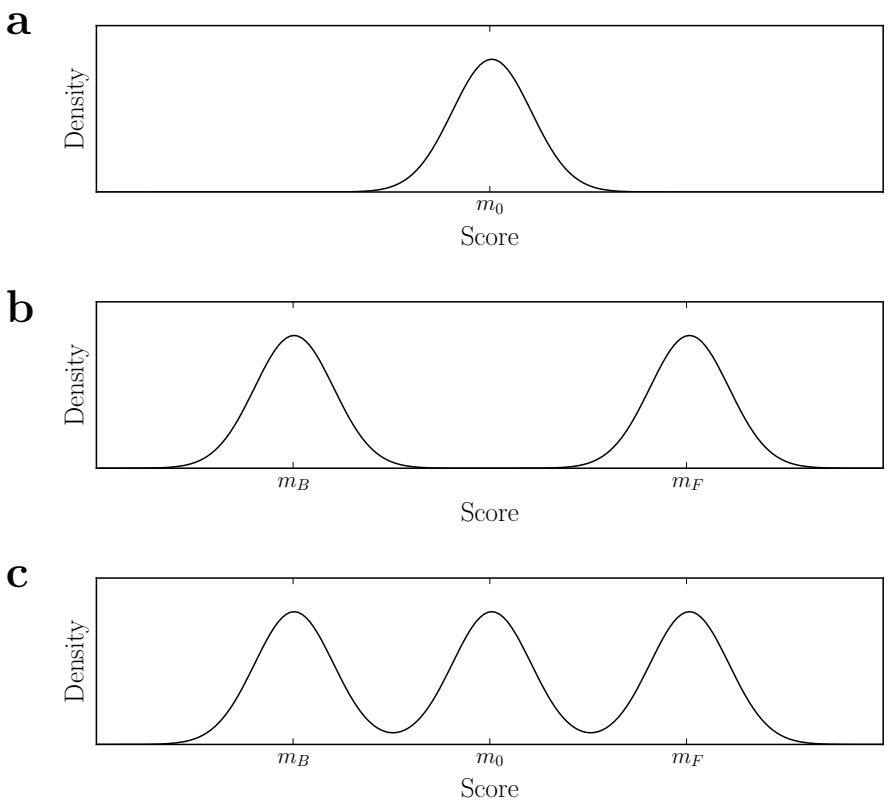


Figure 3.24: Examples of possible distributions of the scores. **(a)** A distribution has only one center of the Gaussian distribution (only non-biased cells). **(b)** A distribution has two centers of the Gaussian distributions (forward wave specific neurons and backward wave specific neurons). **(c)** A distribution has three centers of the Gaussian distributions (forward wave specific neurons, non-biased neurons, and backward wave specific neurons).

Based on this hypothesis, we performed a fitting of the distribution to a Gaussian curve (Figure 3.23) to estimate the distribution of cells not biased on the circuit states. Then we estimated the populations of the activity dependent neurons from the fitting. We assumed that activity dependent neurons would have scores in the outside of the fitted curve. So we calculated the number of the forward wave specific neurons (n_F) which have their own scores lower than -3σ of the fitted curve, and the number of the backward wave specific neurons (n_B) which have their own scores higher than 3σ of the fitted curve. Letting the number of all the neurons be N, the fraction of the forward wave specific neurons n_F/N was 9.8 % and the fraction of the backward wave specific neurons n_B/N was 3.4 % (Figure 3.25).

Also, we divided the larval CNS into two regions, *posterior CNS* which consist of the abdominal neuromeres and *anterior CNS* which consist of the regions other than the abdominal neuromeres. Letting the number of the cells in the posterior CNS be N^{post}, the fraction of the forward wave specific neurons in the posterior CNS $n_F^{\mathrm{post}}/N^{\mathrm{post}}$ was 14 % and the fraction of the backward wave specific neurons in the posteior CNS $n_B^{\mathrm{post}}/N^{\mathrm{post}}$ was 5.0 % (Figure 3.25). The similar statistics were obtained for the cells in the anterior CNS, resulting in $n_F^{\mathrm{ant}}/N^{\mathrm{ant}} = 2.5$ % and $n_B^{\mathrm{ant}}/N^{\mathrm{ant}} = 0.94$ % (Figure 3.25).

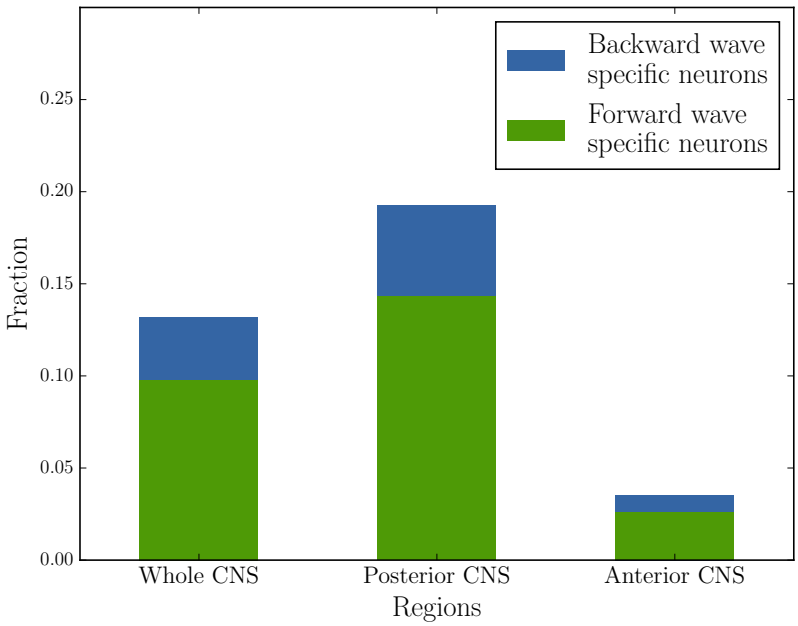


Figure 3.25: The populations of the circuit state dependent neurons. The blue bars indicate the fractions of the backward wave specific neurons, and the green bars indicate the fractions of the forward wave specific neurons.

Forward Wave Specific Neurons

To find the spatial distribution of the forward wave specific neurons in the larval CNS, we selected the neurons which have score s_{F-B} higher than a threshold (3σ of the distribution). Their spatial distribution in the CNS was as Figure 3.26.

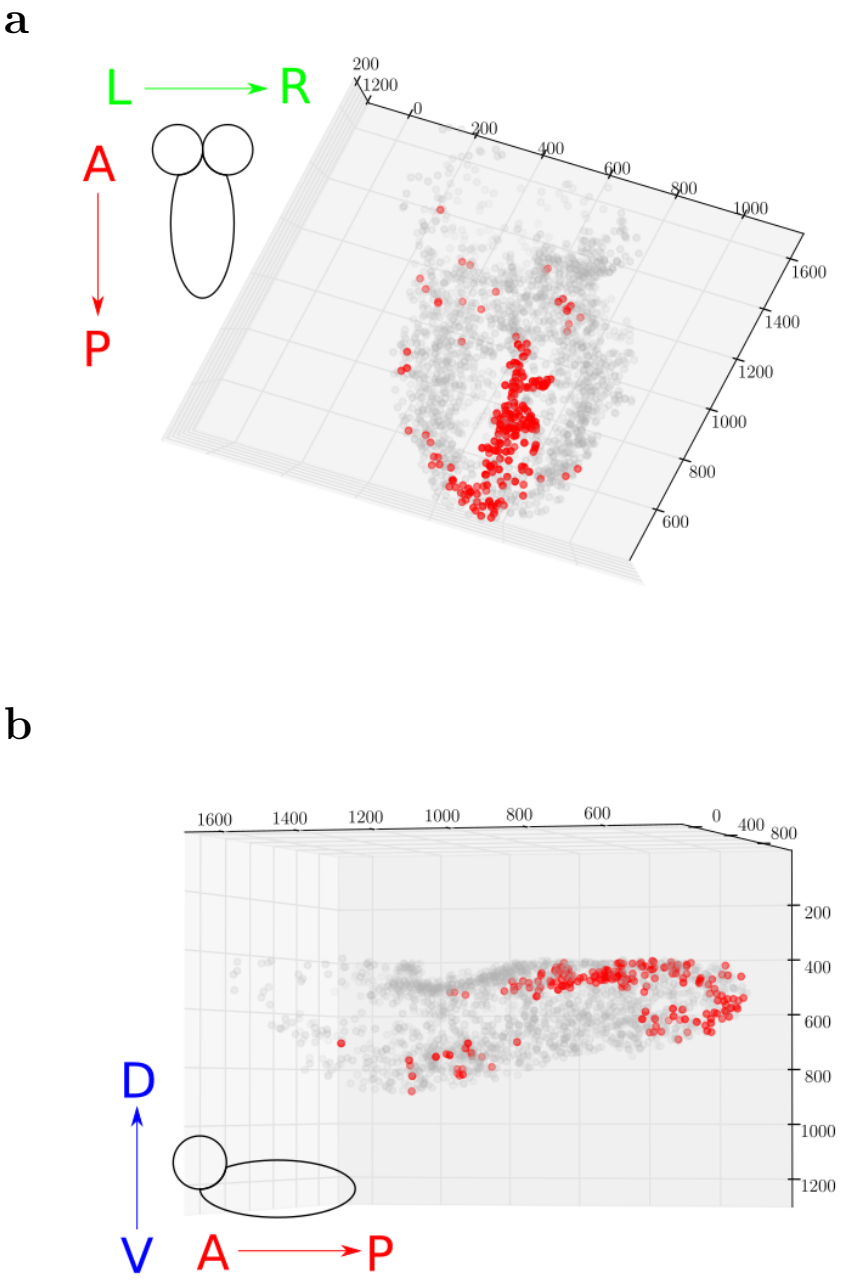
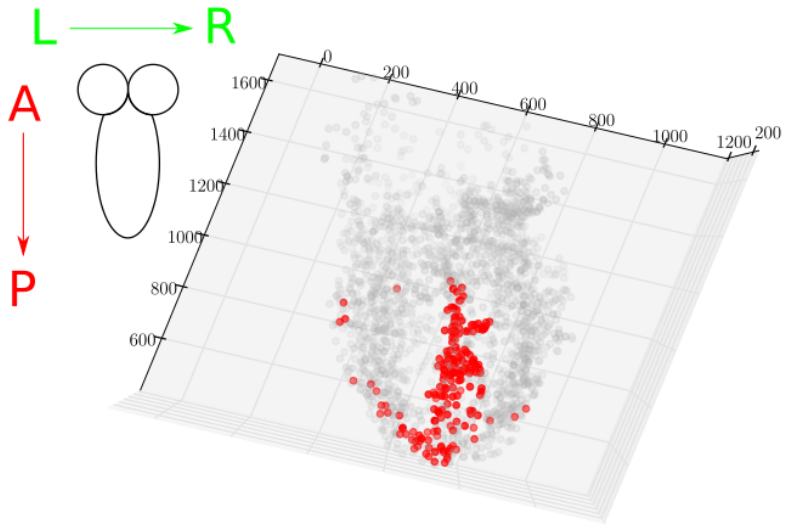


Figure 3.26: Neurons active biased during the forward wave. **(a)** Dorsal view. **(b)** Lateral view.

To investigate the activity timing of the forward wave specific neurons we found, we created a profile of the neurons, which have the average value of the normalized signal in each circuit state. First, we focus on the posterior CNS (Figure 3.27). Figure 3.28a suggests that the forward wave specific neurons in the posterior CNS show activities with a similar spatio-temporal pattern to the forward wave, the cells close to the tail activated early and the cells close to the head activated later.

a



b

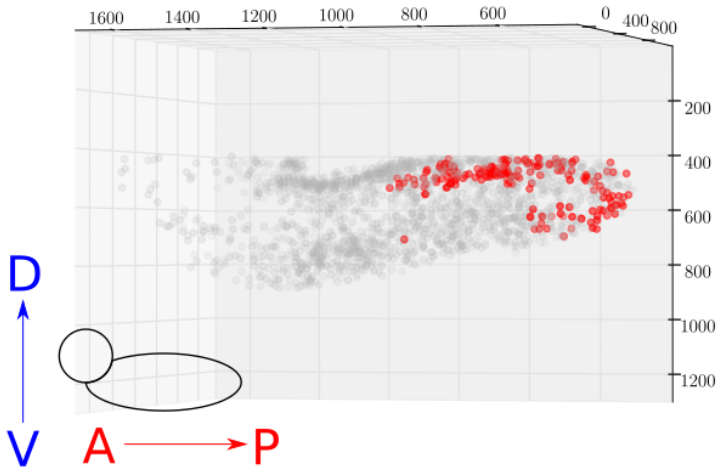


Figure 3.27: The distribution of the forward wave specific neurons in the posterior CNS. **(a)** Dorsal view. **(b)** Lateral view.

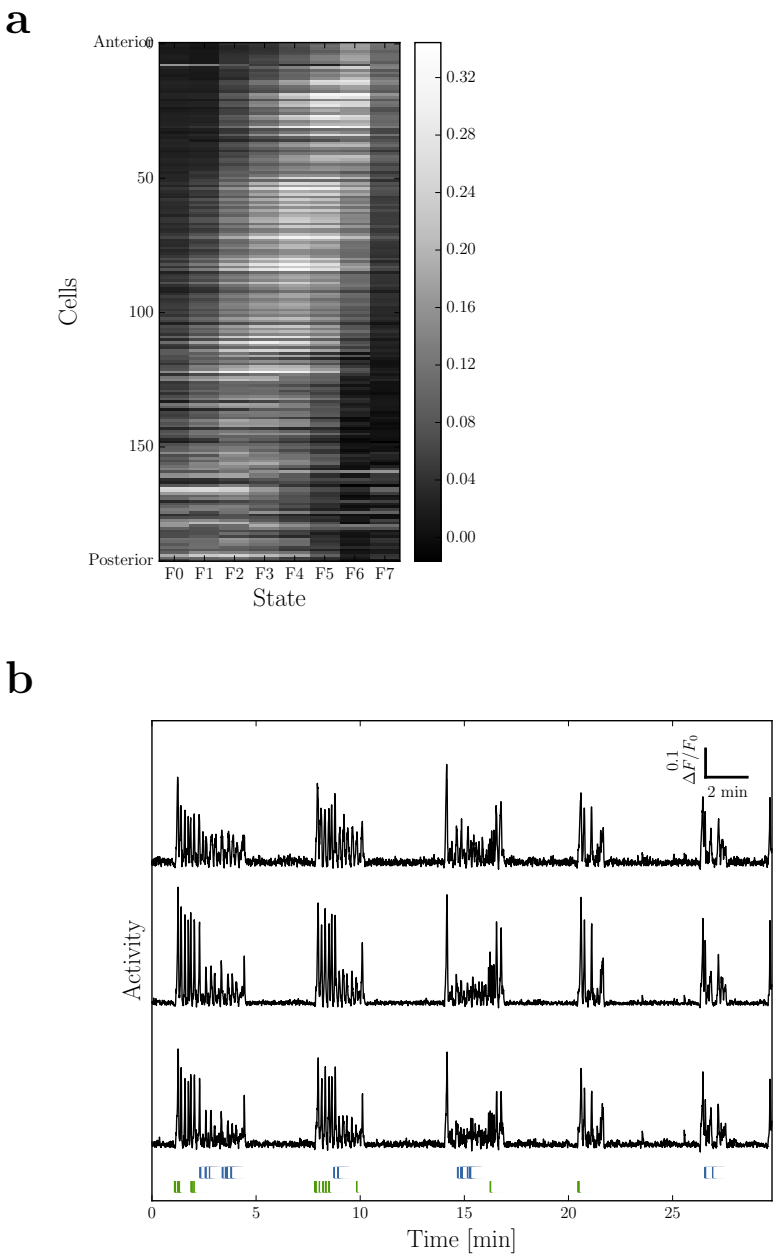


Figure 3.28: Activities of the forward wave specific neurons in the posterior CNS. **(a)** The profiles of the neurons. We sorted the cell numbers so that the bigger the number the closer it would be to the tail. **(b)** Examples of the normalized signals of the neurons. The estimated circuit state trace indicated below the signals (green: the forward wave, blue: the backward wave).

Next, we focus on the anterior CNS (Figure 3.29). By calculating the activity profile (Figure 3.30a), we found that the cells active in state F0 is included in the anterior CNS. Since there would be no motor neurons active at state F0 in the anterior CNS, these neurons would not directly connect to motor neurons. Since the neurons are activated at the initial phase of the forward wave, they are likely related to the initiation of the forward waves. Using the activity profile (Figure 3.30a), we found the candidates which have maximum activity in state F0 (Figure 3.31).

In the forward wave, the motor activity starts from the segment A8 and propagates to A1. Hence by comparing the activity timings of candidates and that of the motor signal in A8, we can double check whether the candidate neurons are activated prior to the forward wave. And Figure 3.31b shows that the candidates are indeed activated in the beginning of the forward wave.

Finally, we obtained the spatial distribution of the candidates (Figure 3.32). The fraction of the candidates out of the forward wave specific neurons in the anterior CNS $n_{\mathrm{cand}}/n_F^{\mathrm{ant}}$ was 59 %.

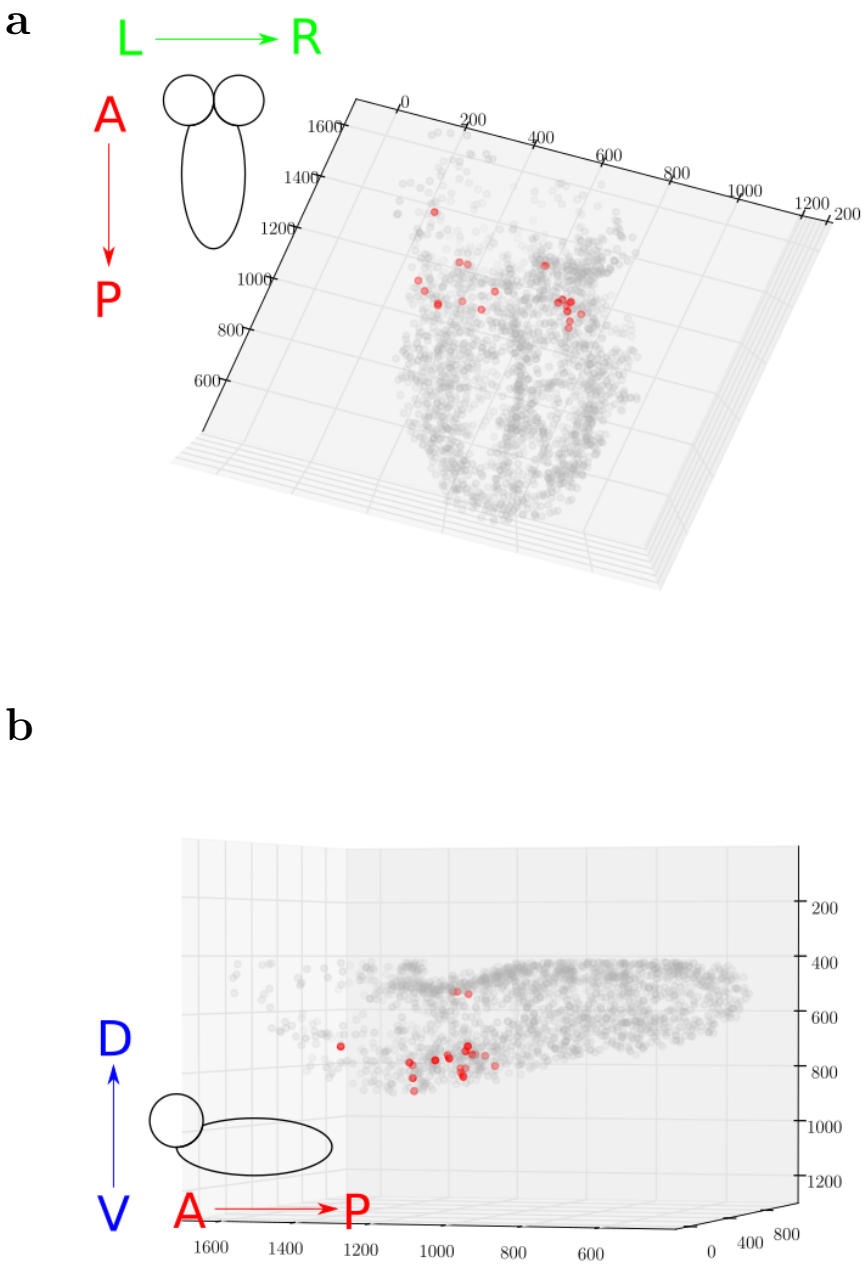


Figure 3.29: The distribution of the forward wave specific neurons in the anterior CNS. **(a)** Dorsal view. **(b)** Lateral view.

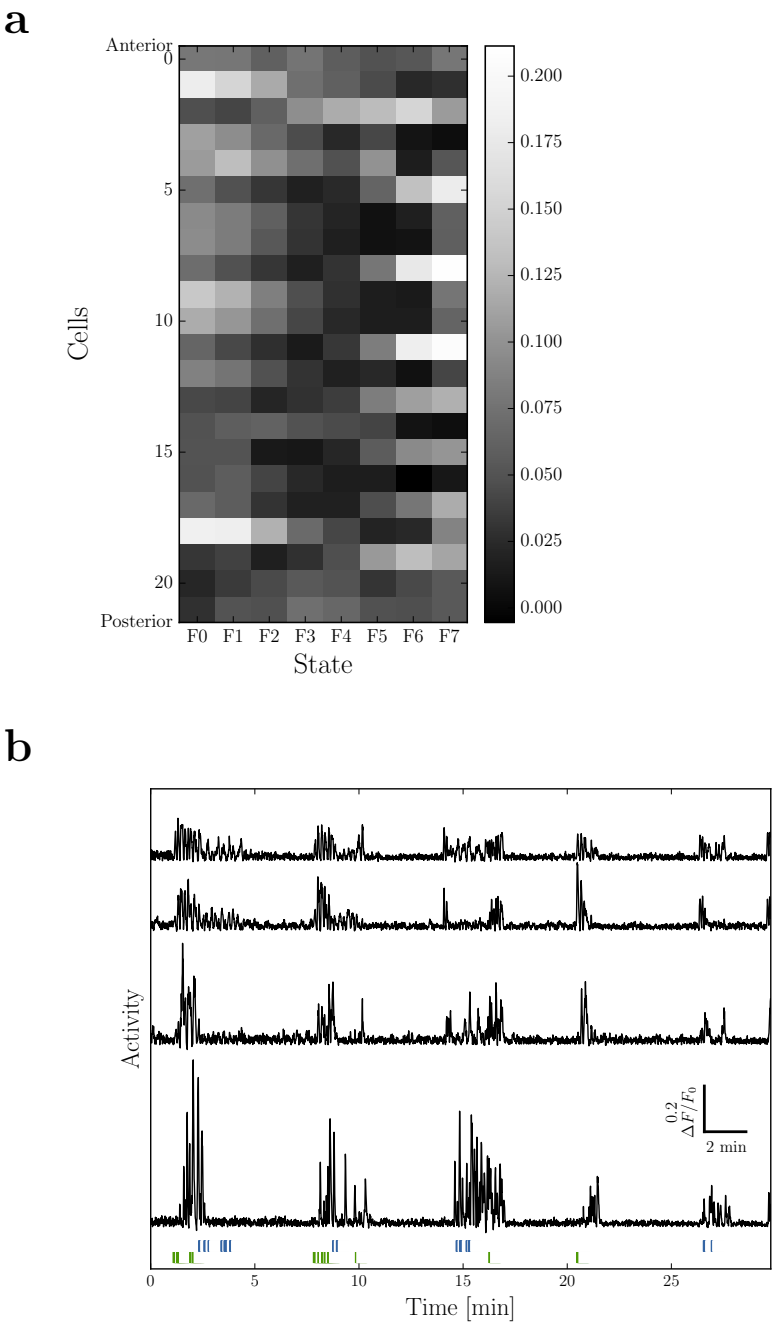
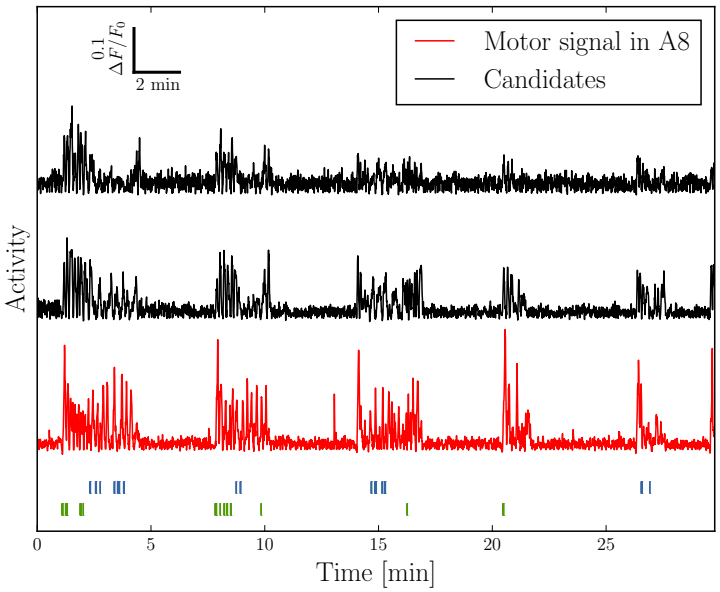


Figure 3.30: Activities of the forward wave specific neurons in the anterior CNS. **(a)** The profiles of the neurons. We sorted the cell numbers so that the bigger the number the closer it would be to the tail. **(b)** Examples of the normalized signals of the neurons. The estimated circuit state trace indicated below the signals (green: the forward wave, blue: the backward wave).

a



b

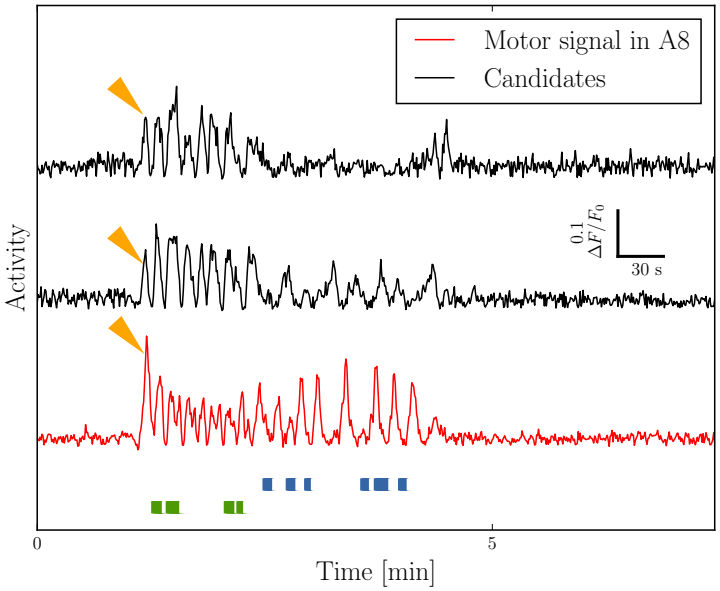


Figure 3.31: Candidates of the forward wave triggering neurons. The red trace is the normalized motor signal of the segment A8. Their activity timings are almost identical (see the arrows, the candidates precede 1 frame or 0.638 s). The estimated circuit state trace indicated below the signals (green: the forward wave, blue: the backward wave). **(a)** The signals of the candidates and that of the motor signal in A8. **(b)** An enlarged view of the time range from 0 min to 7 min.

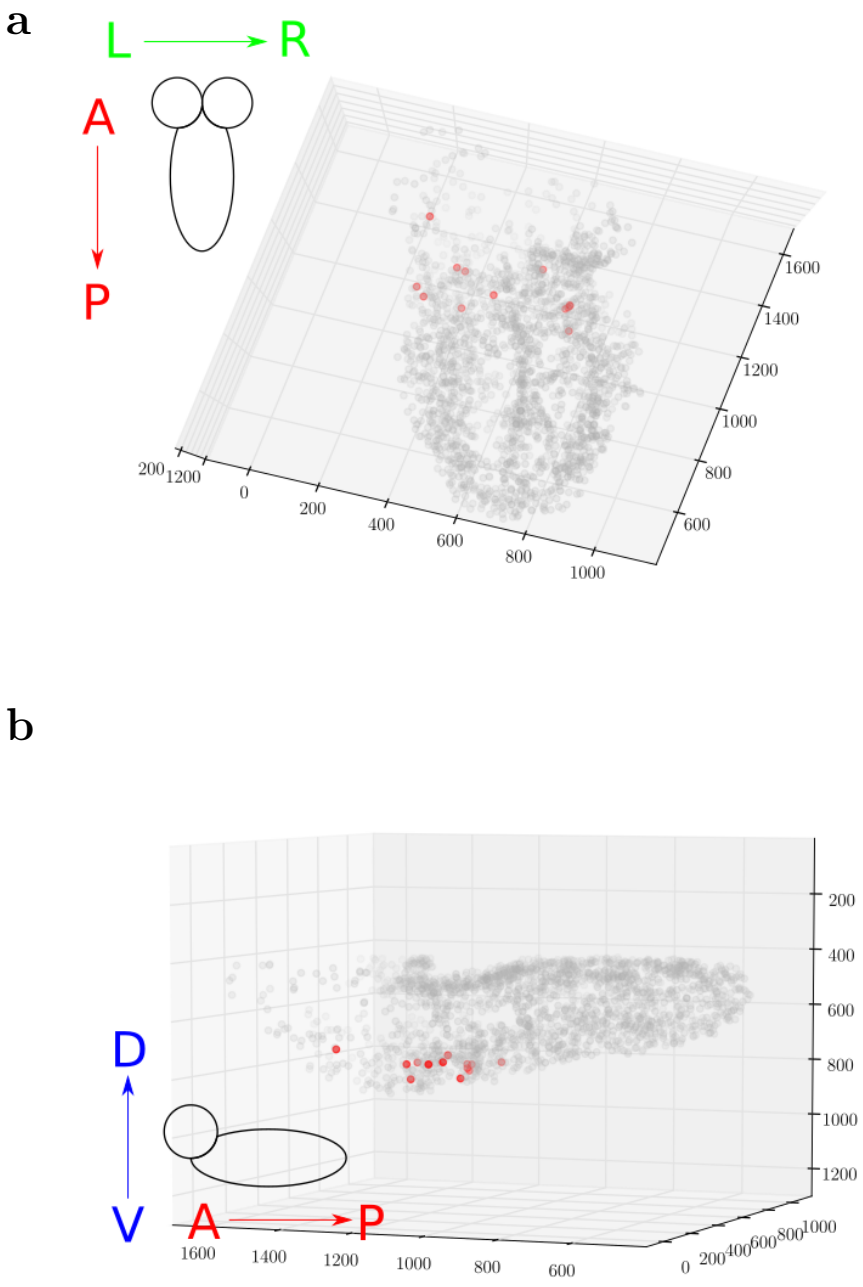


Figure 3.32: The distribution of the candidates of forward wave triggering neurons. **(a)** Dorsal view. **(b)** Lateral view.

Backward Wave Specific Neurons

To find the spatial distribution of the backward wave specific neurons in the larval CNS, we selected the neurons which have score s_{F-B} lower than a threshold (-3σ of the distribution). Their spatial distribution in the CNS was as Figure 3.33.

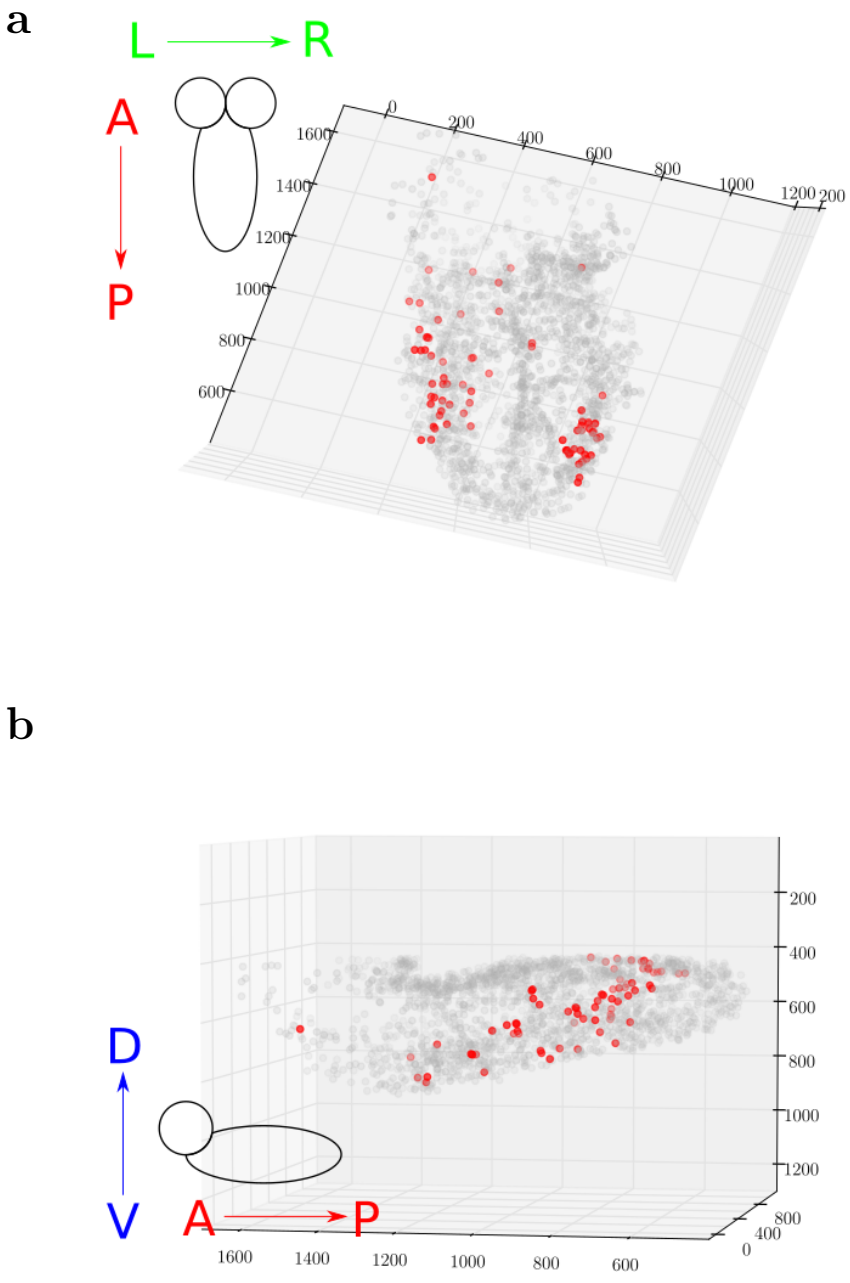
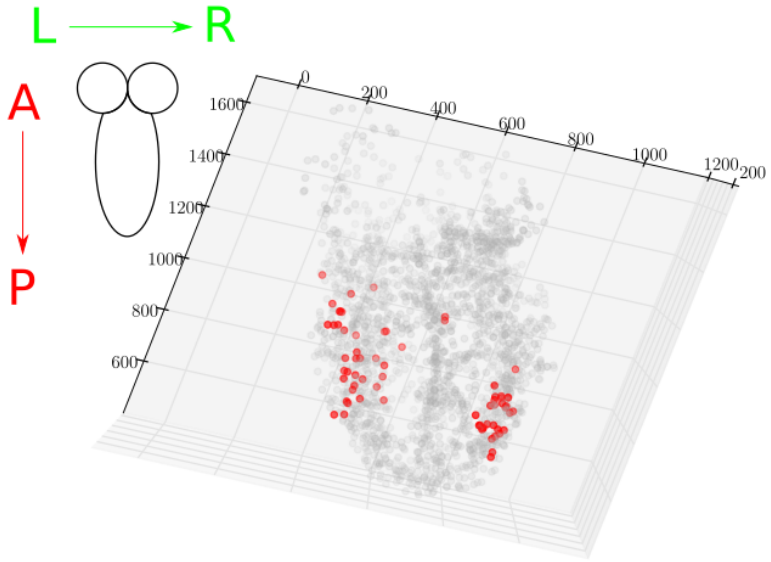


Figure 3.33: Neurons active biased during the backward wave. **(a)** Ventral view. **(b)** Lateral view.

To investigate the activity timing of the backward wave specific neurons we found, we created a profile of the neuron, which have the average value of the normalized signal in each circuit state (Figure 3.34a). Figure 3.35a suggests that the backward wave specific neurons in the VNC show activities with a similar spatio-temporal pattern to the backward wave, the cells close to the tail activated early and the cells close to the head actived later. Therefore, the cells we found here could be part of the backward wave circuitry.

a



b

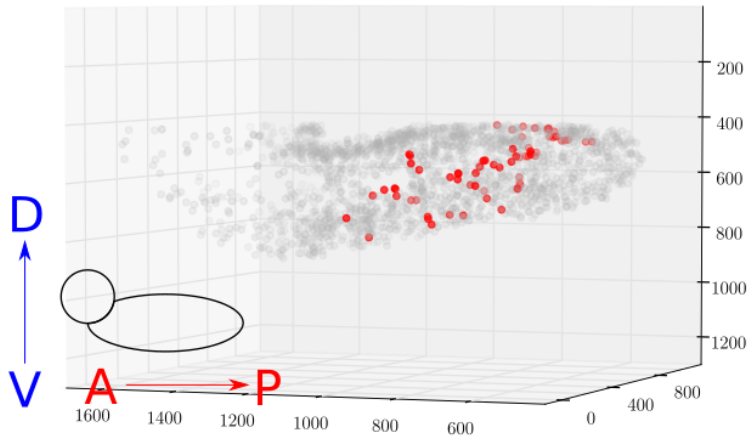


Figure 3.34: The distribution of the backward wave specific neurons in the posterior CNS. **(a)** Dorsal view. **(b)** Lateral view.

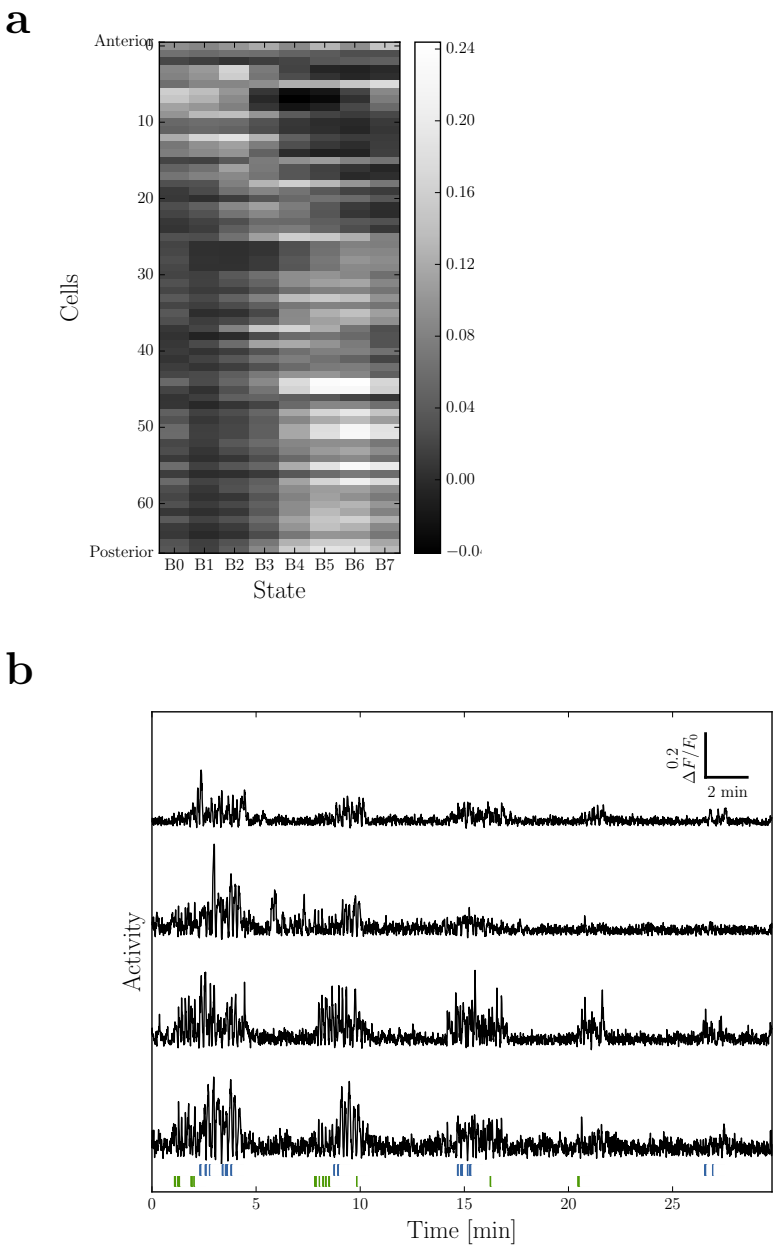


Figure 3.35: Activities of the backward wave specific neurons. **(a)** The profiles of the neurons. We sorted the cell numbers so that the bigger the number the closer it would be to the tail. **(b)** Examples of the normalized signal of the backward wave specific neurons. The estimated circuit state trace indicated below the signals (green: the forward wave, blue: the backward wave).

Chapter 4

Discussion

In order to investigate the neural circuits that produce multiple behavioral patterns, this study extracted and analyzed the activities of thousands of neurons contained in the *Drosophila* larval CNS. To measure the activities from a large number of neurons spreading in a 3-dimensional space, we used a light sheet microscopy capable of recording a wide range of images at a fast frame rate, and the genetically modified larvae expressing GCaMP6f proteins in all the neurons in the CNS.

Since the samples in the calcium imaging movies were moving, the registration was required before extracting the signals. To suppress the influence of photobleaching and the fluctuation due to neural activities, we decided to

estimate translation by detecting a large number of feature points in the movie and tracked them using normalized cross-correlation. Tracking a large number of feature points, we could ignore feature points which fluctuate due to the neural activities. And using normalized cross-correlation, we were able to compare feature points between images distant temporally with each other.

Since *Drosophila* larval CNS contains ten thousand of neurons, manually deciding the positions of the neurons leads to waste of time and labor and a decrease in objectivity. Detecting the neurons automatically, we were able to decide the position of the neurons in a relatively short time, and the result is guaranteed to be the same each time.

To extract the neural activity from the sequence of the fluorescence obtained, we normalized the fluorescence signals assuming the time scale of the neural activity would be small compared to that of the photobleaching (Figure 3.18). Comparing with using the Gaussian blurred signal as the baseline, we could suppress false negative changes especially noticeable at the start and end points of activity series. More accurate normalization allowed us to calculate quantities intuitively and more accurately.

To investigate the activity of the circuit roughly, we used the principal component analysis (PCA) to reduce dimensionality. Applying the PCA on

all the neural activities of the CNS, we could confirm there are at least two different periodic activities at the circuit level (Figure 3.20). We confirmed the two activities correspond to the forward wave and the backward wave, which are related to the forward locomotion and the backward locomotion, respectively.

Using the hidden Markov model, we could detect the states of the larval CNS (Figure 3.21). And using the information of the state in each time frame, we found the neurons whose activity is biased toward the forward or backward wave state (Figures 3.26 and 3.33). Both forward-biased and backward-biased neurons existed not only in the anterior CNS but also in the posterior CNS (Figures 3.27 and 3.34). It suggests that the circuits responsible for the forward locomotion and the backward locomotion are scattered in the VNC. Such system will produce more stable motor patterns than a case in which the circuit is present only in a decision making layer in the brain.

And we compared the number of the forward wave biased neurons n_F and the number of the backward wave biased neurons n_B. We found that n_F was much larger than n_B in both the anterior CNS and the posterior CNS (Figures 3.23 and 3.25).

The distribution of the forward wave biased neurons was concentrated in a

dorsal region of the VNC (Figure 3.26). In the dorsal region, however, almost all neurons are motor neurons. It suggests that the activity intensity of the motor neurons is much stronger in the forward wave. The result was not an numerical error, since the backward wave biased neurons were not found in the dorsal region (Figure 3.33).

We also found F0 phase neurons in the anterior CNS region (Figure 3.32). The activity of these neurons preceded the forward waves (Figure 3.31). Their position and activity timing imply that these neurons trigger the initiation of the forward wave at the posterior-end segment through direct or indirect descending axon projections. To confirm the neurons would trigger the forward wave, we need further experiments that would clarify the causality.

While this work was in progress, a similar study on *Drosophila* larval whole CNS functional imaging was published by Lemon et al. [20]. However, the functional unit in the study was voxel so one cannot estimate the population size (the number of neurons) and the activities obtained would be noisy. We detected the position of the neurons so that we could set the ROIs on the somas. Also, we determined the circuit state by using hidden Markov model whose parameters are learned from the data. Hence we could assign a certain circuit state to each time frame, which cannot be accomplished from visual

inspection. The differences between our study and Lemon's are summarized in Table 4.1.

Table 4.1: Comparison between this study and the previous study.

	Our study	Lemon et al.
Neurons	All	
Microscopy	Light-sheet	
Temporal resolution	2 Hz	5 Hz
Analysis	Quantitative	Qualitative
Functional unit (ROI)	Neuron	Voxel
Circuit state definition	Statistical analysis	Visual inspection

The methods we developed in this study enabled us to obtain the activity profiles of neurons in *Drosophila* larval CNS which generates various behaviors including the forward locomotion and the backward locomotion. This map of the activity profiles provide us a clue for distinguishing the circuits corresponding to the distinct behaviors. By combining the results with optogenetics, we would be able to find the neurons commanding the activity patterns in the circuit.

Acknowledgement

This research was conducted as a collaborative research with Shin Ishii and Ken Nakae of the Graduate School of Informatics, Kyoto University and Shigenori Nonaka and Atsushi Taniguchi of National Institute for Basic Biology. I would like to thank Shin Ishii and Ken Nakae for guidance on analysis method, discussion, and other generous cooperation. Also, I would like to thank Shigenori Nonaka and Atsushi Taniguchi that they willingly allowed me to use the experimental system and helped the experiments.

I would like to express my deepest appreciation to Akinao Nose who gave me guidance in every aspect of this research. I also deeply appreciate Hiroshi Kohsaka who gave me detailed advice on the whole experiment and analysis. Everyone in Nose lab who supported various opinions throughout the research - Etsuko Takasu, Eri Hasegawa, Kaoru Masuyama, Satoko Okusawa, Yuki Itakura, Akira Fushiki, Teruyuki Matsunaga, Shunsuke Takagi, Koichi Teranishi, Tappei Kawasaki, Keisuke Ban, Ryota Mori, Dohjin Miyamoto, Yoshiki Maruta, Yumi Sakamaki, Suguru Takagi, Hitoshi Maruo, Yasuhide Lee, Jeonghyuk Park, Atsuki Hiramoto, Tatsuya Takatori, Maki Kusano, Shoya Ohura, Yuji Matsuo, and Xiangsunze Zeng. I would like to thank

Sawako Niki, Toshie Naoi, Kasumi Shibahara for their efforts to maintain the experimental environment.

Bibliography

[1] Misha B Ahrens, Michael B Orger, Drew N Robson, Jennifer M Li, and Philipp J Keller. Whole-brain functional imaging at cellular resolution using light-sheet microscopy. *Nature methods*, 10(5):413–420, 2013.

[2] Jasper Akerboom, Tsai-Wen Chen, Trevor J Wardill, Lin Tian, Jonathan S Marvin, Sevinç Mutlu, Nicole Carreras Calderón, Federico Esposti, Bart G Borghuis, Xiaonan Richard Sun, et al. Optimization of a gcamp calcium indicator for neural activity imaging. *The Journal of neuroscience*, 32(40):13819–13840, 2012.

[3] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[4] Leonard E Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The annals of mathematical statistics*, 41(1):164–171, 1970.

[5] Jimena Berni, Stefan R Pulver, Leslie C Griffith, and Michael Bate. Autonomous circuitry for substrate exploration in freely moving drosophila larvae. *Current Biology*, 22(20):1861–1870, 2012.

[6] Andrea H Brand and Norbert Perrimon. Targeted gene expression as a means of altering cell fates and generating dominant phenotypes. *development*, 118(2):401–415, 1993.

[7] Tsai-Wen Chen, Trevor J Wardill, Yi Sun, Stefan R Pulver, Sabine L Renninger, Amy Baohan, Eric R Schreiter, Rex A Kerr, Michael B Orger, Vivek Jayaraman, et al. Ultrasensitive fluorescent proteins for imaging neuronal activity. *Nature*, 499(7458):295–300, 2013.

[8] Richard O Duda, Peter E Hart, et al. *Pattern classification and scene analysis*, volume 3. Wiley New York, 1973.

[9] Lyle E Fox, David R Soll, and Chun-Fang Wu. Coordination and modulation of locomotion pattern generators in drosophila larvae: effects of altered biogenic amine levels by the tyramine β hydroxlyase mutation. *The Journal of Neuroscience*, 26(5):1486–1498, 2006.

[10] Akira Fushiki, Maarten F Zwart, Hiroshi Kohsaka, Richard D Fetter, Albert Cardona, and Akinao Nose. A circuit mechanism for the propagation of waves of muscle contraction in drosophila. *Elife*, 5:e13253, 2016.

[11] CH Green, B Burnet, and KJ Connolly. Organization and patterns of inter-and intraspecific variation in the behaviour of drosophila larvae. *Animal Behaviour*, 31(1):282–291, 1983.

[12] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Citeseer, 1988.

[13] Eri Hasegawa, James W Truman, and Akinao Nose. Identification of excitatory premotor interneurons which regulate local muscle contraction during drosophila larval locomotion. *Scientific Reports*, 6, 2016.

[14] Ellie S Heckscher, Shawn R Lockery, and Chris Q Doe. Characterization of drosophila larval crawling at the level of organism, segment, and so-

matic body wall musculature. *The Journal of Neuroscience*, 32(36):12460–12471, 2012.

[15] Yuki Itakura, Hiroshi Kohsaka, Tomoko Ohyama, Marta Zlatic, Stefan R Pulver, and Akinao Nose. Identification of inhibitory premotor interneurons activated at a late phase in a motor cycle during drosophila larval locomotion. *PloS one*, 10(9):e0136660, 2015.

[16] Fredrik Johansson et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*, December 2013. `http://mpmath.org/`.

[17] Hiroshi Kohsaka, Satoko Okusawa, Yuki Itakura, Akira Fushiki, and Akinao Nose. Development of larval motor circuits in drosophila. *Development, growth & differentiation*, 54(3):408–419, 2012.

[18] Hiroshi Kohsaka, Etsuko Takasu, Takako Morimoto, and Akinao Nose. A group of segmental premotor interneurons regulates the speed of axial locomotion in drosophila larvae. *Current Biology*, 24(22):2632–2642, 2014.

[19] Subhaneil Lahiri, Konlin Shen, Mason Klein, Anji Tang, Elizabeth Kane, Marc Gershow, Paul Garrity, and Aravinthan DT Samuel. Two alternating motor programs drive navigation in drosophila larva. *PloS one*, 6(8):e23180, 2011.

[20] William C Lemon, Stefan R Pulver, Burkhard Höckendorf, Katie McDole, Kristin Branson, Jeremy Freeman, and Philipp J Keller. Whole-central nervous system functional imaging in larval drosophila. *Nature communications*, 6, 2015.

[21] JP Lewis. Fast normalized cross-correlation. In *Vision interface*, volume 10, pages 120–123, 1995.

[22] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[23] Eve Marder and Dirk Bucher. Understanding circuit dynamics using the stomatogastric nervous system of lobsters and crabs. *Annu. Rev. Physiol.*, 69:291–316, 2007.

[24] Junichi Nakai, Masamichi Ohkura, and Keiji Imoto. A high signal-to-noise ca2+ probe composed of a single green fluorescent protein. *Nature biotechnology*, 19(2):137–141, 2001.

[25] Robert Prevedel, Young-Gyu Yoon, Maximilian Hoffmann, Nikita Pak, Gordon Wetzstein, Saul Kato, Tina Schrödel, Ramesh Raskar, Manuel Zimmer, Edward S Boyden, et al. Simultaneous whole-animal 3d imaging of neuronal activity using light-field microscopy. *Nature methods*, 11(7):727–730, 2014.

[26] Irwin Sobel and Gary Feldman. A 3x3 isotropic gradient operator for image processing. *a talk at the Stanford Artificial Project in*, pages 271–272, 1968.

[27] Daisuke Takao, Atsushi Taniguchi, Takaaki Takeda, Seiji Sonobe, and Shigenori Nonaka. High-speed imaging of amoeboid movements using light-sheet microscopy. *PloS one*, 7(12):e50846, 2012.

[28] Lin Tian, S Andrew Hires, Tianyi Mao, Daniel Huber, M Eugenia Chiappe, Sreekanth H Chalasani, Leopoldo Petreanu, Jasper Akerboom, Sean A McKinney, Eric R Schreiter, et al. Imaging neural activity in worms, flies and mice with improved gcamp calcium indicators. *Nature methods*, 6(12):875–881, 2009.

[29] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967.

[30] Nikita Vladimirov, Yu Mu, Takashi Kawashima, Davis V Bennett, Chao-Tsung Yang, Loren L Looger, Philipp J Keller, Jeremy Freeman, and

Misha B Ahrens. Light-sheet functional imaging in fictively behaving zebrafish. *Nature methods*, 2014.

Appendix A

Numerical Data

A.1 Circuit State Detection

Indices 0 from 26 correspond to: SQ, FQ, F0, F1, F1, F3, F4, F5, F6, F7, BQ, B1, B2, B3, B4, B5, B6, B7, TLQ, TL0, TL1, TL2, TRQ, TR0, TR1, and TR2 respectively.

The seed of transition matrix is as follows.

transition[0, 0] = 0.96, transition[1, 0] = transition[10, 0] = transition[19, 0] = transition[23, 0] = 0.3.

transition[0, 1] = transition[0, 10] = transition[0, 19] = transition[0, 23] = 0.01.

transition[1:10, 1:10] =

$$\begin{bmatrix} 0.4 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.7 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.7 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.7 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.7 & 0.3 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.7 & 0.3 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.7 & 0.3 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.7 & 0.3 \\ 0.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.7 \end{bmatrix} \tag{A.1}$$

transition[10:19, 10:19] =

$$\begin{bmatrix} 0.4 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.7 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.7 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.7 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.7 & 0.3 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.7 & 0.3 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.7 & 0.3 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.7 & 0.3 \\ 0.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.7 \end{bmatrix} \tag{A.2}$$

transition[19:23, 19:23] =

$$\begin{bmatrix} 0.4 & 0.3 & 0.0 & 0.0 \\ 0.0 & 0.7 & 0.3 & 0.0 \\ 0.0 & 0.0 & 0.7 & 0.3 \\ 0.3 & 0.0 & 0.0 & 0.7 \end{bmatrix} \tag{A.3}$$

transition[23:27, 23:27] =

$$\begin{bmatrix} 0.4 & 0.3 & 0.0 & 0.0 \\ 0.0 & 0.7 & 0.3 & 0.0 \\ 0.0 & 0.0 & 0.7 & 0.3 \\ 0.3 & 0.0 & 0.0 & 0.7 \end{bmatrix} \tag{A.4}$$

The other elements of the transition matrix are set to 0.

Appendix B

Codes

Structure

1. image
    - io: image i/o wrapper
    - codec: image codecs
        - nifti: NIfTI (neuroimaging informatics technology initiative)
        - multitif: single multi-tif
        - singletifs: multiple single-tifs
    - registration: drift correction
    - filters: filters using OpenCL or multiprocessing

- esti: foreground estimation (CNS and neuropil), blob detection

2. mpmath: arbitrary precision floating-point arithmetic [16]
   - defmparray: mparray subclass
   - numeric: array creators
   - twodim_base: 2-dim specific creators
3. signal: normalization and binarization
4. models: hidden markov model

License

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.

3rd Party Addendum

==========

The files

- defmparray.py
- numeric.py
- twodim_base.py
- filters.py

are derived from NumPy, and are available under the Modified BSD License:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of the NumPy Developers nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

B.1 image.registration

```
#############################################################################
##
## Canal: Calcium imaging ANALyzer
##
## Copyright (C) 2015-2016 Youngtaek Yoon <caviargithub@gmail.com>
##
## This file is part of the source code of Canal.
##
## This program is free software: you can redistribute it and/or modify
## it under the terms of the GNU General Public License as published by
## the Free Software Foundation, either version 3 of the License, or
## (at your option) any later version.
##
## This program is distributed in the hope that it will be useful,
## but WITHOUT ANY WARRANTY; without even the implied warranty of
## MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
## GNU General Public License for more details.
##
## You should have received a copy of the GNU General Public License
## along with this program.  If not, see <http://www.gnu.org/licenses/>.
##
#############################################################################

import numpy as np
import multiprocessing as mp
import itertools        # in lattice_points
import skimage.feature  # in find_features
import tqdm
import scipy.ndimage.filters
import scipy.optimize
import scipy.spatial.distance

def concatenate(*args):
    return Concatenator(args)

class Concatenator:
    def __init__(self, args, ndim=None):
        if ndim is not None:
            for num, arg in enumerate(args):
                shape = (1,) * (ndim - arg.ndim) + arg.shape
                args[num] = arg.reshape(shape)

        self._data = args
```

```
    def _subindex(self, index):
        if not isinstance(index, int):
            raise ValueError('Index must be integer')

        lengths = [len(elem) for elem in self._data]
        ends = np.cumsum(lengths)
        for dindex, end in enumerate(ends):
            if index < end:
                return dindex, index - sum(lengths[:dindex])
        else:
            raise IndexError(('index {} is out of bounds for axis 0 ' +
                              'with size {}').format(index, end))

    @property
    def ndim(self):
        return self._data[0].ndim

    @property
    def dtype(self):
        return self._data[0].dtype

    def __len__(self):
        return sum([len(elem) for elem in self._data])

    def __getitem__(self, index):
        dindex, sindex = self._subindex(index)
        return self._data[dindex][sindex]

class MovieMapper:
    """
    Transforms movie.
    """
    def __init__(self, movie, offsets):
        n_times = len(movie)
        movie_vdim = movie.ndim - 1 # dimension of volume

        offsets = np.asarray(offsets, int)
        offset_vdim = offsets.shape[-1]
        if offset_vdim < movie_vdim:    # extend offset filling zeros
            extended = np.zeros((len(offsets), movie_vdim), int)
            extended[:, movie_vdim - offset_vdim:] = offsets
            offsets = extended

        # prepare cropping
        # offset: position of feature in target relative to reference
```

```
        # canvas offset: offset of canvas which is needed to overlap features
        #                so `canvas_offset = -offset` would be a simple solution
        canvas_offsets = -offsets
        valid_begin = canvas_offsets.max(axis=0)
        valid_end = np.min(canvas_offsets + [elem.shape for elem in movie],
                           axis=0)
        cropped_shape = tuple(valid_end - valid_begin)
        crop_offsets = valid_begin - canvas_offsets

        self._movie = movie
        self._anchors = crop_offsets
        self._shape = (n_times,) + cropped_shape

    @property
    def shape(self):
        return self._shape

    @property
    def ndim(self):
        return len(self.shape)

    @property
    def dtype(self):
        return self._movie.dtype

    def __len__(self):
        return self.shape[0]

    def __iter__(self):
        return (self(time) for time in range(len(self)))

    def __call__(self, time):
        if time < 0:
            time += len(self)
        anchor = self._anchors[time]
        vindex = tuple(slice(b, b + s) for b, s
                       in zip(anchor, self.shape[1:]))  # volume
        return self._movie[time][vindex]

    def mapped(self, sl=slice(None), verbose=False):
        res_times = range(len(self))[sl]
        buf = np.empty((len(res_times),) + self.shape[1:], self.dtype)
        if verbose:
            print('Creating movie')
        with tqdm.tqdm(res_times, unit='frame', disable=not verbose) as pbar:
            for bindex, time in enumerate(pbar):
```

```
                buf[bindex] = self(time)
        return buf

    def __getitem__(self, index):
        """
        Returns transformed movie according to offsets.

        Parameters
        ----------
        index: slice
        verbose: bool, optional

        Returns
        -------
        transformed: ndarray
        """
        movie = self._movie
        anchors = self._anchors

        if isinstance(index, tuple):
            index = index + (slice(None),) * (self.ndim - len(index))
        else:
            index = (index,) + (slice(None),) * (self.ndim - 1)

        total = tuple(range(s) for s in self.shape)
        res_index = tuple(elem[i] for elem, i in zip(total, index)) # resolve
        buf_shape = tuple(len(elem) for elem in res_index
                          if isinstance(elem, range))

        # crop
        tindex = res_index[0]
        if isinstance(tindex, int):
            return self(tindex)[index[1:]]
        else:   # slice
            buf = np.empty(buf_shape, self.dtype)
            for bindex, time in enumerate(tindex):
                buf[bindex] = self(time)[index[1:]]
            return buf

def create_transform(kpseries, target_kps, kernel=1):
    offsets = [np.array([np.array(rkp) - tkp for rkp, tkp
                         in zip(kps, target_kps) if rkp is not None])
               for kps in kpseries]
    min_offset = np.min([elem.min(axis=0) for elem in offsets], axis=0)
    max_offset = np.max([elem.max(axis=0) for elem in offsets], axis=0)
```

```
        edges = tuple(np.linspace(begin - 0.5, end + 0.5, end - begin + 2)
                      for begin, end in zip(min_offset, max_offset))
        hist = np.empty((len(offsets),) + tuple(len(elem) - 1 for elem in edges))
        for frame, frame_offsets in enumerate(offsets):
            hist[frame] = np.histogramdd(frame_offsets, edges)[0]
        hist = skimage.filters.gaussian(hist, kernel)

        max_at = [np.unravel_index(frame_hist.argmax(), frame_hist.shape)
                  for frame_hist in hist]
        centers = [range(begin, end) for begin, end
                   in zip(min_offset, max_offset + 1)]
        return [tuple(c[i] for c, i in zip(centers, index)) for index in max_at]

    # for multiprocessing
    def _match_template(args):
        return skimage.feature.match_template(*args)

    class PolyTransform:
        def __init__(self, polys):
            self._polys = polys

        def __call__(self, args):
            return tuple(poly(arg) for poly, arg in zip(self._polys, args))

        def __str__(self):
            format_str = "{} -> {}'"
            return '\n'.join([format_str.format(str(poly), poly.variable)
                              for poly in self._polys])

        def __repr__(self):
            return str(tuple(self._polys))

        def inverse(self):
            inverse_coeffs = []
            for poly in self._polys:
                c0, c1 = poly.coeffs
                inverse_coeff = [1 / c0, -c1 / c0]
                inverse_coeffs.append(inverse_coeff)
            return PolyTransform(tuple(np.poly1d(c) for c in inverse_coeffs))

    class Embedder:
        def __init__(self, ambient_image):
            self._ambient_image = ambient_image

        def embed(self, subimage, interval, n_proc=None, verbose=False):
            if n_proc is None:
```

```
    n_proc = mp.cpu_count() - 1

ambimage = self._ambient_image
## xy-plane registration
# poorman's registration
substack = subimage.max(axis=0)
detector = FeatureDetector((256, 256), (16, 16), 32, 256)   # fix
kps, features = detector.detect(substack, n_proc=n_proc,
                                verbose=verbose)
tracker = FeatureTracker(kps, features)
ambstacks = np.array([ambimage[b::interval].max(axis=0)
                      for b in range(interval)])
kpseries = tracker.track(ambstacks, 8, n_proc=n_proc, verbose=verbose)
amboffset = np.median(create_transform(kpseries, kps), axis=0)
if verbose:
    print('XY shift detected: {}'.format(amboffset))

# map
offsets = [(0, 0)] * len(subimage) + [amboffset] * len(ambimage)
concatenated = concatenate(subimage, ambimage)
planar_mapper = MovieMapper(concatenated, offsets)
mapped = planar_mapper.mapped(verbose=verbose)
subimage, ambimage = mapped[:len(subimage)], mapped[len(subimage):]

## z axis embedding
# evaluate distances
distances = np.empty((len(subimage), len(ambimage)))
if verbose:
    print('Measuring distances')
with tqdm.tqdm(itertools.product(*[range(s) for s in distances.shape]),
               total=distances.size, unit='pair',
               disable=not verbose) as pbar:
    for num, (subindex, ambindex) in enumerate(pbar):
        u, v = subimage[subindex].flat, ambimage[ambindex].flat
        distances.flat[num] = scipy.spatial.distance.euclidean(u, v)

# normalize distances
amb_moment = np.sqrt([np.sum(elem * elem) for elem in ambimage])
distances /= amb_moment
sub_moment = np.sqrt([np.sum(elem * elem) for elem in subimage])
distances /= sub_moment[:, np.newaxis]

# cost function
def cost(a0, a1=interval, distances=distances):   # a1 * x + a0
    fit = np.poly1d([a1, a0])
    n_sub, n_amb = distances.shape
```

```
                subindices = range(n_sub)
                ambindices = [int(round(elem)) for elem in fit(subindices)]
                try:
                    caught = [distances[index] for index in zip(subindices, ambindices)]
                    return np.mean(caught)
                except IndexError:
                    return np.nan

            # brute force search
            grid = (slice(0, int(len(ambimage) - interval * len(subimage)), 0.1),)
            result = scipy.optimize.brute(cost, grid,
                                          finish=scipy.optimize.fmin)
            return PolyTransform([np.poly1d([interval, result], variable='z'),
                                  np.poly1d([1, amboffset[0]], variable='y'),
                                  np.poly1d([1, amboffset[1]], variable='x')])

    class FeatureTracker:
        def __init__(self, keypoints, features):
            """
            Parameters
            ----------
            keypoints: tuple of length N
            features: tuple of length N
            """
            self._keypoints = keypoints
            self._features = features

        def track(self, movie, tolerance, n_proc=None, verbose=False):
            """
            Paramters
            ---------
            movie: ndarray
            tolerance: int
            n_proc: int, optional
            verbose: bool, optional
            """
            if n_proc is None:  # default parameter for n_proc
                n_proc = mp.cpu_count() - 1

            features = self._features
            target_keypoints = self._keypoints
            n_sdim = movie.ndim - 1 # spatial dimension
            n_times = len(movie)
            search_indices = [tuple(slice(0, None) for d in range(n_sdim))
                              for num in range(len(features))]
```

```
def guess_indices(offset, image_shape=movie.shape[1:],
                  feature_shapes=[f.shape for f in features],
                  target_keypoints=target_keypoints, tolerance=tolerance):
    validity, indices = [], []
    for keypoint, shape in zip(target_keypoints, feature_shapes):
        point = tuple(p + o for p, o in zip(keypoint, offset))
        # cache for next search
        begin = [p - tolerance for p in point]
        end = [p + tolerance + 1 + s for s, p in zip(shape, point)]
        # check index range
        if (all([b >= 0 for b in begin]) and
            all([e <= s for e, s in zip(end, image_shape)])):
            validity.append(True)
        else:
            validity.append(False)  # outside of image
        index = tuple(slice(b, e) for b, e in zip(begin, end))
        indices.append(index)
    return validity, indices

def fill_mask(mask, fill, constant):
    dispenser = iter(fill)
    return tuple(next(dispenser) if flag else constant
                 for flag in mask)

with mp.Pool(processes=n_proc) as pool:
    # initial search
    args = ((movie[0], feature) for feature in features)
    if verbose:
        print('Initial search')
    with tqdm.tqdm(args, total=len(features), unit='feature',
                   disable=not verbose) as pbar:
        keypoints = [elem for elem in pool.imap(_locate_feature, pbar)]
    offset = match_points(keypoints, target_keypoints)
    validity, next_indices = guess_indices(offset)

    # tracking
    offsets = np.empty((n_times, n_sdim), int)
    tracked = []
    if verbose:
        print('Tracking')
    with tqdm.trange(n_times, unit='frame', disable=not verbose) as pbar:
        for time in pbar:
            args = ((movie[time][index], feature, index, False)
                    for index, feature, valid in zip(next_indices,
                                                     features, validity)
                    if valid)
```

```
                    keypoints = pool.map(_locate_feature, args)
                    valid_points = [e for e, valid
                                    in zip(target_keypoints, validity) if valid]
                    offset = match_points(keypoints, valid_points)
                    offsets[time] = offset
                    tracked.append(fill_mask(validity, keypoints, None))
                    validity, next_indices = guess_indices(offset)
            return tracked

def lattice_features(image, score, shape, interval, n_keypoints, min_distance,
                     verbose=False):
    # gets lattice (keypoints which reside at top left of features)
    keypoints = lattice_points(image.shape, interval, shape)

    # shift mask (so the centers could be evaluated)
    offset = tuple(s // 2 for s in shape)
    shifted_score = np.zeros_like(score)
    from_index = tuple(slice(o, None) for o in offset)
    to_index = tuple(slice(None, -o) for o in offset)
    shifted_score[to_index] = score[from_index]
    keyids = sorted(range(len(keypoints)),
                    key=lambda i: shifted_score[keypoints[i]], reverse=True)
    keypoints = [keypoints[i] for i in keyids]
    #keypoints = [point for point in keypoints if shifted_mask[point]]

    # check min distance
    with tqdm.trange(len(keypoints), unit='point',
                     disable=not verbose) as pbar:
        selected_ids = []
        for cid in pbar:
            cpt = keypoints[cid]    # candidate point
            for sid in selected_ids:
                spt = keypoints[sid]
                distance = scipy.spatial.distance.euclidean(cpt, spt)
                if distance < min_distance:
                    break
            else:
                selected_ids.append(cid)
            if len(selected_ids) == n_keypoints:
                break
    keypoints = [keypoints[i] for i in selected_ids]

    # create features
    feature_indices = [tuple(slice(origin, origin + width)
                             for origin, width in zip(point, shape))
                       for point in keypoints]
```

```
    features = tuple(image[index] for index in feature_indices)
    return keypoints, features

def _keypoint_index(keypoint, shape):
    # determine begin and end
    begin_offset = tuple(s // 2 for s in shape)
    end_offset = tuple(s - o for s, o in zip(shape, begin_offset))
    begin = tuple(p - o for p, o in zip(keypoint, begin_offset))
    end = tuple(p + o for p, o in zip(keypoint, end_offset))

    return tuple(slice(b, e) for b, e in zip(begin, end))

def features(keypoints, shape, image):
    valid_keypoints, features = [], []
    for keypoint in keypoints:
        index = _keypoint_index(keypoint, shape)
        if all([(i.start >= 0) and (i.stop <= s)
                for i, s in zip(index, image.shape)]):
            valid_keypoints.append(tuple(i.start for i in index))
            features.append(image[index])
    return valid_keypoints, features

def lattice_points(volume_shape, interval, margin):
    """
    Returns lattice points.

    Parameters
    ----------
    volume_shape: tuple of length D
        Shape of D-dimensional lattice.
    interval: tuple of length D
    margin: tuple of length D

    Returns
    -------
    points: list
        lattice points in D-dimensional space.
    """
    ranges = [range(m, s - m + 1, i)
              for i, m, s in zip(interval, margin, volume_shape)]
    points = list(itertools.product(*ranges))
    return points

def _feature_quality(args):
    img, feature, answer = args
    imgblur = skimage.filters.gaussian(img.astype(float), 1)
```

```
        featureblur = skimage.filters.gaussian(feature.astype(float), 1)
        imgnoise = np.std(img - imgblur)
        imgspice = imgnoise * (np.random.rand(*img.shape) * 2 - 1)
        score = skimage.feature.match_template(imgblur + imgspice, featureblur)
        #return scipy.stats.kurtosis(score, axis=None)
        if score.max() > score[answer]:
            return -np.inf
        else:
            return (score[answer] - np.median(score)) / score.std()

    class FeatureDetector:
        def __init__(self, shape, stride, min_distance=None, n_keypoints=None):
            """
            Finds keypoint from the image, whose pattern is a lattice determined
            by `stride`.

            Parameters
            ----------
            shape: tuple of length D
                A shape of the features.
            stride: tuple of length D
                A stride of the lattice.
            min_distance: float, optional
                The minimum distances between the keypoints.
            n_keypoints: int, optional
                If not ``None``, the best `n_keypoints` keypoints would be selected.
            """
            self._shape = shape
            self._stride = stride
            self._min_distance = min_distance
            self._n_keypoints = n_keypoints

        def detect(self, image, n_proc=None, verbose=False):
            """
            Detects keypoints and features from the image.

            Parameters
            ----------
            image: ndarray
                An image from which the detector finds features.
            mask: ndarray, optional
            n_proc: int, optional
                The number of processes to use. If set to default (``None``),
                all cpu would be used.
            verbose: bool, optional
                If ``True``, the progress would be printed.
```

```
    Returns
    -------
    keypoints: tuple
    features: tuple
    """
    if n_proc is None:  # default parameter for n_proc
        n_proc = mp.cpu_count() - 1

    shape = self._shape
    stride = self._stride
    min_distance = self._min_distance
    n_keypoints = self._n_keypoints

    # sampling (lattice pattern)
    keypoints = lattice_points(image.shape, stride, shape)
    feature_indices = [tuple(slice(origin, origin + width)
                             for origin, width in zip(point, shape))
                       for point in keypoints]
    features = tuple(image[index] for index in feature_indices)
    #window_indices = [tuple(slice(f.start - width, f.stop + width)
    #                        for f, width in zip(f_index, stride))
    #                  for f_index in feature_indices]
    window_indices = [tuple(slice(i.start - width, i.stop + width)
                            for i, width in zip(index, shape))
                      for index in feature_indices]

    # evaluates the performances of the features
    if n_keypoints is not None or min_distance is not None:
        with mp.Pool(processes=n_proc) as pool:
            args = ((image[index], feature, shape)
                    for index, feature in zip(window_indices, features))
            #args = ((image, feature) for feature in features)
            scores = np.empty(len(features))
            if verbose:
                print('Inspecting features')
            with tqdm.tqdm(args, total=len(features), unit='feature',
                           disable=not verbose) as pbar:
                for f_num, quality in enumerate(pool.imap(_feature_quality,
                                                          pbar)):
                    scores[f_num] = quality

        # sort
        ranked_ids = sorted(range(len(scores)), key=lambda i: scores[i],
                            reverse=True)
        keypoints = [keypoints[i] for i in ranked_ids]
```

```
            features = [features[i] for i in ranked_ids]

        # check min distance
        if min_distance is not None:
            if verbose:
                print('Checking distances between the key points')
            with tqdm.trange(len(keypoints), unit='point',
                             disable=not verbose) as pbar:
                selected_ids = []
                for cid in pbar:
                    cpt = keypoints[cid]    # candidate point
                    for sid in selected_ids:
                        spt = keypoints[sid]
                        distance = scipy.spatial.distance.euclidean(cpt, spt)
                        if distance < min_distance:
                            break
                    else:
                        selected_ids.append(cid)
                    if (n_keypoints is not None and
                        len(selected_ids) == n_keypoints):
                        break
            keypoints = [keypoints[i] for i in selected_ids]
            features = [features[i] for i in selected_ids]

        # select keypoints and features
        if n_keypoints is not None:
            if verbose and len(keypoints) < n_keypoints:
                print('{} (< {}) key points found'.format(len(keypoints),
                                                          n_keypoints))
            keypoints = tuple(keypoints[:n_keypoints])
            features = tuple(features[:n_keypoints])
        return keypoints, features

def match_points(reference_points, target_points):
    """
    Returns most frequent offset from target to reference.

    Parameters
    ----------
    reference_points: list of length N (equivalent of ndarray of shape (N, D))
        N points in D-dimensional space. All elements should be tuple of size D.
    target_points: list of length N (equivalent of ndarray of shape (N, D))
        N points in D-dimensional space. All elements should be tuple of size D.

    Returns
    -------
```

```
    offset: tuple of length D
    """
    offsets = np.asarray(reference_points) - np.asarray(target_points)
    #mean_offset, std_offset = offsets.mean(axis=0), offsets.std(axis=0)
    hist_range = tuple(zip(offsets.min(axis=0), offsets.max(axis=0)))
    hist_edges = [np.linspace(begin - 0.5, end + 0.5, end - begin + 2)
                  for begin, end in hist_range]
    hist = skimage.filters.gaussian(np.histogramdd(offsets, hist_edges)[0], 1)
    max_at = np.unravel_index(hist.argmax(), hist.shape)
    left_edge = [edges[index] for edges, index in zip(hist_edges, max_at)]
    return tuple(np.ceil(left_edge).astype(int))

def _locate_feature(args):
    return locate_feature(*args)

def locate_feature(image, feature, index=None, apply_mask=True):
    """
    Returns position of feature in image.

    Parameters
    ----------
    image: ndarray of shape (A, B[, C])
    feature: ndarray of shape (a, b[, c])
    index: tuple, optional
        A mask in which it searchs for feature.
    apply_mask: bool, optional
        If True (by default), `index` would be applyed to image before search.
        If False, the offset extracted from the index would be applied to the
        output, however, `index` would NOT affect the search.
        Implemented for the performance reason.

    Returns
    -------
    point: tuple of length 2 (or 3)
    """
    search = image[index] if index is not None and apply_mask else image
    score = skimage.feature.match_template(search, feature)
    max_at = np.unravel_index(score.argmax(), score.shape)

    if index is not None:   # offset
        extended_index = index + (slice(0, None),) * (image.ndim - len(index))
        offset = np.array([i.start for i in extended_index])
        max_at = tuple(max_at + offset)
    return max_at
```

B.2 models

```
##############################################################################
##
## Canal: Calcium imaging ANALyzer
##
## Copyright (C) 2015-2016 Youngtaek Yoon <caviargithub@gmail.com>
##
## This file is part of the source code of Canal.
##
## This program is free software: you can redistribute it and/or modify
## it under the terms of the GNU General Public License as published by
## the Free Software Foundation, either version 3 of the License, or
## (at your option) any later version.
##
## This program is distributed in the hope that it will be useful,
## but WITHOUT ANY WARRANTY; without even the implied warranty of
## MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
## GNU General Public License for more details.
##
## You should have received a copy of the GNU General Public License
## along with this program.  If not, see <http://www.gnu.org/licenses/>.
##
##############################################################################

import numpy as np
import scipy.ndimage.filters
import multiprocessing as mp
import itertools
import mpmath as ap
import wrappers.mpmath as hp
import random
import tqdm

class Quantizer:
    def __init__(self, n_levels):
        self._n_levels = n_levels

    def quantize(self, raw):
        return np.array([self._quantize(elem) for elem in raw])

    def _quantize(self, raw):
        std = raw.std()
        max_std = self._n_levels - 1
        digital = np.ones(len(raw), int) * max_std
```

```
        for thres in range(max_std)[::-1]:
            digital[raw < std * (thres + 1)] = thres
        return digital

class SpatialCodec:
    def __init__(self, observation):
        self._n_pos = len(observation)
        self._radix = observation.max() + 1

    def encode(self, raw):
        moment = self._radix ** np.arange(self._n_pos)
        return np.dot(moment, raw)

class TemporalCodec:
    def __init__(self, observation):
        id_size = observation.max() + 1
        valid_ids = np.array([np.any(observation == oid)
                              for oid in range(id_size)])

        decoder = np.arange(id_size)[valid_ids]
        encoder = np.ones(id_size, int) * -1
        for encoded_id, raw_id in enumerate(decoder):
            encoder[raw_id] = encoded_id

        self._encoder = encoder
        self._decoder = decoder

    def encode(self, raw_id):
        if raw_id < len(self._encoder):
            cand_id = self._encoder[raw_id]
            if cand_id != -1:
                return cand_id
        raise ValueError('Invalid raw id')

    def decode(self, encoded_id):
        if encoded_id < len(self._decoder):
            return self._decoder[encoded_id]
        raise ValueError('Invalid encoded id')

    def encode_observation(self, raw):
        return np.array([self.encode(elem) for elem in raw])

    def encode_emission(self, raw):
        removed = raw.T[self._decoder].T
        return removed / removed.sum(axis = -1, keepdims = True)
```

```
class Simulator:
    def __init__(self, transition, emission):
        self._transition = transition
        self._emission = emission

    def simulate_states(self, n_times, init_dist = None):
        states = np.empty(n_times, int)
        transition = self._transition

        # initial state
        random.seed()
        if init_dist is None:
            rec = transition.diagonal()
            norm_rec = rec / rec.sum()
            states[0] = Simulator._draw(norm_rec)
        elif isinstance(init_dist, int):
            states[0] = init_dist
        elif iterable(init_dist):
            arrdist = np.array(init_dist)
            norm_dist = arrdist / arrdist.sum()
            states[0] = Simulator._draw(norm_dist)
        else:
            raise ValueError('initial distribution must be None or integer or '
                             '1-d iterable')

        # get sequence of states
        for time in range(1, n_times):
            states[time] = Simulator._draw(transition[states[time - 1]])

        return states

    def simulate_observations(self, states):
        emission = self._emission
        random.seed()
        return np.array([Simulator._draw(emission[state]) for state in states])

    def _draw(probabilities):
        rand = random.random()
        for index, val in enumerate(np.cumsum(probabilities)):
            if rand < val:
                return index

def state_masks(state_begin, state_end, path):
    masks = []
    begun = False
    for time, state in enumerate(path):
```

```
        if state == state_begin:
            begun = True
            time_begin = time
        elif begun and state == state_end:
            begun = False
            masks.append(slice(time_begin, time))
    return masks

def _shuffledot(args):
    raw, shuf = args
    np.random.shuffle(shuf)
    return np.dot(raw, shuf)

def _shuffledotmulti(args):
    raw, shuf, n_shuf = args
    ret = np.empty((n_shuf, len(raw), len(shuf.T)))
    np.random.seed()
    for it in range(n_shuf):
        np.random.shuffle(shuf)
        ret[it] = np.dot(raw, shuf)
    return ret

def binary_profile(signals, indices):
    # range of mask: (-1, 1)
    n_times = signals.shape[-1]
    mask = np.zeros((len(indices), n_times), bool)
    for num, index in enumerate(indices):
        mask[num][index] = 1
    return np.dot(signals, mask.T).astype(bool)

def binary_profile_test(signals, positive, negative,
                        n_iter=512, n_proc=None, verbose=False):
    if n_proc is None:
        n_proc = mp.cpu_count() - 1
    # range of mask: (-1, 1)
    n_times = signals.shape[-1]
    mask = positive.astype(int) - negative.astype(int)
    mask = mask.reshape(1, mask.size)

    if n_proc is None:
        n_proc = mp.cpu_count() - 1

    # range of mask: (-1, 1)
    momented = np.dot(signals, mask.T)
    shuffle_momented = np.empty((n_iter,) + momented.shape)
    if verbose:
```

```
        print('Creating shuffled distribution')

    if verbose:
        with tqdm.trange(n_iter) as pbar:
            with mp.Pool(processes=n_proc) as pool:
                args = ((signals, mask.T) for i in pbar)
                for it, ret in enumerate(pool.imap(_shuffledot, args)):
                    shuffle_momented[it] = ret
    else:
        with mp.Pool(processes=n_proc) as pool:
            workiters = np.linspace(0, n_iter, n_proc + 1).astype(int)
            worknums = workiters[1:] - workiters[:-1]
            args = ((signals, mask.T, num) for num in worknums)
            shuffle_momented = np.concatenate(pool.map(_shuffledotmulti, args))

    #return (momented - momented.mean(axis=0)) / momented.std(axis=0)
    return momented

def profile(signals, path, n_iter=512, n_proc=None, verbose=False):
    """
    Parameters
    ----------
    signals: ndarray
    path: ndarray
    states: slice
    n_iter: int
    n_proc: int
    verbose: bool
    """
    if n_proc is None:
        n_proc = mp.cpu_count() - 1

    # range of mask: (-1, 1)
    n_times = signals.shape[-1]
    n_states = path.max() + 1
    mask = np.ones((n_states, n_times)) * 0 #-1
    for state in range(n_states):
        mask[state][state == path] = 1

    momented = np.dot(signals, mask.T)
    shuffle_momented = np.empty((n_iter,) + momented.shape)
    if verbose:
        print('Creating shuffled distribution')

    if verbose:
        with tqdm.trange(n_iter) as pbar:
```

```
            with mp.Pool(processes=n_proc) as pool:
                args = ((signals, mask.T) for i in pbar)
                for it, ret in enumerate(pool.imap(_shuffledot, args)):
                    shuffle_momented[it] = ret
    else:
        with mp.Pool(processes=n_proc) as pool:
            workiters = np.linspace(0, n_iter, n_proc + 1).astype(int)
            worknums = workiters[1:] - workiters[:-1]
            args = ((signals, mask.T, num) for num in worknums)
            shuffle_momented = np.concatenate(pool.map(_shuffledotmulti, args))

    return (momented - momented.mean(axis=0)) / momented.std(axis=0)

def direct_profile(signals, path, verbose=False):
    """
    Parameters
    ----------
    signals: ndarray
    path: ndarray
    states: slice
    n_iter: int
    n_proc: int
    verbose: bool
    """

    # range of mask: (-1, 1)
    n_times = signals.shape[-1]
    n_states = path.max() + 1
    mask = np.ones((n_states, n_times)) * 0
    for state in range(n_states):
        mask[state][state == path] = 1
    mask /= mask.sum(axis=-1, keepdims=True)

    momented = np.dot(signals, mask.T)
    return momented

def out_scatter(locations, masks, titles):
    import matplotlib.pyplot as plt
    import matplotlib.gridspec
    z, y, x = locations.T
    gs = matplotlib.gridspec.GridSpec(2, 1, hspace=0)
    for title, mask in zip(titles, masks):
        fig = plt.figure()
        ax = plt.subplot(gs[0])
        if mask.dtype == bool:
            ax.scatter(x, y, color='grey')
```

```
            ax.scatter(x[mask], y[mask], color='red')
        else:
            ax.scatter(x, y, c=mask)
        plt.ylabel('Y')
        plt.title(title)
        ax = plt.subplot(gs[1], sharex=ax)
        if mask.dtype == bool:
            ax.scatter(x, z, color='grey')
            ax.scatter(x[mask], z[mask], color='red')
        else:
            ax.scatter(x, z, c=mask)
        plt.ylabel('Z')
        plt.xlabel('X')
        plt.xlim(0, 1024)
        plt.show()

def out_plot(signals, masks, titles):
    import matplotlib.pyplot as plt
    import matplotlib.gridspec
    gs = matplotlib.gridspec.GridSpec(2, 1, hspace=0, height_ratios=[10, 1])
    for title, mask in zip(titles, masks):
        fig = plt.figure()
        ax = plt.subplot(gs[0])
        ax.plot(signals[mask].T)
        plt.ylabel(r'$\Delta F / F$')
        plt.title(title)
        ax = plt.subplot(gs[1], sharex=ax)
        ax.imshow(mask.reshape((1, mask.size)), interpolation='none', aspect='auto')
        plt.xlabel('Time [frame]')
        plt.xlim(0, signals.shape[-1])
        plt.show()

def state_labels(prefix, n_phases, quiescence=True):
    n_digit = len(str(n_phases - 1))
    format_str = '{}{{:0{}}}'.format(prefix, n_digit)
    labels = [format_str.format(num) for num in range(n_phases)]
    if quiescence:
        labels = [prefix + 'Q'] + labels
    return labels

def canonical_labels(n_forward, n_backward, n_turning):
    return (state_labels('S', 0) + state_labels('F', n_forward) +
            state_labels('B', n_backward) + state_labels('TL', n_turning) +
            state_labels('TR', n_turning))

def transition_matrix(n_forward, n_backward, n_turning,
```

```
                  propagate, trigger, cool, excite):
# high precision
propagate = ap.mpf(str(propagate))
trigger = ap.mpf(str(trigger))
cool = ap.mpf(str(cool))
excite = ap.mpf(str(excite))

def subtransition(n_states, trigger, cool, propagate):
    sub = hp.diag([1.0 - trigger - cool] + [1.0 - propagate] * n_states)
    q_id = 0
    sub[q_id, q_id + 1] = trigger
    for row in range(0, -n_states, -1):
        sub[row - 1, row] = propagate
    return sub

prob_arr = hp.zeros((n_forward + 1 +
                     n_backward + 1 +
                     (n_turning + 1) * 2 +
                     1,) * 2)
sq_id = 0                           # id of steady quiscence
eqf_id = sq_id + 1                  # id of excited quiescence for forward wave
eqb_id = eqf_id + n_forward + 1     # id of excited quiescence for backward wave
eqtl_id = eqb_id + n_backward + 1   # id of excited quiescence for left turning
eqtr_id = eqtl_id + n_turning + 1   # id of excited quiescence for right turning

# for steady quiescence
prob_arr[sq_id, sq_id] = 1 - 4 * excite
prob_arr[sq_id, eqf_id] = excite
prob_arr[sq_id, eqb_id] = excite
prob_arr[sq_id, eqtl_id] = excite
prob_arr[sq_id, eqtr_id] = excite

# for excited quiescence which triggers forward wave
prob_arr[eqf_id, sq_id] = cool

# for excited quiescence which triggers backward wave
prob_arr[eqb_id, sq_id] = cool

# for excited quiescence which triggers turning
prob_arr[eqtl_id, sq_id] = cool
prob_arr[eqtr_id, sq_id] = cool

# for forward wave
forward_arr = subtransition(n_forward, trigger, cool, propagate)
prob_arr[eqf_id:eqb_id, eqf_id:eqb_id] = forward_arr
```

```
    # for backward wave
    backward_arr = subtransition(n_backward, trigger, cool, propagate)
    prob_arr[eqb_id:eqtl_id, eqb_id:eqtl_id] = backward_arr

    # for left turning
    turning_arr = subtransition(n_turning, trigger, cool, propagate)
    prob_arr[eqtl_id:eqtr_id, eqtl_id:eqtr_id] = turning_arr

    # for right turning
    prob_arr[eqtr_id:, eqtr_id:] = turning_arr

    return prob_arr / prob_arr.sum(axis = -1, keepdims = True)

def emission_matrix(n_waves, n_turnings, sigma):
    n_states = (n_waves + 1) * 2 + (n_turnings + 1) * 2 + 1
    n_cells = n_waves * 2
    n_obs = 2 ** n_cells
    emit_arr = hp.zeros((n_states, n_obs))

    # quiscence
    silent = ap.mpf(n_waves)
    spon = ap.mpf(1)

    sq_id = 0
    emit_arr[sq_id] = spon
    emit_arr[sq_id, 0] = silent

    eqf_id = sq_id + 1
    emit_arr[eqf_id] = spon
    emit_arr[eqf_id, 0] = silent

    eqb_id = eqf_id + n_waves + 1
    emit_arr[eqb_id] = spon
    emit_arr[eqb_id, 0] = silent

    eqtl_id = eqb_id + n_waves + 1
    emit_arr[eqtl_id] = spon
    emit_arr[eqtl_id, 0] = silent

    eqtr_id = eqtl_id + n_turnings + 1
    emit_arr[eqtr_id] = spon
    emit_arr[eqtr_id, 0] = silent

    # seed for templates
    seed = np.eye(n_waves)
    blur = np.array([scipy.ndimage.filters.gaussian_filter1d(vec, sigma)
```

```
                    for vec in seed])

    def subemission(template, n_states):
        n_cells = len(template)
        emission = hp.zeros((n_states, 2 ** n_cells))
        for o_id, mask in enumerate(itertools.product(
                                    *itertools.repeat((False, True), n_cells))):
            for s_id in range(n_states):
                selected = template[np.array(mask[::-1]), s_id]
                emission[s_id][o_id] = selected.mean() if len(selected) != 0 else 0
        return emission

    # forward wave
    forward_template = hp.asmparray(np.r_[blur, blur][:, ::-1])
    emit_arr[eqf_id + 1:eqb_id] = subemission(forward_template, n_waves)

    # backward wave
    backward_template = forward_template[:, ::-1]
    emit_arr[eqb_id + 1:eqtl_id] = subemission(backward_template, n_waves)

    #anterior_subtemplate = np.zeros_like(blur)
    #anterior_subtemplate[:n_turnings] = blur[-n_turnings:, ::-1]

    # left turning
    #left_turning_template = hp.asmparray(np.r_[anterior_subtemplate,
    #                                           np.zeros(blur.shape)])
    #emit_arr[eqtl_id + 1:eqtr_id] = subemission(left_turning_template, n_turnings)

    # right turning
    #right_turning_template = hp.asmparray(np.r_[np.zeros(blur.shape),
    #                                            anterior_subtemplate])
    #emit_arr[eqtr_id + 1:] = subemission(right_turning_template, n_turnings)
    # left turning
    left_turning_template = hp.asmparray(np.r_[blur, np.zeros(blur.shape)])
    emit_arr[eqtl_id + 1:eqtr_id] = subemission(left_turning_template, n_turnings)

    # right turning
    right_turning_template = hp.asmparray(np.r_[np.zeros(blur.shape), blur])
    emit_arr[eqtr_id + 1:] = subemission(right_turning_template, n_turnings)

    return emit_arr / emit_arr.sum(axis = -1, keepdims = True)

def smooth(vec, noise_sigma, background_sigma):
    blur = scipy.ndimage.filters.gaussian_filter1d(vec, noise_sigma)
    background = scipy.ndimage.filters.gaussian_filter1d(vec, background_sigma)
    return blur / background - 1
```

```
def viterbi_path(observation, transition, emission, init_dist, verbose=False):
    '''
    returns viterbi path.

    Parameters
    ----------
    observation: integer array of shape (T,)
    observation[i] is the observation on time i.

    transition: float array of shape (S, S)
    transition[i, j] is the transition probability of transiting from state i to state j.

    emission: float array of shape (S, O)
    emission[i, j] is the probability of observing j from state i.

    init_dist: float array of shape (S,)
    init_dist[i] is the initial probability of being state i.

    Returns
    -------
    viterbi_path: integer array of shape (T,)
    The most likely state sequence.
    '''

    # input parameter validation
    if observation.dtype != int:
        raise ValueError('Condition observation.dtype == int not met.')

    n_states, n_obs = emission.shape
    if transition.shape != (n_states,) * 2:
        raise ValueError('')
    if init_dist.shape != (n_states,):
        raise ValueError('')

    if not observation.max() < n_obs:
        raise ValueError('')
    if not hp.array_equiv(hp.asmparray(transition, ap.iv.mpf).sum(axis=-1), 1):
        raise ValueError('')
    if not hp.array_equiv(hp.asmparray(emission, ap.iv.mpf).sum(axis=-1), 1):
        raise ValueError('')
    if not hp.array_equiv(hp.asmparray(init_dist, ap.iv.mpf).sum(), 1):
        raise ValueError('')

    # most likely path (viterbi path)
    n_times = len(observation)
```

```
    path_probs = hp.zeros((n_times, n_states))
    path_states = np.empty((n_times, n_states), int)

    if verbose:
        print('Determining state for each frame')
    path_probs[0] = emission[:, observation[0]] * init_dist
    with tqdm.trange(1, n_times, unit='frame', disable=not verbose) as pbar:
        for time in pbar:
            for state in range(n_states):
                every_probs = (path_probs[time - 1] * transition[:, state] *
                               emission[state, observation[time]])
                max_prob_state = every_probs.argmax()

                path_states[time, state] = max_prob_state
                path_probs[time, state] = every_probs[max_prob_state]

    # viterbi path
    viterbi_path = np.empty(n_times, int)
    viterbi_path[-1] = path_probs[-1].argmax()
    for time in range(1, n_times)[::-1]:
        viterbi_path[time - 1] = path_states[time, viterbi_path[time]]

    return viterbi_path

def baum_welch_iter(observation, transition, emission, init_dist,
                    n_iter, verbose=False):
    for num in range(n_iter):
        if verbose:
            print('Iter {}'.format(num))
        transition, emission, init_dist = baum_welch(observation, transition,
                                                     emission, init_dist,
                                                     verbose=verbose)
    return transition, emission, init_dist

def baum_welch_converge(observation, transition, emission, init_dist,
                        verbose=False):
    prev_viterbi = viterbi_path(observation, transition, emission, init_dist,
                                verbose)
    for num in itertools.count():
        if verbose:
            print('Iter {}'.format(num))
        transition, emission, init_dist = baum_welch(observation, transition,
                                                     emission, init_dist,
                                                     verbose=verbose)
        next_viterbi = viterbi_path(observation, transition, emission,
                                    init_dist, verbose)
```

```
        if np.all(prev_viterbi == next_viterbi):
            return transition, emission, init_dist
        else:
            prev_viterbi = next_viterbi

def baum_welch(observation, transition_seed, emission_seed, init_dist_seed,
               verbose=False):
    '''
    returns the maximum likelihood estimate of the parameters of a HMM.

    Parameters
    ----------
    observation: integer array of shape (T,)
    observation[i] is the observation on time i.

    transition_seed: float array of shape (S, S)
    transition_seed[i, j] is the transition probability of transiting from state i to sta

    emission_seed: float array of shape (S, O)
    emission_seed[i, j] is the probability of observing j from state i.

    init_dist_seed: float array of shape (S,)
    init_dist_seed[i] is the initial probability of being state i.

    Returns
    -------
    transition: float array of shape (S, S)
    transition[i, j] is the transition probability of transiting from state i to state j.

    emission: float array of shape (S, O)
    emission[i, j] is the probability of observing j from state i.

    init_dist: float array of shape (S,)
    init_dist[i] is the initial probability of being state i.
    '''

    # input parameter validation
    if observation.dtype != int:
        raise ValueError('Condition observation_seed.dtype == int not met.')

    n_states, n_obs = emission_seed.shape
    if transition_seed.shape != (n_states,) * 2:
        raise ValueError('')
    if init_dist_seed.shape != (n_states,):
        raise ValueError('')
```

```
if not observation.max() < n_obs:
    raise ValueError('')
if not hp.array_equiv(hp.asmparray(transition_seed,
                                   ap.iv.mpf).sum(axis=-1), 1):
    raise ValueError('')
if not hp.array_equiv(hp.asmparray(emission_seed,
                                   ap.iv.mpf).sum(axis=-1), 1):
    raise ValueError('')
if not hp.array_equiv(hp.asmparray(init_dist_seed, ap.iv.mpf).sum(), 1):
    raise ValueError('')

# forward procedure
n_times = len(observation)
for_probs = hp.zeros((n_times, n_states))
for_probs[0] = emission_seed[:, observation[0]] * init_dist_seed
if verbose:
    print('Proceding forward')
with tqdm.trange(1, n_times, unit='frame', disable=not verbose) as pbar:
    for time in pbar:
        for state in range(n_states):
            for_probs[time, state] = (np.dot(for_probs[time - 1],
                                             transition_seed[:, state]) *
                                      emission_seed[state, observation[time]])

# backward procedure
back_probs = hp.zeros((n_times, n_states))
back_probs[-1] = 1.0
if verbose:
    print('Proceding backward')
with tqdm.tqdm(range(n_times - 1)[::-1], unit='frame',
               disable=not verbose) as pbar:
    for time in pbar:
        for state in range(n_states):
            back_probs[time, state] = np.dot(transition_seed[state],
                                             back_probs[time + 1] *
                                             emission_seed[:, observation[time]])

# probability
mono_probs = for_probs * back_probs
mono_probs /= mono_probs.sum(axis=-1, keepdims=True)

intermediate = np.array([emission_seed[state][observation[1:]]
                         for state in range(n_states)]).T
# need pbar
binary_probs = (for_probs[:-1][..., np.newaxis] *
                transition_seed[np.newaxis] *
```

```
                back_probs[1:][:, np.newaxis] *
                intermediate[:, np.newaxis])
binary_probs /= binary_probs.sum(axis=-1).sum(axis=-1)[:, np.newaxis,
                                                       np.newaxis]

# update
init_dist = mono_probs[0]
transition = (binary_probs.sum(axis=0) /
              mono_probs[:-1].sum(axis=0)[:, np.newaxis])
transition /= transition.sum(axis=-1, keepdims=True) # normalize
indicator = np.array([observation == obs_id for obs_id in range(n_obs)])
emission = (np.dot(indicator, mono_probs) / mono_probs.sum(axis=0)).T

return transition, emission, init_dist
```

B.3 signal

```
##############################################################################
##
## Canal: Calcium imaging ANALyzer
##
## Copyright (C) 2015-2016 Youngtaek Yoon <caviargithub@gmail.com>
##
## This file is part of the source code of Canal.
##
## This program is free software: you can redistribute it and/or modify
## it under the terms of the GNU General Public License as published by
## the Free Software Foundation, either version 3 of the License, or
## (at your option) any later version.
##
## This program is distributed in the hope that it will be useful,
## but WITHOUT ANY WARRANTY; without even the implied warranty of
## MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
## GNU General Public License for more details.
##
## You should have received a copy of the GNU General Public License
## along with this program.  If not, see <http://www.gnu.org/licenses/>.
##
##############################################################################

import numpy as np
import scipy.ndimage.filters
import canal.local
import multiprocessing as mp
import tqdm
import scipy.stats
import scipy.interpolate
import canal.image.esti

def _sharpen_kernel(vec, width):
    label, label_max = scipy.ndimage.label(vec)
    ret = np.zeros_like(vec)
    for l in range(1, label_max + 1):
        mask = l == label
        start = mask.argmax()
        for index in range(start, min(start + width, vec.size)):
            if mask[index]:
                ret[index] = True
    return ret
```

```
def _sharpen_array(arr, width):
    if arr.ndim == 1:
        return _sharpen_kernel(arr, width)
    else:
        buf = np.empty(arr.shape, bool)
        for num, elem in enumerate(arr):
            buf[num] = _sharpen_array(elem, width)
        return buf

def _sharpen_packed(args):
    return _sharpen_array(*args)

def _sharpen_parallel(arr, width, n_proc, verbose):
    with mp.Pool(processes=n_proc) as pool:
        args = ((elem, width) for elem in arr)
        buf = np.empty(arr.shape, bool)
        with tqdm.tqdm(args, total=len(arr), disable=not verbose) as pbar:
            for num, ret in enumerate(pool.imap(_sharpen_packed, pbar)):
                buf[num] = ret
    return buf

def sharpen(arr, width, verbose=False):
    arr = np.asarray(arr)
    if arr.ndim == 1 or len(arr) == 1:
        return _sharpen_array(arr, width)
    else:
        n_proc = min(mp.cpu_count() - 1, len(arr))
        return _sharpen_parallel(arr, width, n_proc, verbose)

# binarize -> _binarize_array -> _binarize_kernel
def _binarize_kernel(vec, dev):
    # gaussian approximation of background
    center, std = canal.image.esti.statistic_background(vec, 512)
    centered = vec - center
    return centered > dev * std

def _binarize_array(arr, dev):
    if arr.ndim == 1:
        return _binarize_kernel(arr, dev)
    else:
        buf = np.empty(arr.shape, bool)
        for num, elem in enumerate(arr):
            buf[num] = _binarize_array(elem, dev)
        return buf

def _binarize_packed(args):
```

```
    return _binarize_array(*args)

def _binarize_parallel(arr, dev, n_proc, verbose):
    with mp.Pool(processes=n_proc) as pool:
        args = ((elem, dev) for elem in arr)
        buf = np.empty(arr.shape, bool)
        with tqdm.tqdm(args, total=len(arr), disable=not verbose) as pbar:
            for num, ret in enumerate(pool.imap(_binarize_packed, pbar)):
                buf[num] = ret
    return buf

def binarize(arr, dev, verbose=False):
    arr = np.asarray(arr)
    if arr.ndim == 1 or len(arr) == 1:
        return _binarize_array(arr, dev)
    else:
        n_proc = min(mp.cpu_count() - 1, len(arr))
        return _binarize_parallel(arr, dev, n_proc, verbose)

def _normalize_kernel(vec, width):
    # baseline estimation
    base_esti = canal.local.minimum(vec, width)
    base_blur = scipy.ndimage.filters.gaussian_filter1d(base_esti, width)
    flat = vec / base_blur - 1

    # gaussian approximation of background noise
    center, std = canal.image.esti.statistic_background(flat, 512)
    return flat - center

def _normalize_array(arr, width):
    if arr.ndim == 1:
        return _normalize_kernel(arr, width)
    else:
        buf = np.empty(arr.shape)
        for num, elem in enumerate(arr):
            buf[num] = _normalize_array(elem, width)
        return buf

def _normalize_packed(args):
    return _normalize_array(*args)

def _normalize_parallel(arr, width, n_proc, verbose):
    with mp.Pool(processes=n_proc) as pool:
        args = ((elem, width) for elem in arr)
        buf = np.empty(arr.shape)
        with tqdm.tqdm(args, total=len(arr), disable=not verbose) as pbar:
```

```
                for num, ret in enumerate(pool.imap(_normalize_packed, pbar)):
                    buf[num] = ret
    return buf

def normalize(arr, width, verbose=False):
    arr = np.asarray(arr)
    if arr.ndim == 1 or len(arr) == 1:
        return _normalize_array(arr, width)
    else:
        n_proc = min(mp.cpu_count() - 1, len(arr))
        return _normalize_parallel(arr, width, n_proc, verbose)
```