Automated Nematode Tracking System

AUTOMATED NEMATODE TRACKING SYSTEM

BY

ALEXANDER SCIGAJLO, B.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Master of Applied Science

Master of Applied Science (2015) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	Automated Nematode Tracking System
AUTHOR:	Alexander Scigajlo B.Sc., (Electrical Engineering) McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. Shahram Shirani, Dr. Bhagwati Gupta, Dr. P. Ravi Selvaganapathy
NUMBER OF PAGES:	xiv, 81

To L.M.M. with love.

Abstract

Many diseases, such as Parkinson's disease and heavy metal poisoning, are associated with impaired or aberrant locomotion. Because the underlying mechanisms are difficult to study in humans, simpler metazoans like *Caenorhabditis elegans* are commonly employed to model these diseases. *C. elegans* is especially useful in this respect because its innate electrotactic behaviour allows instantaneous manipulation of its locomotion using mild electric fields in a microfluidic environment, the results of which can be captured on video. However, extraction of locomotory data from these videos is a major bottleneck to the throughput of the microfluidic electrotaxis platform. In the present study, we describe the development of novel software to analyze electrotaxis videos in an automated fashion. The software, dubbed the Automated Nematode Tracking System (ANTS), uses efficient, parameterless computer vision techniques to simultaneously track and assess movement characteristics of ambulating animals. In combination with the previously described microfluidic electrotaxis platform, ANTS promises to accelerate research with C. elegans models of locomotory dysfunction.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Shahram Shirani, and my co-supervisors Dr. Bhagwati Gupta and Dr. P. Ravi Selvaganapathy for all of their guidance, help, and support. I also thank Justin Tong, Sangeena Salam, Dr. Pouya Rezai, as well as everyone from the McMaster worm lab for their assistance.

Last but not least, thank you to my family and friends for their patience and endless encouragement.

Notation and abbreviations

N_m — Median Filter Size

$\rho(t)$ — A grayscale pixel value at time t

N_ρ — Number of pixels in the image frame

μ_ρ — Pixel value mean

T — Time period parameter over which the background subtraction algorithm operates

X_T — Pixels of time $t-1$ to $t-T$

B — Number of Gaussian components for background subtraction algorithm

$\hat{\pi}_m$ — Weight for Gaussian component m

$\hat{\mu}_m$ — The estimate of the mean for Gaussian component m

$\hat{\sigma}_m^2$ — The estimate of the variance for Gaussian component m

c_T — Portion of foreground pixels in X_T that are part of the background

o_m — Binary "closeness" value to pixel from component m

α_T — Defined as $\frac{1}{T}$

δ_m — Distance from mean of component m ($\hat{\mu}_)$ to pixel ($\rho(()$ t))

$N(\rho(t); \mu, \sigma^2)$ — Normal distribution with mean μ and variance σ^2 for pixel $\rho(t)$

D_m — Mahalanobis Distance

M — Morphological structuring element

I_b — Binary image

x — Position

v — Velocity

a — Acceleration

θ — Angle

ω — Angular velocity

α — Angular acceleration

A — Kalman filter state transition matrix

H — Kalman measurement transition matrix

R — Measurement noise covariance matrix

Q — Process noise covariance matrix

P — Error Estimate Covariance Matrix

C — Kalman control signal matrix

$u(t)$ — Kalman control signal term

K — Kalman gain

k — Kalman "true" state

z — Measurement of the "true" state k

w — Process noise

ζ — Observation noise

S — Similarity matrix

q — Mean Squared Error

y — Points on a 1D curve

t — Time

c — Savitsky-Golay coefficients

J — Jacobian matrix

γ — Vector representing a polynomial

d — Degree of polynomial

N_{sg} — Sazitsky-Golay window size

V — Vandermonde matrix

m — Number of rows

n — Number of columns

N_p — Number of Points

l — Locally weighted regression function to minimize

$\hat{\vec{\beta}}$ — Locally weighted regression parameter estimates

N_l — Number of parameters $\hat{\vec{\beta}}$ for locally weighted regression

$\eta_k(t_i)$ — Locally weighted regression weighting function

$W(t)$ — Locally weighted regression weighting kernel function

h_i — Smallest number $|t_i - t_j|$, for $j = 1, ..., N_l$

ANTS— Automated Nematode Tracker

API — Application-Program Interface

moc — Meta-Object Compiler

Contents

List of Figures

Chapter 1

Introduction

Diseases that affect the nervous system and muscles manifest as behavioural abnormalities. Many of the most common, including Parkinsons disease (PD) and amyotrophic lateral sclerosis (ALS), specifically involve debilitating motor symptoms. Excessive exposure to toxic substances like methylmercury (MeHg) and 6-hydroxydopamine (6-OHDA) can also damage neurons and negatively affect motor performance. Characterization of the networks underlying these conditions requires in-depth dissection and perturbation of important signaling pathways, which cannot be fully achieved in humans due to ethical concerns as well as humans slow growth and complexity. To circumvent these issues, model organisms are commonly employed.

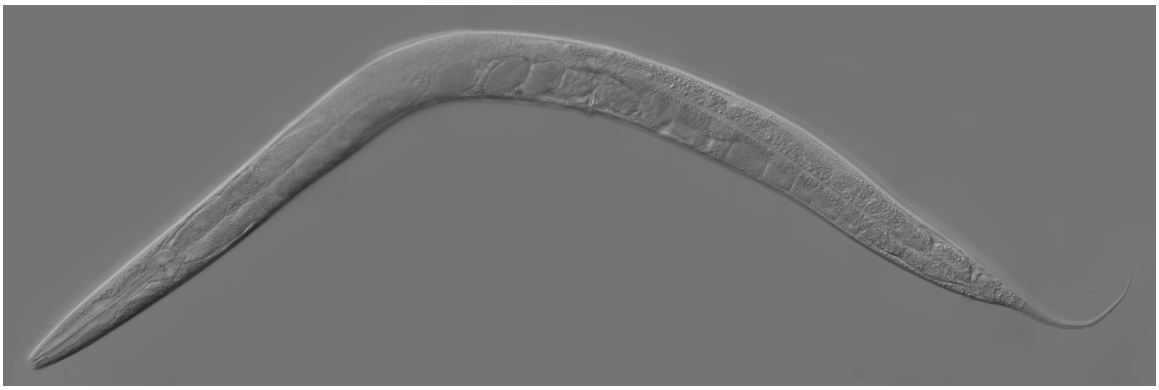


Figure 1.1: A high-resolution image of a *C. elegans* worm (Altun (2006)).

nbe 1.1, is a popular choice for this purpose because it balances ease of culture with conservation of disease mechanisms. A free-living hermaphroditic nematode, *C. elegans* offers short generation time ($\sim$ 3 days), small size ($\sim$ 1 mm in adulthood), high fertility ($\sim$ 300 progeny), simple diet (E. coli bacteria), and body transparency. At the same time, it shares roughly half of its genes with humans, allowing disease-related findings to be translated to human health (Kaletta and Hengartner (2006)). *C. elegans* is especially useful for the modelling of movement disorders due to its innate electrotactic behaviour, which allows on-demand induction and guidance of its locomotion using mild electric fields in a microfluidic environment (Rezai *et al.* (2010); Tong *et al.* (2013)) as seen in Figure 1.2. Electrotaxis was first reported in *C. elegans* by Sukul and Croll (1978), who showed that worms crawl towards the cathode in the presence of a direct current (DC) electric field. Further work demonstrated that this behaviour is mediated by a subset of amphid sensory neurons (Gabel *et al.* (2007)). The microfluidic electrotaxis assay was developed jointly by the Mechanical Engineering department and the Gupta Life Sciences lab, and has since been used to sort worms by size and phenotype (Rezai *et al.* (2012)), model PD-like symptoms following dopamine (DA) C (Salam *et al.* (2013)), and study the toxicity of metal

salts (Tong (2014)).

Figure 1.2: *C. elegans* in a micro channel. The tick marks along the top indicate 1mm increments, while the lower ones indicate 500µm increments.

While the platform's usefulness has been demonstrated, its throughput is somewhat limited by the time and labour required to manually extract locomotory data from recorded videos. Data extraction can be expedited and simplified with the use of worm tracking software. Several such software packages have been reported, including Track-A-Worm (Wang and Wang (2013)), the Multi-Worm Tracker (Swierczek *et al.* (2011)), and the Parallel Worm Tracker (Ramot *et al.* (2008)); however, these packages require additional, specialized hardware such as motorized platforms and/or

a live camera feed, with no option to use pre-recorded videos. Additionally, these programs require the user to determine sensible values for a variety of (often confusing) algorithm parameters.

We have created a new software tracker designed to quickly and robustly extract and analyze information without human interaction or calibration, dubbed the Automated Nematode Tracking System (ANTS). The goal of ANTS was to provide a simple solution for tracking nematodes during a variety of experiment types, requiring no manual input of parameters. The software, which is here demonstrated to be capable of tracking larvae, locomotory mutants, and toxicant-exposed animals in addition to untreated wild-type adults, can be installed on any PC. The software is also capable of batch-processing large numbers of videos. Given that information extraction from experimental videos is a major bottleneck for research of this nature, ANTS promises to grant a large improvement in productivity, while remaining accessible, robust, and versatile.

1.1 Strains and culturing

The following *C. elegans* strains were used in this study: N2 Bristol, CB1430 unc-40(e1430), and RM2702 dat-1(ok157). N2 and RM2702 were originally acquired from the Caenorhabditis Genetics Center (University of Minnesota, St. Paul, MN). CB1430 was obtained from Joseph Culottis lab (University of Toronto, Toronto, ON). Nematodes were grown and maintained at 20 °C on standard nematode growth medium (NGM) agar plates containing Escherichia coli OP50 culture using previously described methods (Brenner (1974)). All experiments used age-synchronous populations obtained by bleach treatment (Stiernagle (2006)).

1.2 MeHg treatments

Methylmercury chloride was obtained from Sigma-Aldrich (St. Louis, MO, USA). *C. elegans* were chronically exposed to a low concentration of MeHg using previously described methods (Tong (2014); Harada *et al.* (2007)). First, MeHg was prepared as a 20 µm stock solution in 10 % dimethylsulfoxide (DMSO). 500 µL of this solution was then spread evenly across the surface of a plate containing 10 mL of agar and allowed 24h for absorption, resulting in final concentrations of 1 µm MeHg and 0.5 % DMSO. Similar methods were used to prepare control plates containing only 0.5 % DMSO. Synchronized L1 larvae were added to plates and allowed to grow to young adulthood.

1.3 Electrotaxis assay

The fabrication of microfluidic, as can be seen in figure 1.3 devices and the electrotaxis assay proper have been previously described (Rezai *et al.* (2010); Tong *et al.* (2013)). Briefly, *C. elegans* were first washed off their culture plates, cleaned, and suspended in M9 buffer. Animals were then aspirated into the microchannel using a syringe attached to the outlet tube. Individual animals were isolated by adjusting the inlet and outlet tubes relative height to hydrostatically manipulate the flow of buffer through the channel. Pressure-induced flow was then eliminated by laying both tubes flat at the same elevation. Next, a 3 V/cm DC electric field was applied and the animals resultant behaviour recorded by camera. Locomotory data was extracted from recorded videos either manually using NIH ImageJ (Rasband (2014)) or automatically with ANTS.

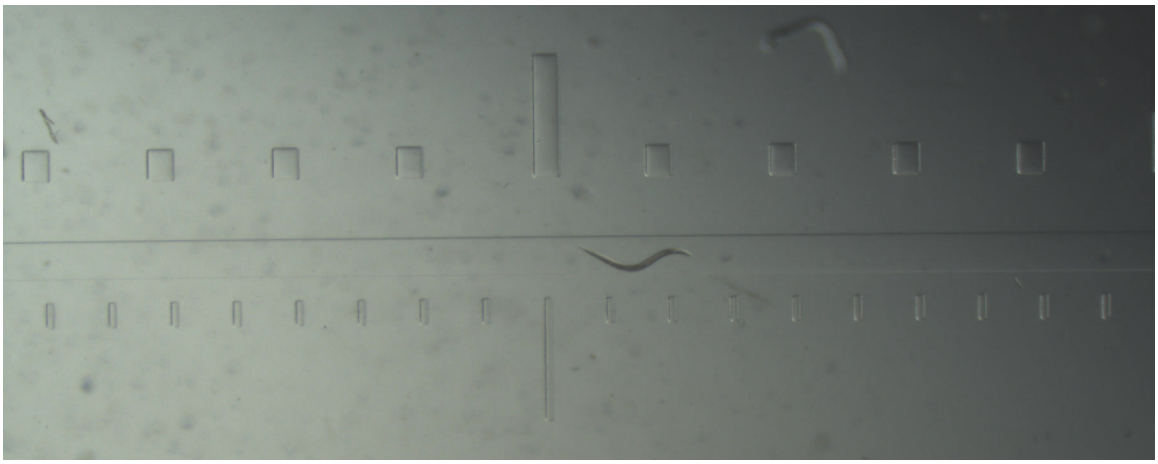


Figure 1.3: Electrotaxis Microchannel

Chapter 2

Previous Work

This chapter briefly describes Existing *C. elegans* tracking software, and similar algorithms. In later chapters we will explain our methods and designs of choice.

2.1 Existing Worm Tracking Software

This section discusses other software packages for tracking nematodes.

2.1.1 Track-A-Worm

The Multi-worm tracker defined in Wang and Wang (2013) is a multi-worm tracker. It uses a motorized stage, stereo-microscope, and a digital camera to aid in tracking the worm movements across a stage. Track-A-Worm extracts the worm from the background using a thresholding and edge detection method. The spline of the worm is then detected by using a cubic interpolation of midpoints.

2.1.2 The Parallel Worm Tracker

The Parallel Worm Tracker (Ramot *et al.* (2008)) is an offline tracking program that performs the analysis after the video capture has been completed. This tracker extracts worms by thresholding the image and performing blob analysis on remaining objects, taking into account whether the background is dark or light. The blobs are assigned to tracks using a greedy algorithm comparing centroid distance and blob area.

2.1.3 Multi-Worm Tracker

The Multi-Worm Tracker, as described in Swierczek *et al.* (2011), is a multi-worm tracker designed to work in real-time. The algorithm searches for areas either lighter or darker than the background depending on settings. Worm detections are associated with detections from previous frames by checking for the blob overlap. A newly detected blob must contain at least 50% of the pixel locations present within a previous blob for the two to be associated.

2.2 Other Related Algorithms

In this section are algorithms that were not used, but were considered for ANTS. They are tracking related algorithms and not full software packages for tracking nematodes.

2.2.1 Automated tracking of multiple *C. elegans*

This algorithm, described in Fontaine *et al.* (2006), allows for a model based automated tracking method for multiple worms. This method employs a central difference

Kalman filter to estimate worm locations and take into account occluding worms. Detected *C. elegans* are modelled as a 4th order periodic B-spline basis function and then can by accurately estimated from noisy data using the central difference Kalman filter.

2.2.2 Multiple hypothesis tracker

The Multiple hypothesis tracker, as described in Reid (1979), works by branching each tracking into as many probable tracks as computer memory allows. It calculates the probability of each possible track, depending on the model of the objects tracked, and will remove the least probable tracks. It is a known to be a very good method of tracking, if computationally expensive.

2.2.3 Particle Filter Methods

Particle filters, such as the Condensation Filter (Isard and Blake (1998)), use the Baysian filter mechanics in order to generate a probability density function model (pdf) of the image. This pdf can be used to track objects in the image. It operates by sampling points from each frame in the video, predicting samples with a expert-chosen prior function. Then, with the updated measured noise values from each new frame, samples are selected and the estimated pdf function can be generated to find new points to sample. Details differ between particle filters.

Chapter 3

Tracker Design

The software is used with pre-recorded videos, which may be captured using whatever camera equipment and software is already available. Multiple videos can be fed into the software to be analyzed with no additional human interaction. For large numbers of videos, ANTS can be left to run unattended overnight to generate its analysis. Other than the real-world width of the video frame in millimetres, the software requires no additional human input per video. Algorithms are used to automatically determine appropriate values for parameters. The algorithms were chosen with the aim of reducing computation time while still retrieving accurate information from the tracked nematodes.

3.1 General Algorithm Overview

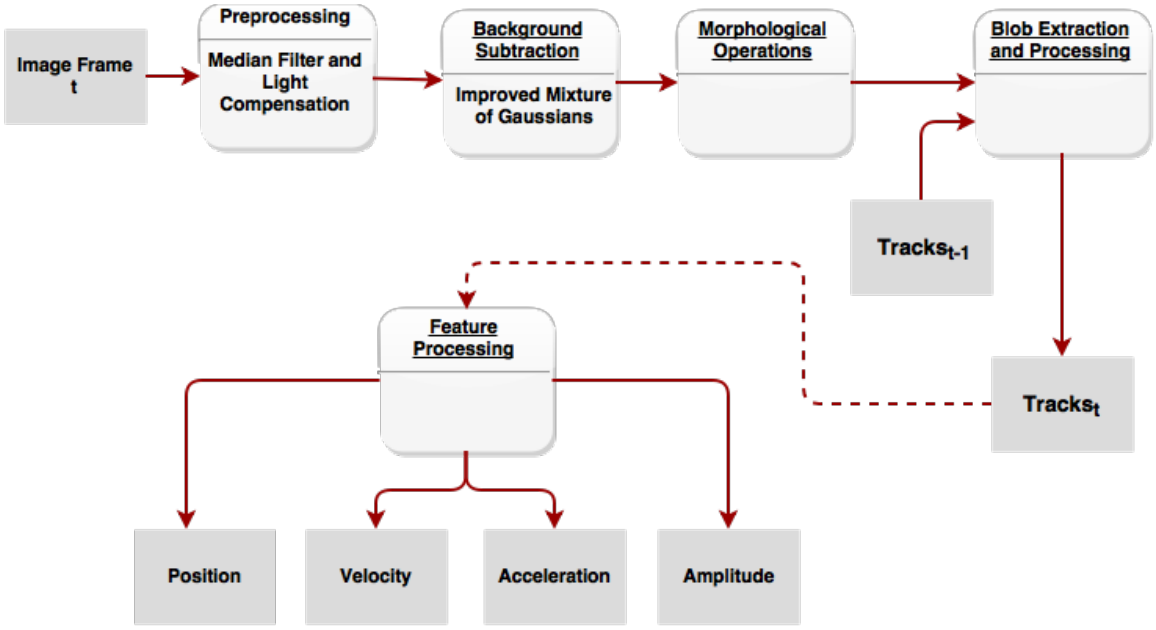


Figure 3.1: An overall view of the ANTS nematode tracking algorithm.

The general overview of the software is shown in Figure 3.1. ANTS retrieves the video data one grayscale image frame at a time. Each frame is preprocessed to remove noise and to account for changes in lighting, as lighting differences can impact the background subtraction algorithm and lead to false detections. Next, to separate objects from the background, ANTS uses a background subtraction method, specifically the Improved Adaptive Gaussian Mixture Model algorithm (Zivkovic (2004)). This takes the input grayscale image and returns a black and white binary image based on changes in motion. This will aid in the segmentation of the nematode from any background, such as an electrotaxis microchannel.

The background subtraction may still have anomalies from noise or impurities,

such as dust, within the microchannel, which are removed using morphological processing. The processed binary image now contains a number of connected pixel regions, or blobs. Given a set of n blobs in the image, a sorted list is created using a heuristic cost function to determine the maximum probability of each blob being the tracked object (the *C. elegans* specimen). Using this heuristic function, ANTS associates the blobs with blobs of previous frames. Blobs that are matched to each other temporally between frames are referred to as tracks. Finally, these tracks are analyzed to extract the position, velocity, and other features of interest.

Each of these steps will be described in further detail in the sections that follow.

3.2 Preprocessing

The preprocessing preformed on the image reduces noise and compensates for lighting changes to get better results at later stages.

3.2.1 Image Resizing

Each video frame is resized to retain a consistent pixel density between videos. ANTS uses a nearest neighbours approach when scaling up and a pixel area relation approach when scaling down. This resizing ensures that the later stages of the tracking algorithm will be using a similar number of pixels. In most cases, the video resolution will have little or no impact on tracking performance. It also has the side effect of speeding up the tracking of videos with high resolutions.

Linear Interpolation

The nearest neighbour interpolation approach selects the value of the pixel closest to the one being sampled and does not take into account any other information. It was chosen for its speed and is used when scaling images up.

Pixel Area Relation

The pixel area relation method was chosen for when the image needs to be scaled down as it is reported to be faster in this use case.

3.2.2 Median Filter

A median filter (Huang *et al.* (1979)) is applied to the image in order to attempt to improve reduce noise before other tracking is done. The median filter is defined as follows.

Given a window of size $N_m \times N_m$, the window is applied to every pixel of the image. The pixel is replaced by the median value of the window. To accomplish this the values in the window are sorted from lowest to highest and then the median is selected from the middle of the list.

The dimension of the window N_m is chosen by the user before running this algorithm.

3.2.3 Lighting Compensation

Lighting is kept consistent between frames by measuring and smoothing the differences in the global illumination. ANTS can then better differentiate the separate objects detected in an individual frame of the image from the background with fewer

false detections due to lighting. Changes to global lighting for each pixel $\rho(t)$, where t is the current frame, are compensated with the following iterative equations to obtain the modified pixel value $\rho(t)'$:

$$\rho'(t) = \left[0.005 + 0.995\left(\frac{\mu_\rho{}'(t-1)}{\mu_\rho^{(t)}}\right)\right]\rho(t) \tag{3.1}$$

Where $\mu_\rho(t)$ is the mean of all pixel values $\rho(t)$ of the frame at time t and $\mu_\rho{}'(t-1)$ is the mean of all pixel values of the previous frame after light compensation. This guarantees that changes to the global illumination between frames will be dampened.

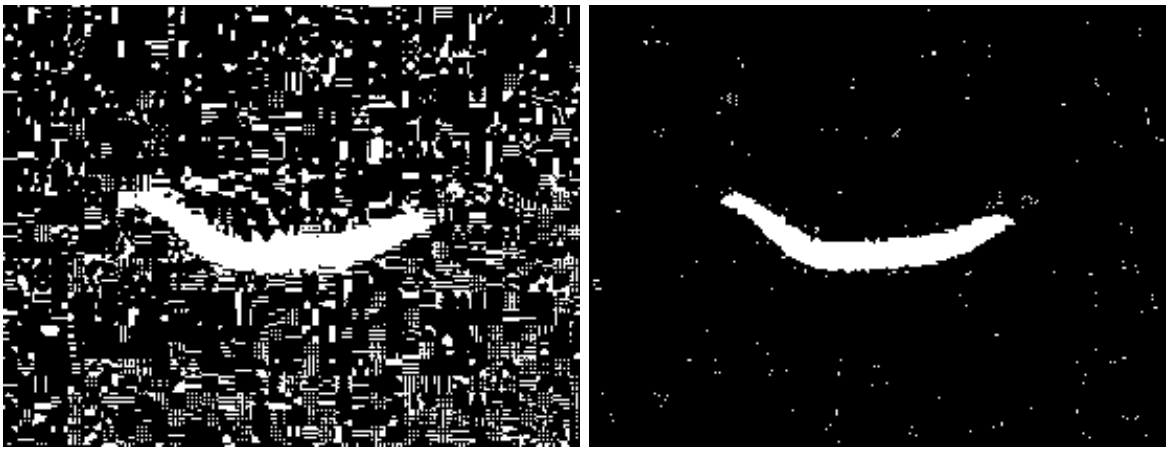
3.3 Background Subtraction

The moving nematodes can be differentiated from the video background through recognizing which pixels in each frame are part of the background or foreground.

The simplest method for doing so is using static background subtraction, where the first frame of the video sequence – where no foreground subject is present – is subtracted from later frames. The assumption is that any pixel difference between the two is due to the foreground of the image. However, this method is especially susceptible to noise; the amount of noise tends to increase with time as the reference background becomes more different over time due to outside factors such as light and moving dust.

Because of this, background subtraction in ANTS is instead performed using the Improved Adaptive Gaussian Mixture Model algorithm Zivkovic (2004), which has been modified to use only grayscale pixel values. A comparison of these two methods

can be seen in Figure 3.2. It can easily be seen that the Improved Mixture Gaussian Model Background Subtraction, shown in Figure 3.2b, is much less susceptible to noise.



(a) (b)

Figure 3.2: The output of the Static background subtraction method(a), compared with the Improved Gaussian Mixture Model method of background subtraction (b). White pixels belong to the foreground and black pixels belong to background

Given previous pixels $X_T = \{\rho(t-1), \rho(t-2), ..., \rho(t-T)\}$, sampled from previous frames $t-1$ to $t-T$, where T is a period determined for the application, the equation to approximate the background model $p(\rho(t)|X_T, BG)$ is as follows:

$$p(\rho(t)|X_T, BG) \sim \sum_{m=1}^{B} \hat{\pi}_m N(\rho(t); \hat{\mu}_m, \hat{\sigma}_m^2) \tag{3.2}$$

where Gaussian Kernel functions $N(\rho(t); \hat{\mu}_m, \hat{\sigma}_m^2)$ are used, with mean and variance estimates $\hat{\mu}_m$ and $\hat{\sigma}_m^2$, respectively. The B largest components are used, with weights $\hat{\pi}_m$, where:

$$B = \arg\min_{b}(\sum_{m=1}^{b} \hat{\pi}_m > (1 - c_T)). \tag{3.3}$$

and c_T is the portion of the data that can belong to the foreground objects without affecting the background model. This is determined by inputs into the algorithm before it is run. The background model changes adaptively with each successive frame; its values are updated with iterative equations 3.4 to 3.6:

$$\hat{\pi}_m \leftarrow \hat{\pi}_m + \alpha_T(o_m(t) - \hat{\pi}_m) - \alpha_T c_T \tag{3.4}$$

$$\hat{\mu}_m \leftarrow \hat{\mu}_m + o_m(t)\left(\frac{\alpha_T}{\hat{\pi}_m}\right)\delta_m \tag{3.5}$$

$$\hat{\sigma}_m^2 \leftarrow \hat{\sigma}_m^2 + o_m(t)\left(\frac{\alpha_T}{\hat{\pi}_m}\right)(\delta_m^2 - \hat{\sigma}_m^2) \tag{3.6}$$

where:

$$\alpha_T = \frac{1}{T} \tag{3.7}$$

$$\delta_m = \rho(t) - \hat{\mu}_m \tag{3.8}$$

This algorithm works by finding the clusters of pixels in previous frames which are considered "close" to the new observed pixel. $o_m(t)$ signifies this "closeness" and is set to 1 when a new pixel has a difference of less than three standard deviations from previous pixels. The distance from a group of observations with mean $\hat{\mu}_m$ to

new pixels $\rho(t)$ (using δ_m from eq. 3.8), is defined for scalar values as follows:

$$D_m^2\left(\rho(t)\right) = \frac{\delta_m^2}{\hat{\sigma}_m^2} \tag{3.9}$$

If there are no close components, a new component will be created with $\hat{\pi}_{M+1} = \alpha_T$, $\hat{\mu}_{M+1} = \rho(t)$, and $\hat{\sigma}_{M+1}^2 = \hat{\sigma}_0^2$, where $\hat{\sigma}_0^2$ is an approximate initial variance. The c_T term in (3.4) is a predetermined fixed percent of X_T samples required to support a component. This provides a very convenient method of differentiating the movements of the *C. elegans* from the background micro-channel, dust particles, and changing lighting conditions that may be present in the video.

3.4 Morphological Operations

The binary image extracted by the background subtraction is morphologically closed to remove small objects while closing any small holes in the image. Morphological closing is defined as a dilation of the binary image followed by the erosion (Serra (1983)). Letting I_b be the binary image and M the structuring element closing is defined by the following set operations:

$$dilation : I_b \oplus M = \left\{ \rho \;\middle|\; (\hat{M})_\rho \cap I_b \neq \varnothing \right\} \tag{3.10}$$

$$erosion : I_b \ominus M = \{ \rho \mid M_\rho \subseteq I_b \} \tag{3.11}$$

$$closing : I_b \bullet M = (I_b \oplus M) \ominus I_b \tag{3.12}$$

Where $(\hat{M})_\rho$ is a reflection of the structuring element around it's origin and then translated to the location of ρ and M_ρ is only the translation of the structuring

element. An appropriately sized structuring element has been chosen for the software beforehand.

3.5 Blob Extraction

The results from the background subtraction, as shown in Figure 3.2b, will contain a number of connected foreground pixel regions, or blobs. These blobs need to be extracted from the image, since some of them represent the moving worms in the image.

3.5.1 Blob Labeling Algorithm

ANTS uses a labeling algorithm based on the paper by Chang *et al.* (2004). This is a linear-time algorithm that obtains the external and internal contours of each blob in the binary image.

The algorithm scans the binary image a pixel at a time from left to right for each line. When scanning the image, the following rules are applied:

- When an external point of a blob is detected for the first time, a complete external trace of the outside is made, all points in the contour are assigned a label, and the surrounding white pixels are marked with a negative number to prevent future traces.

- When a previously labeled internal or external contour of a blob is encountered the black pixels that follow are assigned the same label until a white pixel is detected.

- If a nonnegative white pixel is encountered while performing the last step then an internal contour is encountered for the first time. The internal contour is traced and the same label is applied to it as the external contour and the surrounding white pixels are marked with a negative number.

A contour trace finds the internal and external contours given an unlabeled point found on the outside of a blob. If the point is determined to be an isolated point, the point is surrounded by white pixels and then the tracing can end. Otherwise, the tracer will output each new point on the contour until the first and second points that the tracer started with are returned one after the other.

The tracer is 8-connected, meaning that after with each new point it will search in a clockwise manner within the neighboring eight points for the next contour point. This eight connected search space is numbered from 0 to 7 clockwise starting at the pixel on the right of the current contour point. The initial search position, assuming this position is not the contour starting point, is set to $d + 2(mod 8)$ where d is the position of the previously detected contour point.

The tracer finds the first black pixel while labeling all white pixels that it encounters with a negative integer. This black pixel will be the next contour point.

The algorithm only needs to scan the binary image a single time to retrieve all relevant information to label the image.

3.6 Track Assignment

At this point in the algorithm, except for the first frame, there will have been previous detections. A requirement of a multiple-target tracker is that each detected object of each frame needs to be associated with detections from previous frames. This way

we can get a record of how each individual worm moves within the timespan of the video. This association of detections from one frame to the next is referred to as a track.

Each track requires a Kalman filter to estimate the true location of the worm, given that there will be inherent noise in the system.

3.6.1 Kalman Filter

Each track has an associated Kalman Filter (Kalman (1960)). A Kalman Filter takes the current state (in the case of ANTS, the position, velocity, acceleration, and angle) of the tracked object and predicts what the next state will be. This prediction will automatically update the Kalman's filter's estimate of the state. This estimate is combined with the current measurement to develop the next prediction of the state.

Model

The worm can be modeled simply with the following kinetic equations:

$$x(t+1) = x(t) + v(t) + 0.5a(t) \tag{3.13}$$

$$v(t+1) = v(t) + a(t) \tag{3.14}$$

$$a(t+1) = a(t) \tag{3.15}$$

and similar angular equations:

$$\theta(t+1) = \theta(t) + \omega(t) + 0.5\alpha(t) \tag{3.16}$$

$$\omega(t+1) = \omega(t) + \alpha(t) \tag{3.17}$$

$$\alpha(t+1) = \alpha(t) \tag{3.18}$$

These equations represent the position $x(t)$, velocity $v(t)$, acceleration $a(t)$, angle θ, angular velocity $\omega(t)$, and angular acceleration $\alpha(t)$. From these equations we can construct the Kalman Filter state and measurement transition matrices A and H as follows:

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.19}$$

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0.5 \end{bmatrix} \tag{3.20}$$

The measurement noise covariance (R), process noise covariance (Q), and the

initially set error estimate covariance matrix (P) are as follows:

$$R = 10I$$
$$Q = 1e^{-4}I$$
$$P = 0.1I$$

The process and measurement noise is approximated as Gaussian noise as follows:

$$w \approx N(0, Q)$$
$$\zeta \approx N(0, R)$$

Prediction

The predicted state $\hat{k}'(t)$ is defined as:

$$\hat{k}'(t) = A\hat{k}(t-1) + Cu(t) + w(t) \tag{3.21}$$

As ANTS does not use the control matrix and signal term, C and $u(t)$ respectively, so the above equation simplifies to:

$$\hat{k}'(t) = A\hat{k}(t-1) + w(t) \tag{3.22}$$

$\hat{k}'(t)$ is the predicted position and angle of the worm given the previous state estimate $\hat{k}(t-1)$, and the process noise $w(t)$.

The error covariance needs to be predicted for the next measurement correction, to update the error covariance the following equation is used:

$$P' \leftarrow APA^T + Q \tag{3.23}$$

Estimate Update

First we need to update the Kalman Gain with the following equation:

$$K \leftarrow P'H^T(HP'H^T + R)^{-1} \tag{3.24}$$

In order to update the state estimate:

$$\hat{k}(t) = \hat{k}'(t) + K(z(t) - H\hat{k}'(t)) \tag{3.25}$$

where $z(t)$, the new measured state, is calculated with the newly measured position and angle values $k(t)$ as follows:

$$z(t) = Hk(t) + \zeta \tag{3.26}$$

Finally, the error covariance is updated:

$$P \leftarrow (I - KH)P' \tag{3.27}$$

This process allows for error in the measurement of the worms and gives a smoother position estimate than using the raw detected points.

3.6.2 Linear Assignment Problem

ANTS needs to assign worms to previous tracks while minimizing error. Given that the linear assignment problem defined as follows:

> A number of workers need to each be assigned to a number of jobs. Each worker can only work on one of these jobs and each worker has an associated cost of performing each job. This cost may be different depending on the worker and the job. The minimum cost needs to be found in the assignment of workers to jobs.

This problem is very similar to matching detected worms to tracks by needing to match blobs from the current frame with those of previous frames based on their similarity as the cost.

This allows the tracker to extract the knowledge of the change in position over time for the tracked objects. To accomplish this, a cost matrix S is used to compare new blobs with those in the previous frame. Where each element $S_{i,j}$ is the distance squared as shown:

$$S_{i,j} = \left(\vec{x}_j - \hat{\vec{x}}'_i\right) \cdot \left(\vec{x}_j - \hat{\vec{x}}'_i\right) \tag{3.28}$$

The square root term is avoided as it makes no difference to the cost matrix for the purposes of this assignment problem and will cause the algorithm to favour closer distances more strongly.

The cost matrix S compares each of the newly observed blobs in the matrix as columns, and previously detected tracks as rows. By subtracting the current measured position $\vec{x}_i$ from the predicted position $\hat{\vec{x}}'_j$, which is computed with the Kalman Filter and calculating the square, the blob cost can be calculated. This cost matrix can be

extended with more blob features such as extent and area. The above matrix is then run through an assignment algorithm; we use the Munkres algorithm, which was selected for its simplicity and strong performance.

3.6.3 Munkres Algorithm

The Munkres Algorithm (Munkres (1957)) for linear assignment problems is used with the cost matrix computed from the similarity equation, where a higher similarity measure indicates a higher cost. A simple overview of the algorithm is as follows:

Step 1 Subtract the smallest element in the input matrix S from every element in S.

Step 2 If each row and column of the matrix has only one 0 value then a solution has been found and each column and row with 0 are assigned; otherwise continue.

Step 3 Subtract the minimum element in each of the columns of S with all elements within that column and check if a solution is found; otherwise continue.

Step 4 Assign any rows with one zero, from top to bottom, and cross out other zeros within the same column. Then mark the rows in accordance with the following rules:

- Mark all rows with no assignments.
- Mark all unmarked columns that contain a zero in a new row.
- Mark all rows that contain assignments in newly marked columns.

Repeating the above rules for all non-assigned rows.

Step 5 Remove all marked columns and unmarked rows and go back to step 1 with the modified matrix S and continue until S is empty.

After this process is complete, the blobs in the current frame are assigned to blobs of previous frames. ANTS works with the assumption that each blob in the video frame is either an update to objects previously observed, or a new object being observed for the first time. If there are leftover blobs after assignment, a new object is assumed to have been detected. In the opposite case, where an old observation is not assigned to a new observation most likely due to noise, the worm detection is assumed to have stayed at the same position.

3.6.4 Track Splitting and Merging

Sometimes, it is possible for the tracker to mistakenly identify a blob containing two overlapping worms as being a single worm. Alternatively, a single worm that has ventured close to the edge of the video frame and become stationary may be detected as being two worms. For edge cases like these, tracks are permitted to split off into multiple tracks, or be re-merged into one track.

When track splitting occurs, a new track will simply be created, while the other track continues on as normal.

For merging to occur, a common time segment t_a to t_b is first found between tracks by comparing the first and last detection times.

At each frame, the tracks contain the position vector $\vec{x}$. Tracks can have missing data at a certain frame time. Tracks may have missing data for some frames if they have not been assigned a detection. To ensure that at each frame there is data to compare, any missing position points are linearly interpolated.

The two tracks are then compared using the mean squared error equation as follows:

$$q = \frac{1}{t_b - t_a} \sum_{i=t_a}^{t_b} \|\vec{x}_0^{(i)} - \vec{x}_1^{(i)}\|^2 \tag{3.29}$$

Where $x_0^{(i)}$ is a point that is part of the longer track and $x_1^{(i)}$ is the shorter track, both tracks at frame i.

ANTS will then merge the two tracks together. Any duplicate position and angle information that are for the same frame will be averaged, while if information is missing from one track, the value from the other track will be used.

Chapter 4

Results Filtering

At this point ANTS has completed tracking the worms in the video and has the raw information extracted from tracking the worms. These results now need to be processed into a form that can be usefully interpreted.

Results from the tracked worm require smoothing and other filtering in order to increase the signal-to-noise ratio and retrieve robust results. The important design decisions regarding this are discussed in this chapter.

4.1 Savitsky-Golay Filter

We now apply the Savitsky-Golay smoothing filter (Savitzky and Golay (1964)) to improve the signal-to-noise ratio and reduce the effect of outliers in the data. It is given by the following equation, where y'_k are the smoothed points:

$$y'_k = \sum_{i=-N_{sg}}^{N_{sg}} c_i y_{k+i} \tag{4.1}$$

The c_i coefficients are calculated using a least squares estimate. Starting with γ as a polynomial of degree d, and a window width N_{sg}:

$$\gamma_i = c_d t_i^d + c_{d-1} t_i^{d-1} + \cdots + c_1 t_i^1 + c_0 t_i^0, \text{ for } i = 1 \ldots 2N_{sg} + 1 \tag{4.2}$$

And a Vandermonde matrix V of size $N_{sg} \times d$, defined as:

$$V = \begin{bmatrix} 1 & t_1 & t_1^2 & \ldots & t_1^d \\ 1 & t_2 & t_2^2 & \ldots & t_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_{N_{sg}} & t_{N_{sg}}^2 & \ldots & t_{N_{sg}}^d \end{bmatrix} \tag{4.3}$$

Each row is a geometric progression of t_i, where i is the row number, and is useful for least squares fitting as follows:

$$\gamma = Vc \tag{4.4}$$

where γ is a vector representing a polynomial γ_i. This gives us the standard least squares fit for calculating the coefficients c:

$$c = (V^T V)^{-1} V^T \gamma \tag{4.5}$$

To get the derivative value, the least squares needs to be recalculated for each value of y'_k, as only the middle estimated coefficient is used for each.

To smooth the data, coefficients only need to be calculated once with values:

$$t = 0, 1, 2, \ldots, (2N_{sg} - 1), 2N_{sg} \tag{4.6}$$

and:

$$\gamma_i = \begin{cases} 1 & \text{, if } i = N_{sg}\text{, (center value)} \\ 0 & \text{, otherwise} \end{cases} \tag{4.7}$$

This calculation can be reused for all windows.

The Savitsky-Golay filter method is equivalent to fitting data to a polynomial but has a lower computational cost. For ANTS the degree d is set to 2 and N_{sg} is set to 5% of the number of points in the input.

4.2 Local Regression

The variant Local Regression filter (Cleveland and Devlin (1988)), or LOWESS, is another algorithm used for data fitting and smoothing. This method is used to further smooth the velocity and acceleration data outputted by the Savitzky-Golay filter. As with Savitzky-Golay, a polynomial function is fitted to the data; with LOWESS, however, a weighted least squares fitting method is used.

This weighted least squares equation, which is minimized to obtain the parameter estimates $\hat{\vec{\beta}}$, is as follows:

$$\sum_{k=1}^{N_p} \eta_k(t_i) \left[y_i - \sum_{j=0}^{d} t_i^j \hat{\beta}_j \right]^2 \tag{4.8}$$

The weights $\eta_k(t_i)$ that Local Regression uses are determined from the following equation:

$$\eta_k(t_i) = W\left(\frac{|t_k - t_i|}{h_i} \right) \tag{4.9}$$

Where h_i is the smallest value $|t_k - t_i|$, out of all numbers $k = 1, 2, ..., N_l$ and $W(t)$ is the tricube equation from Cleveland (1979) for robust smoothing:

$$W(t) = \begin{cases} \left(1 - |t|^3\right)^3 & \text{, if } |t| < 1 \\ 0 & \text{, if } |t| \geq 1 \end{cases} \tag{4.10}$$

4.3 Feature Extraction

ANTS quantifies the motion of the *C. elegans* specimen with velocity, acceleration, frequency, and amplitude of body oscillation, and can be further extended in the future for other features, such as more specific posture information.Additionally, it saves data in a way that simplifies future analysis, should additional features of interest be identified later in the research process. For data analysis beyond the software's current capabilities, the raw position, speed and posture data can be exported into a CSV spreadsheet to be analyzed later with external software (such as Excel or Matlab)

The nematode's posture line is calculated from a thinning of the chosen blob to reduce it to a thinned curve. Position information is gathered from the centroid of the nematode blob's bounding box (as seen in Figure 4.1) and from the center of the calculated posture line. The software in its current form extracts the velocity, acceleration, amplitude, and frequency information of the *C. elegans* specimens. The methods of extracting and quantizing features are described below.

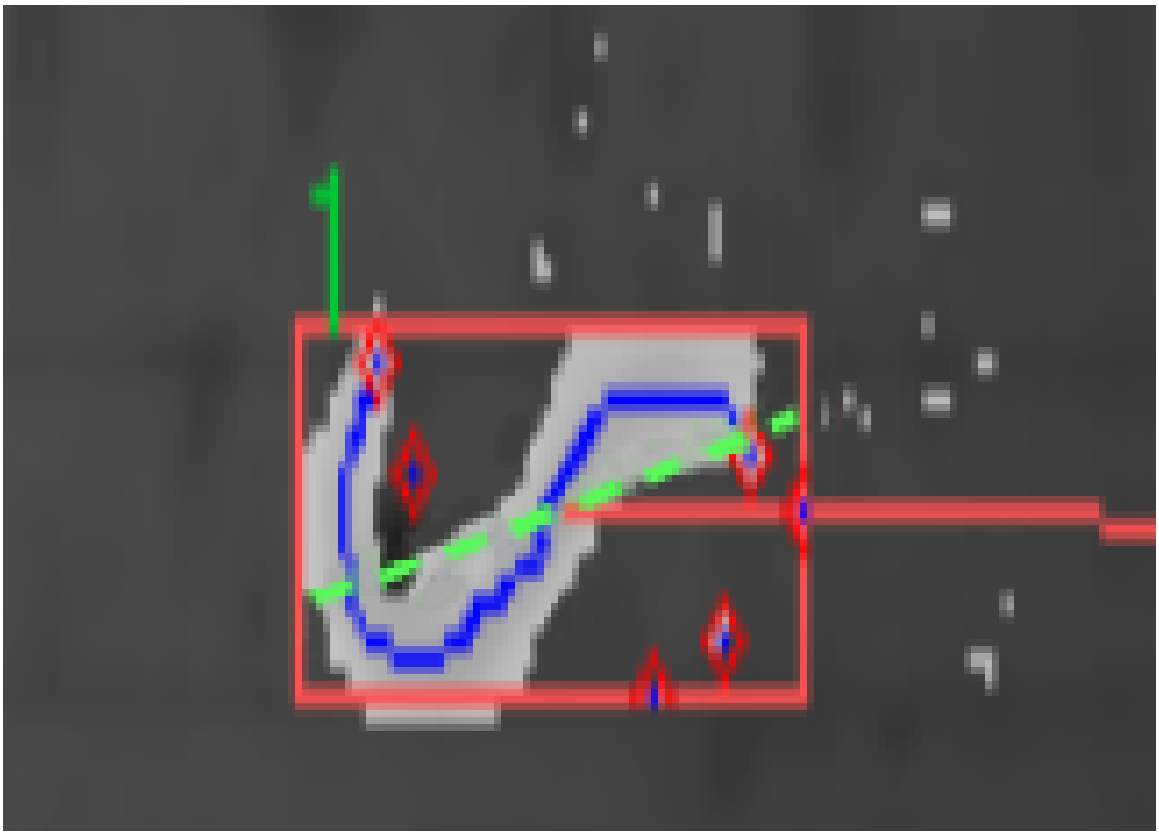


Figure 4.1: A nematode during tracking, showing the bounding box (in red), morphologically thinned curvature (in blue), and a dotted line representing angle (in green).

4.4 Velocity and Acceleration Calculation

The automated tracker extracts the position information relative the top left corner of the video. The velocity and acceleration information can be calculated by taking the first and second derivatives, respectively, of the position. It should be noted, however, that the position information is generally very noisy and is easily affected by outliers. A local regression smoothing function is therefore applied to position values, which reduces the effects of outliers and allows for more accurate modelling.

The output x-position compared with the ground truth, found manually, are shown in Figure 4.2 and the resulting output velocity graph can be seen in Figure 4.3.

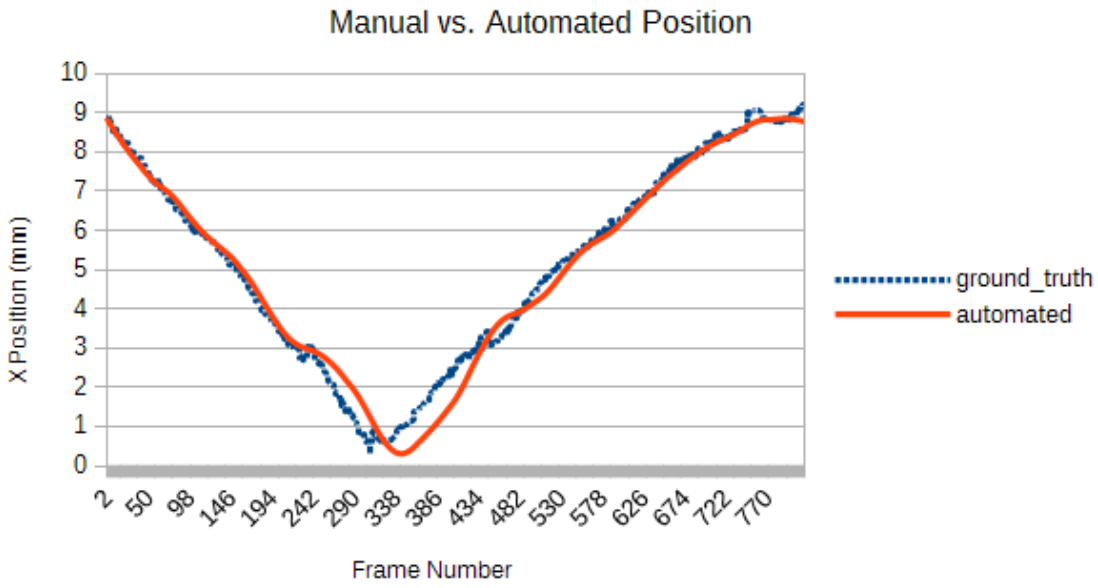


Figure 4.2: The x-position ground truth found manually compared to the automated software output.

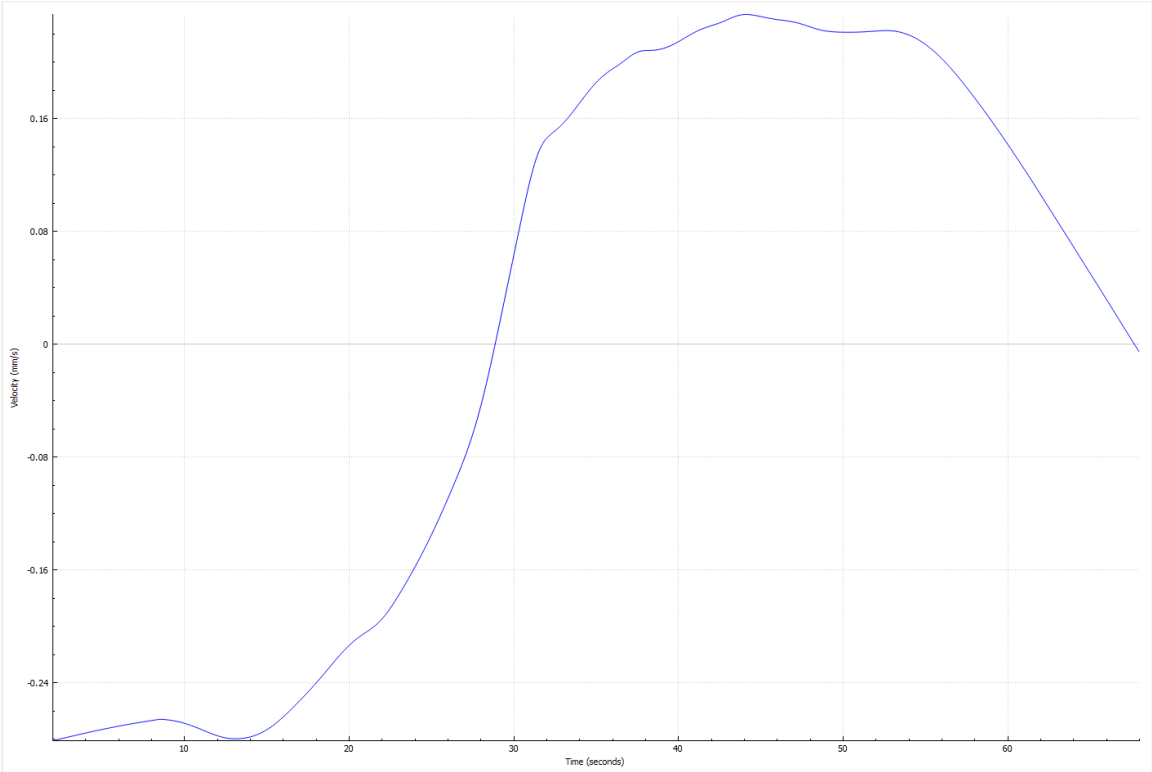


Figure 4.3: Output graph of tracked velocity from the software.

4.5 Amplitude Calculation

The instantaneous amplitude of the nematode's motion is determined by drawing a line to connect the two endpoints of the nematode, and measuring the pixel distance between the highest and lowest point on the nematode's body relative to this line, as shown by the green line in Figure 4.1.

To extract the amplitude information in the micro-channel setup, it was enough to just measure the height of the bounding box as the worms were restricted in motion in that direction in the channel. However, such an approach will not work for general cases, so amplitude can more accurately be extracted for worms in any orientation

by measuring the distance from the centerline, defined as a straight line from head to tail of the worm, to furthest point on the curve.

4.6 Frequency

Frequency information can be extracted by measuring the amplitude over time and then finding the number of critical points within the specified period of time.

4.7 Average Velocity

ANTS returns the total average velocity of the tracked worm across the entire video, as well as the maximum of each motion segment's average velocity, which is detected between local minima and maxima points of acceleration. The minima and maxima points are filtered by persistence (Kozlov and Weinkauf (2013); Edelsbrunner *et al.* (2002)) to ensure more robust results. Persistence is a measure of how robust a critical point is to local noise, it is measured as the absolute difference between pairs of maxima and minima. This method considers the time when the worm speeds up and later slows down, while being resistant to less important peaks caused by noise. The time that the video starts and ends are also considered "peaks" in acceleration to include videos where the worm is moving at the start or end.

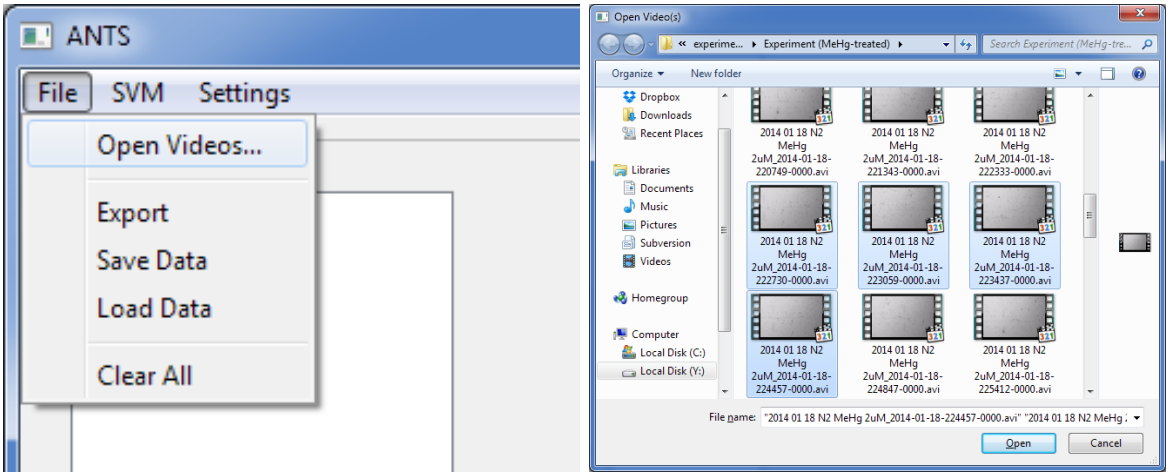
Chapter 5

Program Design

5.1 Usage

ANTS comes with default tracker settings that have been tested to work well for most experiment videos. The only setting that needs to be set per batch of videos is the proper width of the recorded area in millimeters. This can be adjusted under general settings in the options dialog. Each video within the batch needs to have the same width for accurate results, but resolution can differ between videos without any issues.

Videos can then be opened within ANTS (under File...→Open Videos), as shown in Figure 5.1a. A dialog window will open, similar to the one in Figure 5.1b, and will allow the user to choose multiple videos. These videos will automatically be loaded and analyzed when Open is pressed.



(a) (b)

Figure 5.1: ANTS File Menu(a), Open Video(s) Dialog (b).

5.1.1 Tracking and Plot Interface

The interface has been designed for simplicity and to aid the researcher in the use of the ANTS software tool.

The analysis interface is shown in Figure 5.2. Progress bars for each loaded videos are shown on the left. Hovering the mouse over the progress bar will show the file name that the bar is associated with. Double clicking on the progress bar will show the worm video being tracked, or display the plot and results interface as shown in Figure 5.3.

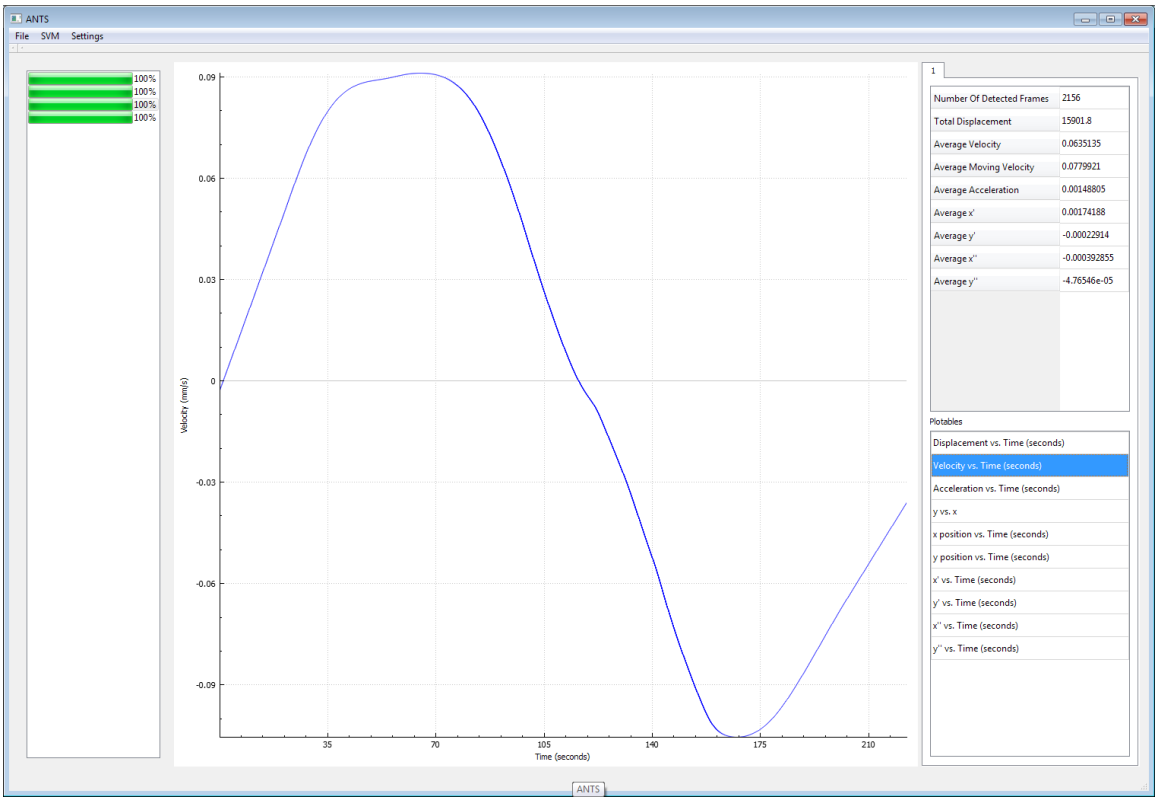
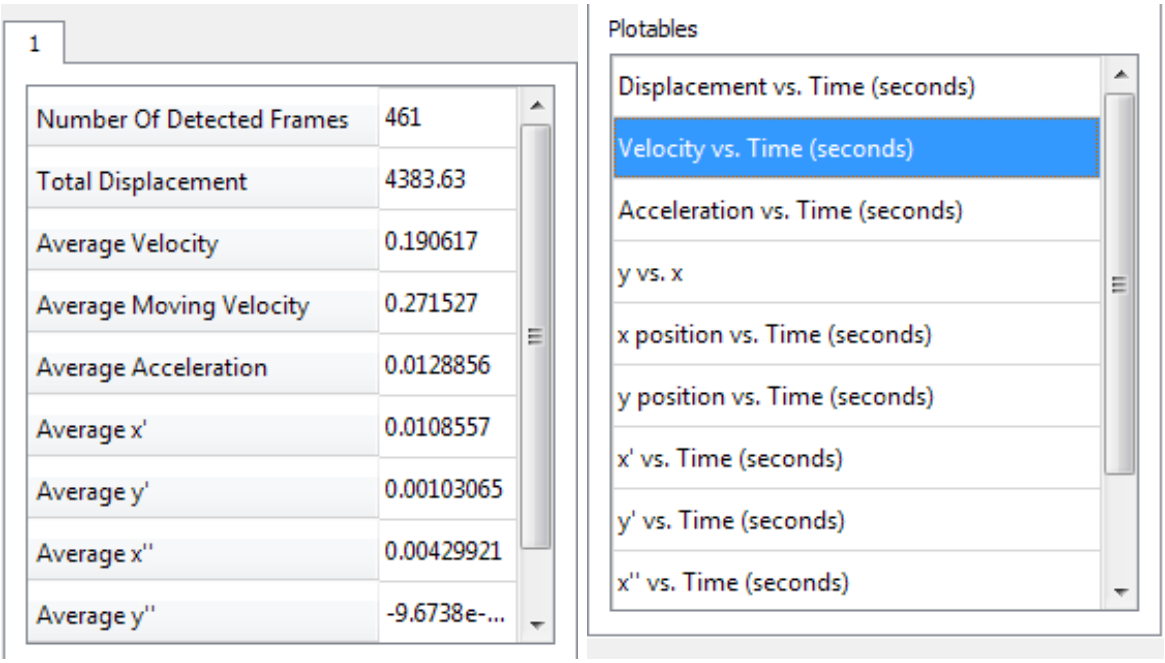


Figure 5.2: ANTS Plotting and Results Interface. (note: not all features are present in this version)

The plot and result interface on the right displays scalar values on the top table and plottable values on the bottom. Clicking on the plottable values will display the graph selected on the main display window in the centre. Closeups of the outputs can be seen in 5.3



(a) (b)

Figure 5.3: Scalar Results(a), Plottables (b).

5.2 Frameworks and APIs Used

ANTS is written with C++ and uses the Qt 5 cross-platform application framework to power its graphical user interface. This interface, as shown in Figures 5.4, is designed to provide convenient feedback to the researcher and make processing videos simpler to understand. It is uncluttered and labeled, making it intuitive and easy to learn. Progress bars on the left show the tracking progress of the videos. Tracking errors that do occur will be easy for the user to detect, and tracking related parameters can be changed between runs if necessary within the options menu.

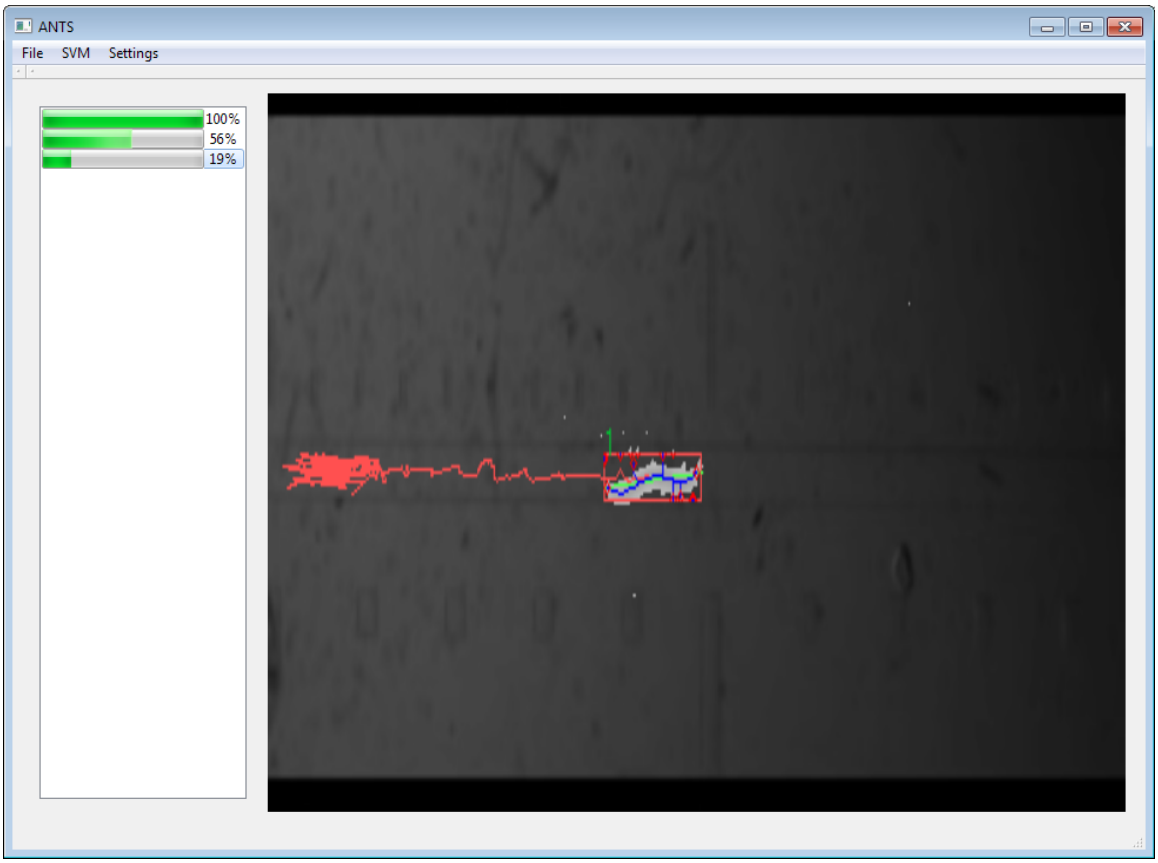


Figure 5.4: Worm Tracking Interface.

In addition, the QCustomPlot library is used to plot the data extracted by ANTS. Data processed does not need to be plotted with external software, though the raw data is exported as well as a CVS file.

5.2.1 OpenGL

OpenGL is an API for rendering both 2D and 3D graphics to the screen. In the case of ANTS it is used directly to render video to the user. Technical details on how the OpenGL is used described when detailing the CQtOpenCVViewerGl class.

GLSL Shaders

GLSL is a programming language that allows direct manipulation of the different stages of the graphics pipeline. ANTS only requires fairly basic GLSL shaders in order to display the video to the user. It is compiled and loaded into the graphics chip as part of the initialization process of ANTS.

5.2.2 OpenCV

This is an image and computer vision library written in optimized C++. It contains many useful and optimized functions for analyzing images and videos. It also contains useful matrix and math functions that have uses for image processing.

5.2.3 Qt Application Framework

Qt is an application cross-platform framework currently developed and owned by Digia. It provides a abstraction for creating graphical applications. Qt supports many different platforms by providing a common API to many display protocols.

Widgets

Widgets are the main abstraction for creating user interfaces with Qt. They can receive and display information from and to the user. They are modular graphical elements that are used together to create an overall interface.

Metaobject compiler

The Metaobject compiler, or moc, is Qt's preprocessor that generates C++ code form custom Qt macros with additional information. It is used by Qt to extend the

capabilities of C++ and allow for language constructions such as the signals and slots system which ANTS uses extensively.

Signals and Slots System

The signals and slots system allows the interface widgets to send information to other objects in a program in a controlled and loosely coupled manner. These signals and slots can be reassigned at runtime and are a useful tool for interface design. They can also be used for a Qt created thread to communicate with other threads.

5.3 Program Structure

The following section discusses the design decisions and general software organization of ANTS.

5.3.1 Model-View-Controller

Model-View-Controller, as described in Gamma *et al.* (1994), is a programming design pattern for user interfaces which separates the concerns of the data from the output display and data manipulation methods. ANTS uses this design pattern throughout for handling and displaying the data. The basic description of the design of a model-view-controller is as follows.

Model The model is where all logic that concerns how the data is stored is located. It Signals the view when changes to the underlying data occur.

View The view interprets and displays the model to the user.

Controller The controller is the logic that the user interacts with and can signals the underlying model to change data.

5.3.2 Main Window

The main window class has the responsibility of starting up and closing the main interface to the user and all other graphical widgets. It holds the track manager and is required to respond to completed process signals from videos that have finished their analysis. The dependency structure of Main Window can be seen in Figure 5.5.

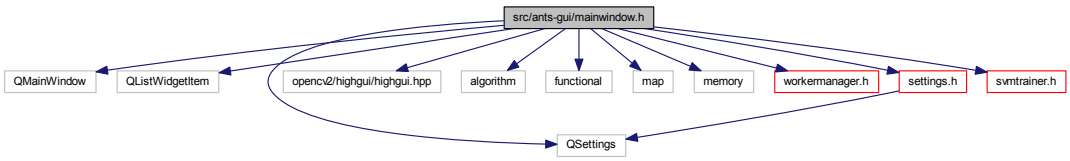


Figure 5.5: Main Window dependencies

The Main Window allows for the interaction with ANTS. It handles the interface aspects of opening videos and signals the objects that will handle analysis and display. It is also responsible for allowing the user to access the settings dialog.

The main window is a top level file that only is included by the main.cpp, its source file, and the automatically generated Metaobject compiler source.

The main.cpp file is only responsible for starting and initializing the initial conditions of the software.

The important Main Window class members to note for Main Window's main functionality is shown in Listing A.1. Important functions and data members it this class are as follows:

Functions

on_actionOpen_triggered() This slot function will receive a signal when the user pushes the Open Videos button on the main interface, described within the mainwindow.ui file, which defines the general graphical layout. It creates the file dialog for the user to select videos to be analyzed and sends this information to the Main Window's workerManager object.

errorModel(QString, QProgressBar*) This function is signaled if there are any issues within the processed thread. The QString will denote which video had the error and the ProgressBar pointer will allow the error to be reported to the user in the interface.

Variables

workerManager This references the worker manager class that handles the creation, connections, completion of threads, and creating their associated widgets.

fileList Is a String list of the video files currently loading within ANTS.

5.3.3 Worker Manager

The worker manager class is responsible for queuing and managing the worm video analysis tasks that need to be processed. The total number of threads it can create is configurable within the settings. The default value is equal to the number of cores on the computer minus one. It also connects the signals and slots of the main window and worm data widget objects to the newly created threads. The dependency structure of worker manager can be seen in Figure 5.6.

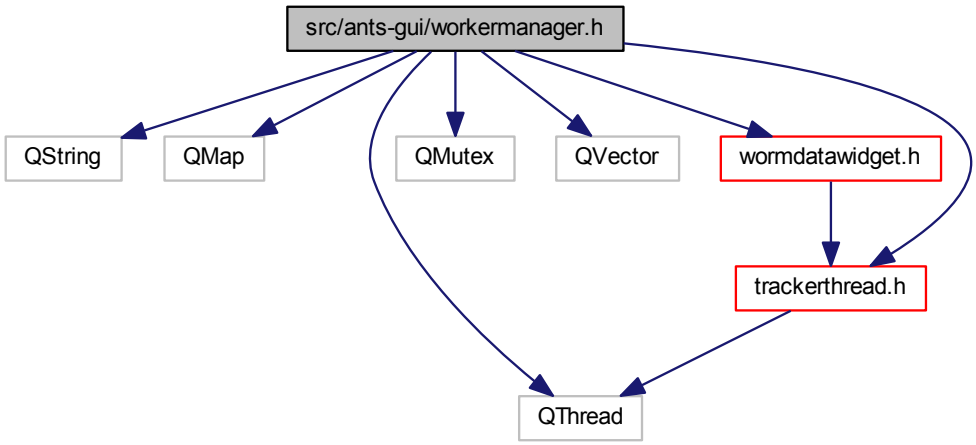


Figure 5.6: Worker Manager header dependencies

The worker manager depends on Qt's included threading functionality and the worm data widget class as it will be signalling it.

The worker manager is sent a pointer to a worm data widget class from the Main Window when new videos are input into ANTS. The worm data widget is the widget for which this process will output information to.

Within Listing A.2, the important class members are described as follows:

Functions

setUpTracker(QString fileName, QProgressBar* prog, WormDataWidget* widget)

This function sets-up and queues a new tracker process to analyze the video at the filepath "fileName". It also connects the process to slots within a worm data widget the Main Window object, and to the **finishThread()** slot. Then the private function **runThread** is called.

runThread Runs the next thread that has been queued up if the maximum number of threads are not currently already running.

Variables

threadQueue Holds the name of the associated threads that are currently running.

threadMap Holds the information for all threads, queued, active, and finished.

5.3.4 Worm Data Widget

The worm data widget is responsible for displaying the worm data within the main window. It receives a signal from the associated video analysis thread when the thread has finished processing. The worm data widget then spawns a new table-view and a display widget to show the extracted worm data.

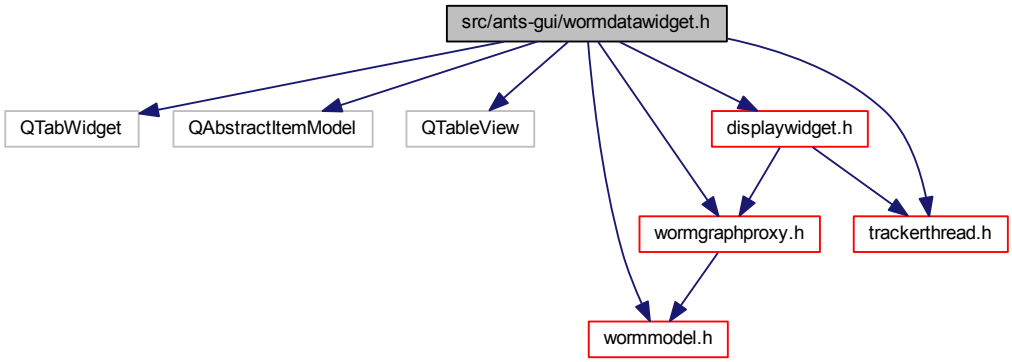


Figure 5.7: Worm Data Widget header dependencies

Shown in Figure 5.7 are the dependencies that worm data widget requires. In addition to the those already mentioned, the worm data widget creates the tabs for

each worm in an individual video. The QAbstractItemModel allows this class to access elements of the worm model for initial setup. Inclusion of the Worm Graph Proxy class allows the worm data widget to use the worm model in plotting the data.

As shown in Listing A.3 the important members of this clase are:

Functions

changeModel(WormModel* model) This slot is connected to the associated video tracking thread, and will receive a signal when the thread completes its execution. It will then configure this object and call **setupTabs()**.

setupTabs() This private function sets up and creates the tab Widget and display widget for this object.

Variables

tabWidget Is a pointer to QTabWidget created in the worm data widget class constructor.

5.3.5 Display Widget

The display widget is responsible for displaying either the video of the worm being tracked or a selected plot depending on whether the worm's analysis process has completed.

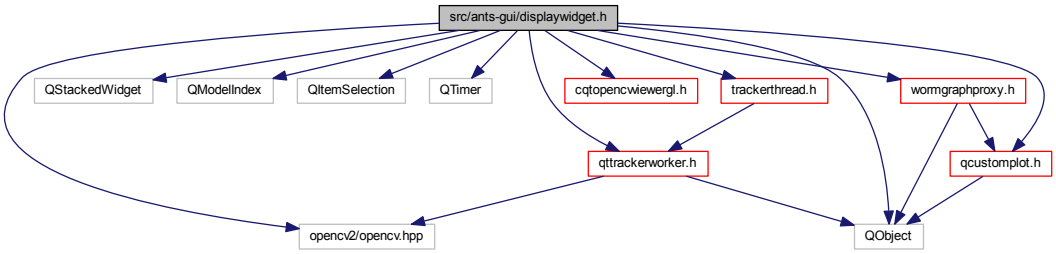


Figure 5.8: Display Widget header dependencies

Shown above in Figure 5.8 are the classes that the Display Widget depends on. The important dependencies are the OpenCV viewer class, which is required to display the worm video; the tracker worker information, which is needed to know what process this display widget belongs to; the QCustomPlot class, which handles the plotting capabilities of the display widget; and the worm graph proxy class, which gives the information for the custom plot to plot the data.

From the code snippet shown in Listing A.4 important class members to note are:

Functions

setModel(WormGraphProxy*) Sets the data model that holds the graph information.

dataSelected(QModelIndex const&) This slot draws the plot based on where the QModelindex points to. This slot is connected to the table view that displays the graph-able data to the user in Worm Data Widget.

updateFrame() This slot displays the image when it receives a signal.

videoAvailiable(cv::Mat img) This slot is signaled by the video worker that is associated with this object. It sets up the next frame to be displayed and sets the **videoFlag** variable to true;

Variables

timer Timer is used to display the video at a maximum of 24 frames per second. This QTimer object signals the **updateFrame()** slot. The timer is first started when the **startVideo()** function is called, resets after a frame is displayed, and is stopped if **stopVideo()** is called or when the video has ended.

video This holds the widget that sets up the OpenGL context and displays the video. The CQtOpenCVViewerGl class will be discussed in more detail later.

model This member is a non-owning Pointer to the plot model proxy associated with this display. The Worm Graph class will be discussed in a later section.

5.3.6 Worm Model

The worm model is part of the model-view-controller pattern mentioned earlier. When the Worm Tracker Thread completes its analysis, a worm model is created and signals all views that use this model.

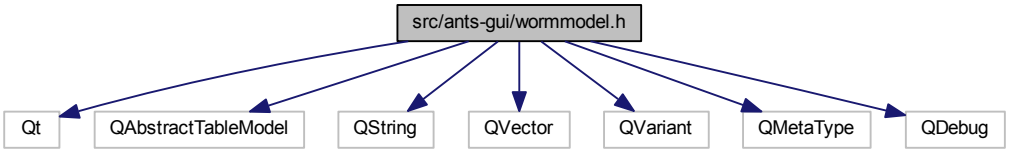


Figure 5.9: Worm Model header dependencies

The include structure of the worm model is shown in Figure 5.9. Dependency-wise this class requires only a few select Qt classes in order to set up the model.

The code snippet shown in Listing A.5 shows the general overview of the worm model class. The important member functions to note for the worm model's main functionality are:

Functions

QVariant headerData(...) This function returns the text labels as shown in the vertical axis of the table in Figure 5.3a and as the content of the table shown in Figure 5.3b. It is an overridden function of QAbstractModel.

bool append(...) This function and the two private helper functions it calls are used as a convenient method to setup the model. It is called after the video thread has completed its analysis.

save(QString) This function serializes the data in the model and saves it to a file.

load(QString) This function loads the serialized file back into the model.

exportFile(QString) This function saves the information in the model into a csv file.

Variables

worms This member variable is the underlying data structure of the worm analysis that the worm model abstracts.

5.3.7 Worm Graph Proxy Model

This function is similar to the worm model class except that it uses the a worm model function to populate its data. It allows the user to see the data from the worm model in new ways. In the case of the worm graph proxy model it shows the plottable data in plotted form.

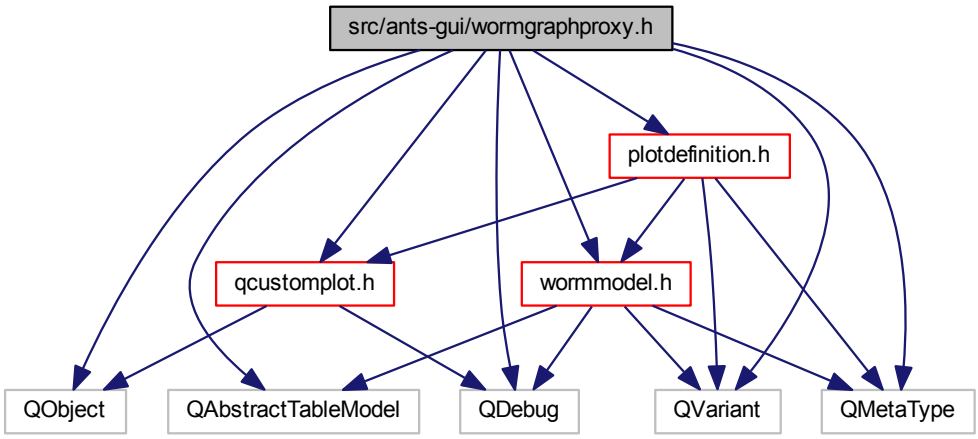


Figure 5.10: Worm Graph Proxy dependencies

Seen in Figure 5.10 the worm graph proxy requires a plotting library, in this case QCustomPlot, and the worm model. It also uses the plot definition class which will be explained later.

The code snippet shown in Listing A.6 shows the general class layout and is explained as follows:

Functions

setDefinitions(...) This function defines what kind of plots need to be generated from the input vector of PlotDefinition data types.

getPlot(QModelIndex const&) Retrieves a plot from an index to a worm graph proxy model.

processSourceModel() A private function that is called internally in order to generate all the plots from the input source model.

Variables

plotDefinitions This member defines what is plotted and how the plots will look like.

model The generated plots are pre-computed and stored in the model member variable data structure.

sourceModel The source worm model which is used as the underlying data to plot the data.

5.3.8 Tracker Worker

The tracker worker is the function that analyses the worm data. It is responsible for tracking the worm, formatting, and returning the data for the worm model.

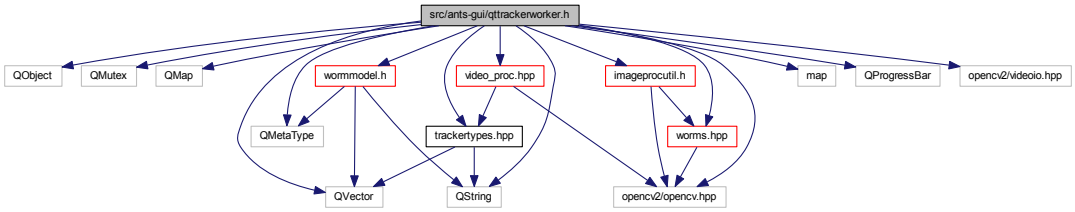


Figure 5.11: Tracker Worker header dependencies

As shown in Figure 5.11 the tracker worker requires the worm, imgproutil, and vid_proc dependencies, which contain the tracking algorithm, as well as the OpenCV library. The tracker worker class is the process that is spawned off the main thread when analyzing a video.

The important member functions to note for the tracker worker class from Listing A.7 are as follows.

Functions

convertToQMap(Track&) This function formats, analyses, filters the data, and returns a worm structure data type.

convertToQVector(Tracks) This function formats the data into a QVector of indidual Worm data from the same video.

createModel(Tracks) This function creates and populates the worm model with the extracted data.

finished() This function signals the worker manager that the thread has completed.

returnModel(WormModel*) This function signals associated classes with a pointer to the final worm model retrieved from analysis.

error(QString, QString) This function signals classes that need to be aware that the process has failed. This prevents the entire program from crashing if there has been an error in analysis.

errorModel (QString,QProgressBar*) This function signals main to display to the user through the progress bar text that there has been an error with this process.

updateProgress(QProgressBar*, int vale) This signals the main window to update the progressbar with the current progress value.

videoAvailiable(cv::Mat) This is a signal to the associated display widget function which will display the frame.

process() This is signaled when the containing thread leaves the process queue in the worker manager.

Variables

mVideo This is an OpenCV video capture object for opening and reading the image. Each frame is returned in OpenCV's Matrix format for easy manipulation.

progress This is a pointer to the progress bar from the main window function associated with this process.

5.3.9 OpenCV-OpenGL Viewer

This is a custom created class to view the images within an OpenGL rendering context.

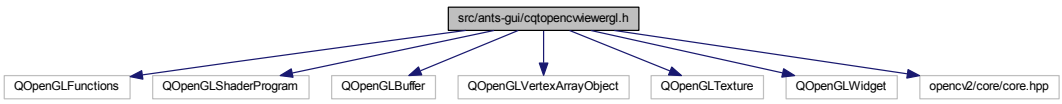


Figure 5.12: Tracker Worker header dependencies

As shown in Figure 5.12 this function uses the exposed OpenGL API interface within Qt. This allows a few convenience functions but they are fairly unabstracted low level OpenGL calls.

The important member functions to note from Listing A.8 are:

Functions

initializeGL() Initializes the OpenGL context and compiles the basic OpenGL GLSL shaders to display the image frames. It then calls the private **LoadVideoArea()** function which sets up the video draw area, binds it to a vertex array object, and sets up the attribute arrays.

renderImage() This function renders the image into the OpenGL context.

Variables

vbo The vertex buffer object holds the image area and texture coordinates to be sent to the graphics processing unit.

texture This is the OpenGL texture object which is used on the overlaid on the video area. This is the video frame image to be displayed.

program Pointer to the loaded OpenGL shader program.

5.3.10 PlotDefinition

This class is handles the instructions to plot the data in a functional and centralized manner. It is used by the graph proxy model in order to plot the data.

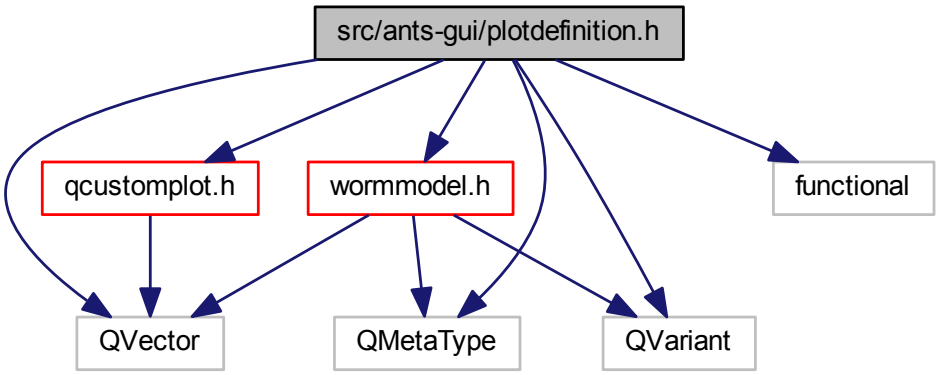


Figure 5.13: Tracker Worker header dependencies

As shown in Figure 5.13 the plot definition requires knowledge of the worm model class in order to be able know how to reference it when accessing the plot.

The important member functions to note from Listing A.8 are:

Functions

setFunction(...) This is where the the class member curve is set.

Variables

curve A functional object that defines how the graph looks and what data it uses from the worm model.

Chapter 6

Results

Traditionally, features are extracted from each video using manual measurements by trained researchers. To measure the accuracy of the automated tracking, the features extracted by the software are compared with those manually extracted by researchers at the Gupta Life Science's lab at McMaster0.

The amount of deviation in the tracking results can increase during corner cases, particularly with slow worms, immobile worms, or multiple worms in close proximity. Variations from the manually chosen time-span used for calculating the average velocity also impact the deviation between the software and manual results.

A summary of average velocity results from both manual analysis and processing using ANTS is shown in Table 6.1. Detailed results for each strain of nematode may be viewed in Tables B.1 – B.7. We briefly discuss these results below, as well as reasons for deviations in tracker performance between video sets.

Video Set	**Deviation (%)**	**Number of Videos**	**Average Video Duration**	**Processing Time Per Video**	**Resolutions**
N2	15.85	10	01:21.69	00:24.92	320 x 240
N2 Cu 50µM	26.08	19	01:12.55	00:15.79	320 x 240
CB1420 mutant	38.09	25	04:11.98	01:06.96	320 x 128
MeHg treated	16.92	40	02:19.93	00:27.96	320 x 128
fc2	19.65	32	03:07.40	00:37.00	1920 x 780
larval stage 30hrs	21.03	23	02:50.31	00:30.05	1288 x 516
larval stage 48hrs	21.70	20	01:30.96	00:17.82	1288 x 720

Table 6.1: Average velocity accuracy for each set of videos.

6.1 Results from Mutant Analysis

The software has proven to be capable of analyzing the motion of mutant worms consistently and accurately with few exceptions. Mutant worms may have different results depending on the speed of the tracked worm. ANTS specifically looks for moving objects within the video, so stationary worms can sometimes be mistaken as part of the background. In the electrotaxis microchannel, MeHg treated worms were accurately analyzed within 17% of the manual analysis, as shown in Table B.3.

CB1430 unc-40(e1430) experiment videos are analyzed less accurately, with an average deviation of 38% from manual analysis, making the analysis of the CB1430 mutants the least accurate of all batches analyzed. The software's total average velocity in these cases was closer to the measured result, implying that the software picked a different time segment to average over for measuring average moving velocity.

RM2702 dat-1(ok157) experiment videos were analyzed with an average accuracy of about 20% of manual analysis.

6.2 Results from Different C. elegans Development Stages

ANTS handles tracking larval stages with similar accuracy to adult worms. ANTS analyzed larval stage N2 worms at 30 and 48 hour development stages, with resulting accuracies within 21% and 22% of the manual analysis, respectively. Full results can be observed in Tables B.6 and B.7.

Chapter 7

Conclusion

Using ANTS to automate experimental measurements and to generate and visualize data in different ways has huge impacts on overall researcher workflow and productivity. The ability to retrieve results quickly allows for considerably faster experimental iteration.

Human measurements can also be error prone and difficult to catch, while software errors can usually be deduced fairly easily by quickly viewing the output. Software results will be more consistent between runs, and have the capability to retrieve finer position information in less time than manual analysis.

With the added QT graphical user interface, a variety of useful analysis information is conveniently displayed, and researchers new to ANTS can quickly learn and use the software with very little effort. Additional analysis can be preformed on the data using external software if necessary, so custom analysis and interpretation of the data is possible.

The ANTS framework is also open source and readily extensible for adding further automated features in the future. Furthermore, its modular architecture makes it

easy to substitute alternative algorithms at each processing step to improve future performance.

Some important future work that would further improve the software would be by adding a scripting system. This would allow researchers to easily add new functionality, types of plots, and set up new ways to measure the worms as they are being tracking without needing to modify the C++ code. Another future feature would be adding the ability to handle occlusion cases, when two worms overlap, and accurately track those cases.

With its powerful features and ease of use, ANTS shows strong potential for widespread adoption in *C. elegans* research.

Appendix A

Code Listings

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private slots:
    void on_actionOpen_triggered();
    void on_wormVideoList_itemActivated(QListWidgetItem *item);
    void on_actionOptions_triggered();
    void on_actionTrain_triggered();
    void on_actionSave_All_triggered();
    void on_actionLoad_Data_triggered();
    void on_actionClear_All_triggered();

    void newModel(WormModel*);
    void updateProgressBar(QProgressBar*, int);
    void errorModel(QString, QProgressBar*);

    void on_actionExport_triggered();

private:
    Ui::MainWindow* ui;
    WorkerManager workerManager;
    QStringList fileList;
    QSettings settings;
    QVector<WormModel*> modelRefs;

    // Settings related functions
```

```
30      void loadSettings();
31      void saveSettings();
32  };
```

Listing A.1: mainwindow.h code snippet

```
1   class WorkerManager : QObject
2   {
3       Q_OBJECT
4
5   public:
6       WorkerManager(QWidget*);
7       ~WorkerManager();
8
9       void setUpTracker(QString fileName, QProgressBar* prog,
        WormDataWidget* widget);
10      TrackerThread* tracker(QString fileName);
11      bool displayable(QString fileName);
12
13  private slots:
14      void finishThread();
15      void errorString(QString, QString);
16
17  private:
18      void runThread();
19
20      int runningThreads;
21      int maxThreads;
22      QVector<QString> threadQueue;
23      QMap <QString, TrackerThread*> threadMap;
24      QWidget* parent;
25  };
```

Listing A.2: wormManager.h code snippet

```
1   class WormDataWidget : public QSplitter
2   {
3       Q_OBJECT
4   public:
5       WormDataWidget(QWidget* parent=0);
6       ~WormDataWidget();
7       void setTracker(TrackerThread*);
8       void display();
9       void stopDisplay();
10
11  public slots:
12      void changeModel(WormModel* model);
```

```
13
14 signals:
15     void selectionChanged (const QItemSelection& selected);
16     void modelChanged(WormModel* model);
17     void videoAvailiable(cv::Mat);
18
19 private:
20     void setupTabs();
21     void reset();
22
23     QWidget *parent;
24     DisplayMode mode;
25     WormModel* model;
26     WormGraphProxy* graphProxy;
27     QTabWidget* tabWidget;
28     DisplayWidget* displayWidget;
29 };
```

Listing A.3: wormDataWidget.h code snippet

```
1  class DisplayWidget : public QStackedWidget
2  {
3      Q_OBJECT
4
5  public:
6      DisplayWidget(QWidget* parent=0);
7      ~DisplayWidget();
8
9      void startVideo();
10     void stopVideo();
11
12 public slots:
13     void setModel(WormGraphProxy* model);
14     void dataSelected(QModelIndex const& index);
15     void clearPlot();
16
17 private slots:
18     void updateFrame();
19     void videoAvailiable(cv::Mat img);
20
21 private:
22     QTimer* timer;
23     CQtOpenCVViewerGl* video;
24     WormGraphProxy* model;
25     bool videoFlag;
26     cv::Mat displayImg;
27 };
```

Listing A.4: displaywidget.h code snippet

```
class WormModel : public QAbstractTableModel
{
public:
    WormModel(QObject* parent = 0);
    ~WormModel();

    int rowCount(const QModelIndex &parent=QModelIndex()) const;
    int columnCount(const QModelIndex &parent=QModelIndex()) const;
    QVariant data(const QModelIndex &index, int role=Qt::DisplayRole)
    const;
    QVariant headerData(int section, Qt::Orientation orientation, int
    role) const;
    Qt::ItemFlags flags(const QModelIndex &index) const;
    bool setData(const QModelIndex &index, const QVariant &value, int
    role = Qt::EditRole);
    bool insertRows(int row, int count, const QModelIndex &index =
    QModelIndex());
    bool removeRows(int position, int rows, const QModelIndex &index =
    QModelIndex());

    void clear();
    QVector<QPair<int, QVector<QPair<QString, QVariant>>>> getData();
    void setfileName(QString);
    QString getFileName();

    template<typename... Args>
    bool append(int const& label, Args const&...  data)
    {
        const bool noRows = this->rowCount(QModelIndex()) <= 0;
        const int colIdx = this->columnCount(QModelIndex());
        const int rows = sizeof...(Args)/2 - 1;
        if(noRows) beginInsertRows( index(0,rows), 0, rows);
        beginInsertColumns( index(colIdx,colIdx), colIdx, colIdx);
        QVector<QPair<QString, QVariant>> vec = appendHelper(QVector<
    QPair<QString, QVariant>>(), data...);
        worms.append( qMakePair(label,vec) );
        endInsertColumns();
        if(noRows) endInsertRows();
        return true;
    }

    int getRow(QString rowName);

    bool save(QString filePath);
    bool load(QString filePath);
    bool exportFile(QString filePath);

private:
```

```
43      QVector<QPair<QString, QVariant>> appendHelper(QVector<QPair<QString
        , QVariant>> vector){return vector;}
44      template<typename T, typename... Args>
45      QVector<QPair<QString, QVariant>> appendHelper(QVector<QPair<QString
        , QVariant>> vector, QString const& name, T const& value, Args const
        &... data)
46      {
47          QVariant varValue;
48          varValue.setValue(value);
49          QPair<QString, QVariant> pair(name, varValue);
50          vector.append(pair);
51          return appendHelper(vector, data...);
52      }
53
54      // Data
55      QString fileName;
56            //worm label       // Pair of Variable Name and Value
57      QVector<QPair<int, QVector<QPair<QString, QVariant>>>> worms;
58  };
```

Listing A.5: wormModel.h code snippet

```
1   class WormGraphProxy : public QAbstractTableModel
2   {
3   public:
4       WormGraphProxy(QObject* parent=0);
5       ~WormGraphProxy();
6
7       void setSourceModel(QAbstractItemModel* sourceModel);
8
9       int rowCount(const QModelIndex &parent=QModelIndex()) const;
10      int columnCount(const QModelIndex &parent=QModelIndex()) const;
11      QVariant data(const QModelIndex &proxyIndex, int role = Qt::
        DisplayRole) const;
12      Qt::ItemFlags flags(const QModelIndex &index) const;
13
14      QVariant headerData(int section, Qt::Orientation orientation, int
        role) const;
15      QMap<int, QVariant> itemData(const QModelIndex &index) const;
16      void revert();
17      bool setData(const QModelIndex &index, const QVariant &value, int
        role);
18      bool setHeaderData(int section, Qt::Orientation orientation, const
        QVariant &value, int role);
19      bool submit();
20
21      void setDefinitions(QVector<PlotDefinition> definitions);
22      QCustomPlot* getPlot(QModelIndex const& proxyIndex);
```

```
23
24 private:
25     QVector<PlotDefinition> plotDefinitions;
26     QVector<QVector<QCustomPlot*>> model;
27     QAbstractItemModel* sourceModel;
28
29     void processSourceModel();
30
31 };
```

Listing A.6: wormGraphProxy.h code snippet

```
1  class QtTrackerWorker : public QObject
2  {
3      Q_OBJECT
4  public:
5      explicit QtTrackerWorker(QString filepath, QProgressBar *prog,
       QWidget *parent = 0);
6      ~QtTrackerWorker();
7
8      struct SharedImage
9      {
10         QMutex mutex;
11         cv::Mat image;
12     };
13
14     QWidget* parent;
15     SharedImage shared;
16     const QString fileName;
17
18 private:
19     cv::VideoCapture mVideo;
20     Worm convertToQMap(Track&);
21     Worms convertToQVector(Tracks);
22     WormModel* createModel(Tracks);
23     QProgressBar* progress;
24
25 signals:
26     void finished();
27     void returnModel(WormModel* model);
28     void error(QString, QString);
29     void errorModel(QString,QProgressBar*);
30     void updateProgress(QProgressBar*, int value);
31     void videoAvailiable(cv::Mat);
32
33 public slots:
34         void process();
```

```
};
```

Listing A.7: trackerworker.h code snippet

```
class CQtOpenCVViewerGl : public QOpenGLWidget
{
    Q_OBJECT

public:
    explicit CQtOpenCVViewerGl(QWidget *parent = 0);
    ~CQtOpenCVViewerGl();

signals:
    void imageSizeChanged( int outW, int outH ); /// Used to resize the image outside the widget
    void drawImage( cv::Mat image );

public slots:
    bool showImage( cv::Mat image ); /// Used to set the image to be viewed
    bool showImage( QImage image );

protected:
    void initializeGL(); /// OpenGL initialization
    void paintGL(); /// OpenGL Rendering
    void resizeGL(int width, int height);        /// Widget Resize Event

    void renderImage();
    void updateScene();

private:
    void LoadVideoArea();
    static const QString vertexSource;
    static const QString fragmentSource;


    bool mSceneChanged;           /// Indicates when OpenGL view is to be redrawn

    QImage mRenderQtImg;            /// Qt image to be rendered
    cv::Mat mOrigImage;              /// original OpenCV image to be shown

    QColor mBgColor;     /// Background color

    int mOutH;                    /// Resized Image height
    int mOutW;                    /// Resized Image width
    float mImgRatio;               /// height/width ratio
```

```
41
42      int mPosX;                  /// Top left X position to render image
        in the center of widget
43      int mPosY;                  /// Top left Y position to render image
        in the center of widget
44
45      QOpenGLVertexArrayObject vao;
46      QOpenGLBuffer vbo;
47      QOpenGLTexture texture;
48      QOpenGLShaderProgram* program;
49
50  };
```

Listing A.8: cqtopencvviewergl.h code snippet

```
1   class PlotDefinition
2   {
3   public:
4       PlotDefinition();
5       PlotDefinition(std::function<QVector<QCustomPlot*>(PlotDefinition*,
        WormModel*)> func);
6       ~PlotDefinition();
7
8       QString name;
9
10      void setRow(int row);
11      void setRow(QString rowName);
12      void setFunction(std::function<QVector<QCustomPlot*>(PlotDefinition
        *, WormModel*)> func);
13      QVector<QCustomPlot*> getPlottableRow(WormModel* model);
14      QVector<QString> selectedRows() const;
15      void setName(QString const& name);
16
17  private:
18      QVector<QString> rows;
19      std::function<QVector<QCustomPlot*>(PlotDefinition*, WormModel*)>
        curve;
20
21  };
```

Listing A.9: plotdefinition.h code snippet

Appendix B

Result Tables

Manual Data gathered by the McMaster Gupta Lab.

Video ID	Manual (mm/s)	Automated (mm/s)	Error (%)
162723	0.1716	0.1443	15.88
162955	0.1993	0.1652	17.13
163230	0.1866	0.1467	21.37
163459	0.2959	0.2548	13.88
163720	0.1518	0.1407	7.32
163936	0.2239	0.1652	26.22
164232	0.2536	0.1998	21.20
164458	0.1918	0.2013	4.98
164710	0.2888	0.2494	13.65
164951	0.3125	0.2597	16.90
		Average:	15.85

Table B.1: N2 Automated vs. Manual Tracking Average Velocity Comparison

Video ID	Manual (mm/s)	Automated (mm/s)	Error (%)
133015	0.2963	0.2287	22.82
133308	0.2348	0.1470	37.39
133457	0.2500	0.1946	22.16
133802	0.3036	0.3361	10.73
133957	0.2851	0.2606	8.60
134327	0.3235	0.2685	17.00
134524	0.2425	0.1525	37.12
134801	0.2292	0.2582	12.69
134922	0.2888	0.2134	26.09
135120	0.2577	0.1705	33.83
135300	0.2397	0.1691	29.48
135452	0.3400	0.3154	7.24
135614	0.2969	0.2584	12.97
140916	0.1898	0.1345	29.10
141159	0.1789	0.1404	21.57
141352	0.1842	0.0692	62.43
141744	0.3804	0.2435	36.00
141940	0.2625	0.2054	21.76
142236	0.3273	0.1751	46.49
		Average:	26.08

Table B.2: N2 Cu50µM Automated vs. Manual Tracking Average Velocity Comparison

Video ID	Manual (mm/s)	Automated (mm/s)	Error (%)	Notes:
201148	0.2615	0.2692	2.94	
201421	0.0281	0.0208	25.82	slow worm
201807	0.095	0.0895	5.78	slow worm
202315	0.0264	0.0232	12.24	partial second worm
202846	0.0528	0.0757	43.30	slow worm
204142	0.1343	0.1232	8.25	slow
204553	0.0463	0.0605	30.71	
205232	0.0561	0.0579	3.30	interference
210132	0.0896	0.0832	7.20	
210616	0.1532	0.1163	24.08	
220302	0.0846	0.0613	27.60	interference
220749	0.0844	0.1155	36.86	interference
221343	0.1027	0.0842	18.05	second worm + interference
222333	0.186	0.1401	24.65	interference
222730	0.1883	0.1666	11.51	interference
223059	0.1286	0.1148	10.72	second worm + interference
223437	0.1705	0.1330	22.01	
224457	0.0676	0.1357	100.67	Many overlapping worms + interference
224847	0.1649	0.1752	6.26	second worm + interference
225412	0.1676	0.1580	5.71	second worm + interference
171808	0.1582	0.1567	0.94	
172246	0.1306	0.1111	14.91	partial second worm
172605	0.133	0.1284	3.43	second worm
173025	0.1285	0.1033	19.65	
173459	0.0953	0.1373	44.07	
174235	0.1392	0.1539	10.56	
174650	0.1516	0.1674	10.43	
175558	0.1926	0.1876	2.60	
180139	0.0708	0.0607	14.27	partially immobile worm
180640	0.1442	0.1368	5.10	
180927	0.1381	0.1329	3.77	
182124	0.1759	0.1719	2.26	
182701	0.1511	0.1733	14.69	
183013	0.0935	0.0777	16.85	second worm
183422	0.1979	0.1591	19.59	
183610	0.125	0.1260	0.80	second worm
184034	0.1401	0.1203	14.11	
184355	0.2019	0.1861	7.83	
184910	0.1471	0.1320	10.28	
		Average:	16.92	

Table B.3: MeHg-Treated Automated vs. Manual Tracking Average Velocity Comparison

Video ID	Manual (mm/s)	Automated (mm/s)	Error (%)
133416	0.0477	0.1150	141.30
135720	0.0241	0.0223	7.54
141300	0.0122	0.0211	72.72
150508	0.0390	0.0776	99.07
151334	0.0370	0.0341	7.87
160500	0.0267	0.0226	15.39
163657	0.0815	0.0758	6.95
171243	0.0244	0.0616	153.00
184659	0.0487	0.0402	17.50
190609	0.0460	0.0512	11.31
191339	0.0610	0.0560	8.17
191847	0.0478	0.0523	9.38
192446	0.0396	0.0283	28.43
192902	0.0396	0.0282	28.91
193449	0.0638	0.0798	25.00
195038	0.0964	0.0848	12.08
195651	0.0785	0.0967	23.09
200426	0.0342	0.0231	32.27
201358	0.0529	0.0595	12.52
202205	0.0725	0.0719	0.76
202740	0.0281	0.0350	24.49
203547	0.0839	0.0693	17.40
204311	0.0435	0.1199	175.82
205231	0.0947	0.0748	20.98
210946	0.0750	0.0751	0.20
		Average:	38.09

Table B.4: CB1430 unc-40(e1430) Automated vs. Manual Tracking Average Velocity Comparison

Video ID	Manual (mm/s)	Automated (mm/s)	Error (%)
165021	0.1392	0.0650	53.34
152851	0.1845	0.1463	20.7
153508	0.1091	0.0969	11.19
155826	0.1593	0.0657	58.77
162955	0.1252	0.1109	11.44
163638	0.1654	0.1372	17.01
164429	0.1807	0.1630	9.8
181137	0.2389	0.2061	13.74
181751	0.1844	0.1746	5.35
165244	0.2308	0.1824	20.98
170611	0.2586	0.2518	2.63
171411	0.3574	0.2880	19.43
171822	0.2864	0.2420	15.49
173621	0.2090	0.2095	0.26
175455	0.2272	0.2068	8.95
180228	0.1716	0.0897	47.71
180403	0.1340	0.0968	27.79
162020	0.1950	0.2116	8.56
163145	0.2022	0.1497	25.97
163522	0.2200	0.1746	20.65
164330	0.2168	0.1506	30.53
165046	0.2471	0.2319	6.15
170409	0.1782	0.1710	4.04
175223	0.1095	0.0825	24.65
162813	0.1983	0.1496	24.55
152931	0.2019	0.1582	21.61
153508	0.1850	0.1233	33.37
154407	0.1825	0.1668	8.6
155525	0.1911	0.1880	1.62
155948	0.1429	0.1757	22.95
160631	0.1779	0.1922	8.05
161948	0.1952	0.1117	42.81
		Average:	19.65

Table B.5: RM2702 dat-1(ok157) Automated vs. Manual Tracking Average Velocity Comparison

Video ID	Manual (mm/s)	Automated (mm/s)	Error (%)
210143	0.1979	0.1658	16.21
210259	0.1814	0.1728	4.76
210730	0.1361	0.1092	19.75
211755	0.1987	0.1001	49.62
212210	0.1160	0.0732	36.89
222144	0.1131	0.0760	32.84
224733	0.1883	0.1478	21.50
230212	0.1668	0.1391	16.58
230823	0.1633	0.1197	26.71
234338	0.0777	0.0654	15.81
235254	0.1090	0.0907	16.74
955	0.1757	0.1227	30.17
222623	0.1738	0.1364	21.51
223208	0.1273	0.1425	11.94
223935	0.1319	0.1429	8.35
225134	0.1494	0.0966	35.32
225757	0.1741	0.1058	39.22
231030	0.0981	0.0690	29.67
232256	0.1124	0.1121	0.26
232901	0.1313	0.1235	5.95
234908	0.0414	0.0386	6.75
235930	0.1240	0.1328	7.10
659	0.1111	0.0777	30.10
		Average:	21.03

Table B.6: Experiment (N2 30hrs larval) Automated vs. Manual Tracking Average Velocity Comparison

Video ID	Manual (mm/s)	Automated (mm/s)	Error (%)
195841	0.2549	0.1560	38.80
200536	0.2859	0.1983	30.63
201550	0.2353	0.2438	3.61
210859	0.2213	0.2173	1.80
211214	0.2715	0.2544	6.28
211625	0.2850	0.1732	39.24
212058	0.3433	0.4776	39.12
212522	0.3889	0.2115	45.60
212820	0.3485	0.3150	9.60
213505	0.3809	0.2558	32.85
215927	0.3644	0.2639	27.57
220323	0.1548	0.1397	9.74
220759	0.2829	0.2010	28.95
221101	0.3379	0.2663	21.19
221715	0.2694	0.1940	27.97
222933	0.2933	0.2745	6.43
223730	0.3138	0.2701	13.95
224559	0.2872	0.2285	20.45
225306	0.3406	0.3201	6.02
232226	0.2967	0.2247	24.27
		Average:	21.70

Table B.7: Experiment (N2 48hrs larval) Automated vs. Manual Tracking Average Velocity Comparison

Bibliography

Altun, Z. F. (2006). C. elegans worm image. Donated to Wikipedia.

Brenner, S. (1974). The genetics of Caenorhabditis elegans. *Genetics*, **77**(1), 71–94.

Chang, F., Chen, C.-J., and Lu, C.-J. (2004). A linear-time component-labeling algorithm using contour tracing technique. *computer vision and image understanding*, **93**(2), 206–220.

Cleveland, W. S. (1979). Robust locally weighted regression and smoothing scatterplots. *Journal of the American statistical association*, **74**(368), 829–836.

Cleveland, W. S. and Devlin, S. J. (1988). Locally weighted regression: an approach to regression analysis by local fitting. *Journal of the American Statistical Association*, **83**(403), 596–610.

Edelsbrunner, H., Letscher, D., and Zomorodian, A. (2002). Topological persistence and simplification. *Discrete and Computational Geometry*, **28**(4), 511–533.

Fontaine, E., Burdick, J., and Barr, A. (2006). Automated tracking of multiple c. elegans. In *Engineering in Medicine and Biology Society, 2006. EMBS'06. 28th Annual International Conference of the IEEE*, pages 3716–3719. IEEE.

Gabel, C. V., Gabel, H., Pavlichin, D., Kao, A., Clark, D. A., and Samuel, A. D. (2007). Neural circuits mediate electrosensory behavior in caenorhabditis elegans. *The Journal of neuroscience*, **27**(28), 7586–7596.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Harada, H., Kurauchi, M., Hayashi, R., and Eki, T. (2007). Shortened lifespan of nematode caenorhabditis elegans after prolonged exposure to heavy metals and detergents. *Ecotoxicology and environmental safety*, **66**(3), 378–383.

Huang, T., Yang, G., and Tang, G. (1979). A fast two-dimensional median filtering algorithm. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, **27**(1), 13–18.

Isard, M. and Blake, A. (1998). Condensationconditional density propagation for visual tracking. *International journal of computer vision*, **29**(1), 5–28.

Kaletta, T. and Hengartner, M. O. (2006). Finding function in novel targets: C. elegans as a model organism. *Nature Reviews Drug Discovery*, **5**(5), 387–399.

Kalman, R. (1960). A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, **82**(Series D), 35–45.

Kozlov, Y. and Weinkauf, T. (2013). Extracting and filtering minima and maxima of 1d functions. `http://people.mpi-inf.mpg.de/~weinkauf/notes/persistence1d.html`.

Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, **5**(1), 32–38.

Ramot, D., Johnson, B. E., Berry, T. L., Carnell, L., and Goodman, M. B. (2008). The Parallel Worm Tracker: a platform for measuring average speed and drug-induced paralysis in nematodes. *PloS one*, **3**(5), e2208.

Rasband, W. (1997-2014). Imagej.

Reid, D. B. (1979). An algorithm for tracking multiple targets. *Automatic Control, IEEE Transactions on*, **24**(6), 843–854.

Rezai, P., Siddiqui, A., Selvaganapathy, P. R., and Gupta, B. P. (2010). Electrotaxis of caenorhabditis elegans in a microfluidic environment. *Lab on a Chip*, **10**(2), 220–226.

Rezai, P., Salam, S., Selvaganapathy, P. R., and Gupta, B. P. (2012). Electrical sorting of caenorhabditis elegans. *Lab on a chip*, **12**(10), 1831–1840.

Salam, S., Ansari, A., Amon, S., Rezai, P., Selvaganapathy, P. R., Mishra, R. K., and Gupta, B. P. (2013). A microfluidic phenotype analysis system reveals function of sensory and dopaminergic neuron signaling in c. elegans electrotactic swimming behavior. In *Worm*, volume 2, page e24558. Taylor & Francis.

Savitzky, A. and Golay, M. J. (1964). Smoothing and differentiation of data by simplified least squares procedures. *Analytical chemistry*, **36**(8), 1627–1639.

Serra, J. (1983). *Image analysis and mathematical morphology*. Academic Press, Inc.

Stiernagle, T. (2006). Maintenance of C. elegans. *WormBook : the online review of C. elegans biology*, pages 1–11.

Sukul, N. C. and Croll, N. A. (1978). Influence of potential difference and current on the electrotaxis of caenorhaditis elegans. *Journal of nematology*, **10**(4), 314.

Swierczek, N. a., Giles, A. C., Rankin, C. H., and Kerr, R. a. (2011). High-throughput behavioral analysis in C. elegans. *Nature methods*, **8**(7), 592–8.

Tong, J. (2014). *Chemical and genetic screening applications of a microfluidic electrotaxis assay using nematode Caenorhabditis elegans*. Ph.D. thesis.

Tong, J., Rezai, P., Salam, S., Selvaganapathy, P. R., and Gupta, B. P. (2013). Microfluidic-based electrotaxis for on-demand quantitative analysis of caenorhabditis elegans' locomotion. *Journal of visualized experiments: JoVE*, (75).

Wang, S. J. and Wang, Z.-W. (2013). Track-A-Worm, An Open-Source System for Quantitative Assessment of C. elegans Locomotory and Bending Behavior. *PloS one*, **8**(7), e69653.

Zivkovic, Z. (2004). Improved adaptive Gaussian mixture model for background subtraction. *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, (2), 28–31 Vol.2.