Tracking *C. elegans*

Valerie Simonis

September 4, 2014

Abstract

This paper presents and discusses the challenges and primary considerations in developing a single-worm motorized tracking system in the aim of recording and quantifying *C. elegans* locomotory behavior given certain experimental manipulations.

1 Introduction

The nematode *Caenorhabditis elegans* is a model organism for neural studies due to its simple structure (959 somatic cells) and known connectome (302 neurons and 8000 connections). Traditionally, this nematode is studied to model its behavioral output, observed through motion, given different experimental manipulationsn. Given that a nematode's behavior is only modulated by internal states (genetics) and external stimuli (such as presence or absence of food or touch-stimulus), manipulating these states and stimuli can lead to discovery about neural circuits' purposes and activity. For example, mutations can be applied to a worm to simulate human diseases like Alzeihmer's, Autism Spectrum Disorders and other neural-disruption diseases. The impact of such genetic manipulations can then be measured to infer linkages between genes, neurons and behavior. *C. elegans* related research often aims to discover which neurons control behavior with the expectation that findings can be generalized to aid in the development of a human brain activity map to better understand diseases caused by neural disruption.

A common practice in the *C. elegans* Neuroscience research field is to quantify a worm's locomotory behavior using Computer Vision techniques. This interdisciplinary approach aims to provide more precise measurement and description tools of *C. elegans'* locomotory behavior to make inferences in the field of Neuroscience.

Specifically, our long term goal is to identify neurons affecting the food related behavior of *C. elegans*. In order to discover this, we propose to quantify a wild-type worm's off-food behavior and find similarly behaving mutant worms when placed in an on-food environment. For our purposes, worm food is an *E. coli* bacterial lawn spread on an agar plate assay. Knowing which worm(s) from a list of possible candidate mutants behave on-food as the wild-type (naturally occurring) off-food, will suggest which neurons are part of the food-related behavior neural circuit. Knowing how similarly these worms behave will infer the strength of certain neural paths over others under the food-presence condition. In order to observe and quantify the intricacies of *C. elegans'* food related behavior, we aim to build and program a system to record, measure, quantify and model *C. elegans* behavior through video and image processing.

2 Background

Due to the difficulty of humans to observe and record worm activity with precision over either long or short time scales, it has become almost standard to record freely moving *C. elegans* either one at a time or in a group. These videos are then processed after the fact, to segment the worm and provide intermediate representations of

the worm's body (binary image, skeleton and sampled skeleton points). Features are then extracted from the intermediate representations and further mined for patterns to quantify phenotypes as in [1] or to support hypotheses linking genetic manipulation to observed locomotory behavior.

Some recording implementations ([2, 3]) focus on one area of the dish, negating the need to recenter the worm(s) under study. Quantitative methods developed for this type of system are used to track the centroid of multiple worms only in that area of the assay and do not require a motorized tracker. Other implementations ([4]) used either a motorized stage (usually when using a microscope providing this functionality) or a motorized X-Y translation stage upon which a camera is affixed. Moving the camera over the stage (containing the assay and thus animal under study) is preferred to minimize the possible effects of the mechanical motion on the nematode's locomotory behavior.

We aim to record one worm at a time and are interested in comparing its locomotory behavior off-food to candidate mutant worms on food. It has been suggested in [5] that wild-type *C. elegans'* off-food behavior is remarkably different from an on-food environment. Worms under this condition can travel longer distances in more frenetic behavioral patterns (ie: fast with many turns), requiring a larger dish to give the nematode space to search for food without colliding with the edge of the dish. Other research groups use an *E. coli* bacterial lawn (food) to restrict the area in which the animal will visit, and to minimize its maximal speed as explained in [6]. This highlights the importance of developing a system for which a minimal amount of parameters need to be adjusted for appropriate motorized tracking for any type of worm under any experimental condition. Examples of parameters influencing motorized tracking include worm-finding sub-sampling rate of the video stream (ie: how often to get a worm location) and the decision boundary determining when the camera on an X-Y translation stage is instructed to move. For this system to be useful, the motorized tracking parameters need to be generalizable to any mutant- or wild-type of worm regardless of speed and activity.

2.1 Acquiring Video for Analysis

A critical first step in analyzing and quantifying worm behaviors involves acquiring video. In order to capture subtle differences in individual animal behavior, it is of high importance to capture worm movement at appropriate time and imaging resolution under imaging conditions favoring accurate extraction of intermediate representations of the worm's per-frame posture.

A high and accurate frame rate will provide time scales suitable for motion analysis. It will also provide more image data from which to extract features, allowing data scientists to throw out video frames for which segmentation and skeletionizing is ineffective, in which the image has become corrupt, or the nematode has been blurred by the motion of the camera. Traditionally challenging postures to segment or skeletonize are looped or coiled [7]. The practice of throwing out these frames seems flawed, as often the frames being thrown out are those of high importance for locomotory analysis.

In order to provide an advantage to the segmentation process, optimizing the image quality is important. This can be accomplished by providing even and uniform illumination at time of capture, facilitating the process of finding intensity thresholds that best separate a worm from its agar background. In order to minimize unwanted image artifacts effecting the selection of only worm pixels during segmentation, it is recommended to ensure proper care in the pouring of the agar (substrate upon which the worm is crawling), and imaging the plate with few fingerprints on the dish (trivial, but important!) Also, the dish should be imaged with the agar portion of the dish facing up, allowing possible condensation observed over long recording periods to travel downwards, minimizing its impact on the image quality.

Currently, most trackers available are written in MATLAB or C++ and require the Image Analysis Toolbox in order to acquire frames from a camera. Many require expensive hardware, making *C. elegans* motorized tracking an expensive and exclusive endeavour. For a complete overview of currently available trackers, please refer to [8].

2.2 Quantifying Worm Behavior through Video Analysis

Though slightly outside the scope of our current investigation, it is important to note some of the standard methods for extracting features. Given our position to control the quality of the images to be provided to the feature extraction and analysis routines, some design decisions at our current stage may impact the effectiveness of further study.

A popular approach in the field is to generate decompositional features from the videos acquired [4]. In general, four "intermediate" representations are calculated using the image processing sequence from Figure

Figure 1: Standard image processing sequence for intermediate representations

The original gray scale image is only used to provide information about the animal's *translucency* (for head/tail identication, mainly). From the gray scale image, a binary image is generated by applying a threshold segmentation. From this image *size* and *shape* features are calculation (as **morphology features**). From the binary image, the centroid is found as a representation of the worm's *position* extrapolated to **trajectory features** across multiple frames (such as *direction, displacement, velocity and acceleration*). From the segmentation, a medial axis is determined using skeletonization techniques. This is a complex problem when applied to looped or curled postures in which occlusion might occur [7]. This skeleton is then downsampled to a number of representative points (anywhere between 13 [9] and 49 [1]) ordered from head to tail. The head and tail distinction can be automatically attempted using the presence of intensity, curvature or width profile differences between a nematode's head and tail, though this task is not inherently trivial.

Features extracted from these intermediate representations have been widely used by the Schafer research lab for phenotype analysis as inputs to: phenotype classification using decision trees [10] or random forests [6] or phenotypic clustering [11]. Also, these features can be used to annotate video frames with some commonly described behaviors such as crawling and turns (reversals, shallow turns, omega loops, and gamma loops.)

Other approaches to extracting features involve tracking only the worm's centroid frame-to-frame to get information about the worm's complete path (or trail). This path's curvature can be modeled as piece-wise harmonic functions [12] or using differential tangent angles and arc lengths [13].

Furthermore, [14] discovered that any worm *posture* can be represented using four basic shapes (or eigenworms). Both eigenworm and trail analyses model posture over time. An advantage to using eigenworms over trail analysis is these basic shapes are learned from all frames (so each frame contributes), but any individual frame can be modelled using this approach (frame-centric approach). Trail analysis models postures across multiple frames without frame-to-frame distinction. In [15], the eigenworm model is coupled with center-of-mass trajectory analysis to find differenences in movement strategy.

Using unsupervised learning techniques, [16] uses eigenworm representations of the worm **postures** developed in [14] to identify and discover behavioral motifs (or patterns) to use in clustering phenotypes. We aim to build on this approach, focusing on a smaller, more focused subset of candidate mutants of interest to Dr. Kim, instead of a wide variety of mutants as profiled in [1] in order to find model posture-similarity between two individual worms. Furthermore, our ability to capture longer sequences may result in more significant patterns through more data.

In terms of observing worms off-food, it has been attempted by [13] for 30 min using a camera mounted on a stereomicroscope at an unpublished frame rate, and [17] for 70 min for 3 mutant types in a 15cm assay at 3 fps. [13] concludes that worms off-food use shallow turns most frequently as a method of orientation (over reversals and omega turns) by analyzing features extracted from the track of a worm represented as a spline. [V: plan on expanding this section as I look more closely]

Given our goal of determining similarity between two worms based on their locomotory behavior, once we complete the image acquisition software, we will investigate the utility of many of the features described, including: **morphology**, **posture**, and **trajectory** features.

3 Methodology

The following section outlines the various components of tracker and how they are connected and controlled programmatically.

3.1 Tracker Overview

The components of the tracker are as follows:

- *base*
  - 2 stepper motors
  - EiBotBoard (motor controller)
  - X-Y translation mechanism
  - camera
- *housing*
  - elevation rig
  - dish stage
  - lighting mount
  - lighting diffuser
- *computer*

The *base* and *housing* components (see Figure

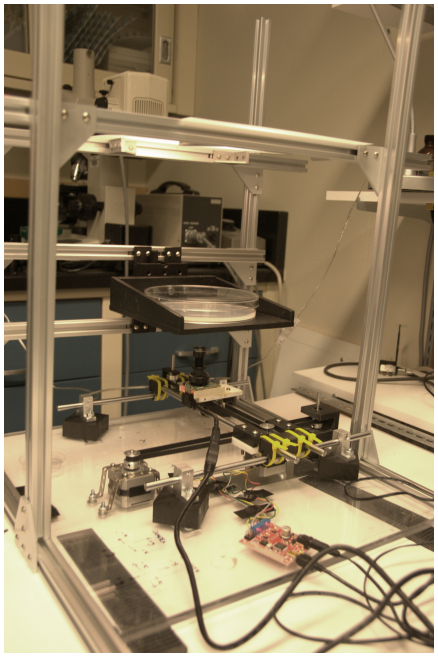

Figure 2: *housing* and *base*

The *base* (see Figure
It is important to note that the *dish stage* (see Figure

3.2 Hardware

This section specifies hardware parts used, pricing, and references to purchasing websites.

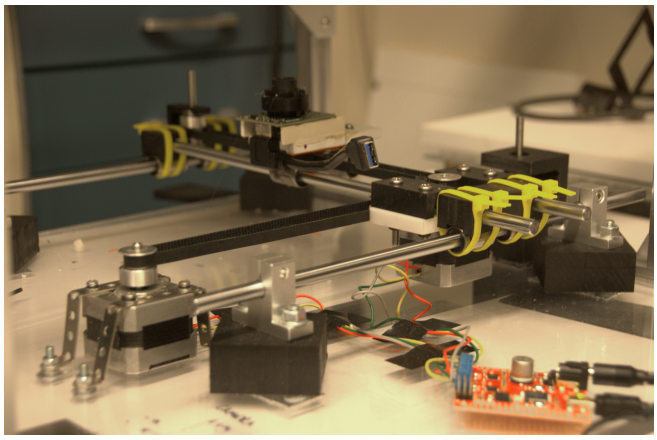

Figure 3: Close-up of *base*

3.2.1 Motors and Motor Control Board

The stepper motors being used are easily available from the Evil Mad Scientist website [18]. The price point is $ 16/motor.

The motor control board is the `EggBotBoard` [19]. The price point is $50.

3.2.2 Imaging

The current camera being used is e-con System's `See3CAM_10CUG`. [20] It is a 1.5MP USB 3.0 camera capable of up to 45 fps at a resolution of 1280 $\times$ 960. It is a monochrome camera capable of delivering directly in FOURCC code GREY format. It can also deliver images in YUYV format. It is important to note that other cameras may be compatible with the system.

3.2.3 Computer

We are using a computer dedicated to the tracking system, though any laptop with similar configuration (outlined in Appendix

3.3 Software

The tracking sofware was developed using the OpenCV 3.0.0 (dev) library [22] in Python. The implementation leverages the fact OpenCV's Python library is bound to methods written in C++ and handles memory allocation, and interfaces with the `video4linux (V4L & V4L2)` camera drivers (for devices called "UVC compatible").

This section specifies the algorithms and methods used specifically for finding the location of the worm in an image and deciding when to re-center the worm in the field of view. Also, it provides specification on how the images are captured from the camera and how the worm location is dispatched to the motor control board to move the camera.

3.3.1 Finding the Worm

The algorithm implemented for finding the worm is based on a simple motion tracking mechanism: the difference image. It leverages the fact that immobile background artifacts (like tracks or dips) subtract out to black, leaving only the moving worm as a white object in the difference image. The maximum pixel intensity in the difference image appears to always locate the worm, though no guarantee is placed that this point is the same on the worm's body frame to frame, or that it coincides with the centroid (center of mass).

```
Data: current frame, reference frame
Result: wormLocation (row,col)
while frame from camera do
    if have reference frame and camera not just moved then
        subtracted image = reference frame - current frame;
        if confidence in cropping then
            crop subtracted image;
        end
        blur subtracted image;
        get maximum pixel intensity in blurred subtracted image;
        update wormLocation;
        set justMoved to False;
    else
        set current image as reference image;
    end
end
```

Algorithm 1: findWorm

The `findWorm()` method is only called at a rate of 10 fps, subsampling from the stream being emitted by the camera. This ensures more guarantees that the worm will not subtract itself out. To further improve the localization of the worm, a Gaussian smoothing filter ($\sigma = 9, n = 45 \times 45$) is also applied.

For efficiency when applying the Gaussian smoothing function, the space in the image in which the worm is sought is restricted by cropping (`calculateCrop()`) during times when the confidence is high that the worm still resides in the `cropRegion` of the frame being processed. The `cropRegion` is always centered on the previous sample's `wormLocation`. The cropping mechanism is in action after variable `nPauseFrames` (current 20 frames) have elapsed, following the program start-up and after the camera has moved.

Necessarily, a new reference image is acquired after the camera has moved and is done in the `decideMove()` method (specified in

In the code, this method is called the `processFrame()` method as part of the `Finder` class in the `finder.py` module.

3.3.2 Deciding When to Re-center the Worm in the Field of View

The algorithm for the `move()` decision engine, was built trying to minimize heuristics for generalization to worms of any speed or activity.

```
Data: wormLocation (row, col)
Result: Move or not move motors
if not on break and has reference image then
    if reference image is old then
        get new reference image;
    end
    if wormLocation is outside boundary then
        if wormLocation is not ridiculous then
            set justMoved to True;
            clear reference image;
            start break;
            center the worm;
        end
    end
end
```

Algorithm 2: decideMove

The current length of time during which the motors will be on break after a move has just been completed, is set to 3 sec (by `breakDuration`). This ensures that the motors won't continually move. It corrects for possible localization errors after the camera has moved and a new reference image is being used. This is a very important check. It's possible that the `justMoved` flag is redundant.

A frame reference is considered old if is has been used for more than `REFPING` seconds (currently set to 600,000, and hence never called). This method was included for worms who are almost immobile or remain in a certain area. When the reference frame is old, subtracting the current and reference frames removes the worm, which then receives no location. Its utility has not been tested since the Gaussian blur was added to the `findWorm` implementation.

A worm location is considered to be outside the boundary if it is outside the rectangle formed by `boundRows`, `boundCols` centered about the center of the frame.

A worm location is considered ridiculous if it is above a threshold set in `MAXONEFRAME` (currently set to 500 pixels). Anecdotally, this filtering mechanism has not evaluated to True since a Gaussian blur was added to the `findWorm()` method, and may not be necessary.

It is important to note that the parameters `boundCols, boundRows`, `MAXONEFRAME` and `REFPING` are set in the `finderArgs` dictionary in the `Tracker` class defined in the `tracking.py` module. This was mostly done to set many of the parameters that were

changed often during testing to one place. These parameters are passed as arguments to the `Finder` object when initialized from `Tracker` object.

The underlying geometry of these constructs in illustrated in Figure

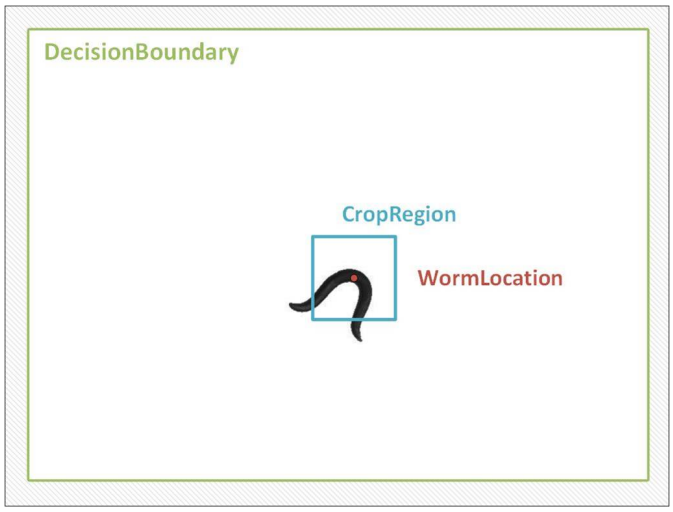

Figure 4: Geometry of the `findWorm` and `decideMove()` methods

The schematic, detailed overview of the system can be seen in Figure

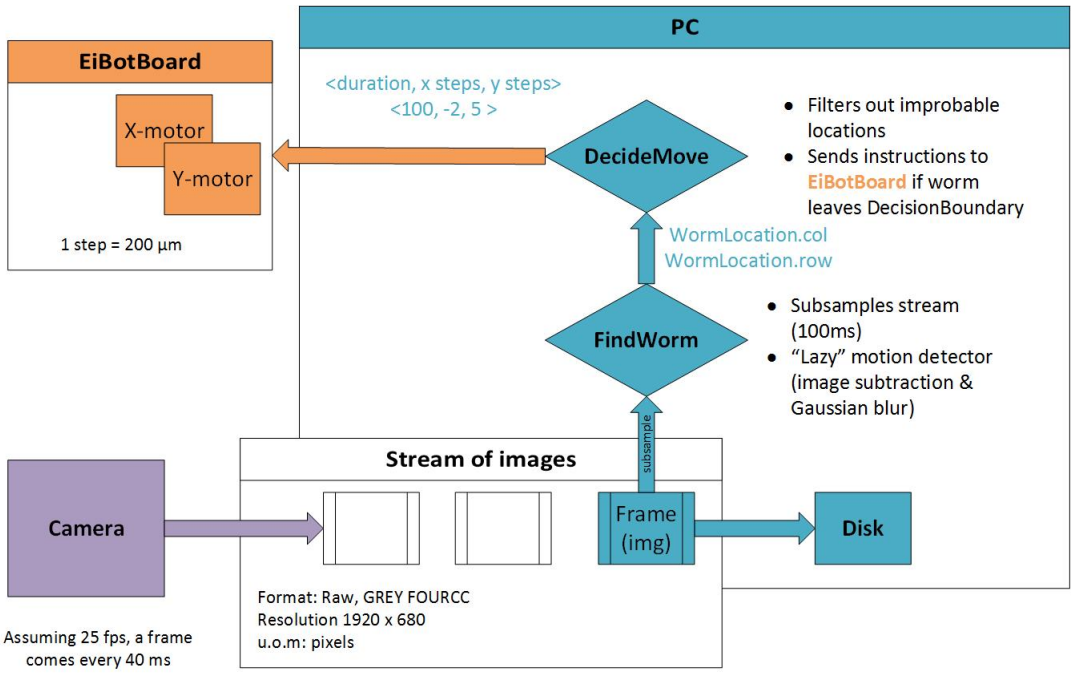

Figure 5: System diagram

3.3.3 Linking Hardware to Software

Motors One of the main motivations to use python to write this software is that there exists sample code for sending instructions to the EBB. [23, 24]. The module `easyEBB.py` defines the `easyEBB` class and all methods related to motor control. A brief selection is outlined below:

- `__init__(resolution, sizeMM, stepMode)` instantiates an `easyEBB` object with the following parameters (and associated defaults):
  - `resolution = (1280,960)`
  - `sizeMM = 10` (the default frame width in mm)
  - `stepMode = 5` (full step mode)
- `centerWorm( self, duration, colWormPix, rowWormPix )`: besides constructor (and destructor), is the only method called from the client code. Given a worm at pixel location `(rowWormPix ,colWormPix)`, it will convert the pixel instructions (to center worm) to motor step instructions (`move` method). It is important to note that the motors will only move in integer step instructions, meaning the camera may not move exactly to the desired location.

  [Note: it has been requested by collaborator Dr. Christopher Brandon to send the move commands separately for X and Y directions to limit vibration]
- `pixUmStepConversions( self, stepMode )`: uses resolution and frame width (in mm) to convert pixel measurements to steps.
- `move( self, duration, xstep, ystep )`: sends move command to motor control board
- Driver methods:
  - `openSerial()`: scans the `\dev` folder for EBB and opens serial connection
  - `closeSerial()`: closes the serial connection opened above
  - `enableMotors()`: when serial connection is open, enables motors to receive `move` commands
  - `disableMotors()`: when serial connection is open, disables motors from receiving `move` commands

The motors can operate at full step (200 μm/step) or in fractional steps (e.g.: $1/2, 1/4, ...$). It is preferred to use full steps due to its precision over the fractional step modes. It is possible to instantiate the EBB object `easyEBB` in fractional mode in the constructor by assigning the keys from the `stepOpts` dictionary to the `stepMode` variable:

```
self.stepOpts = {1: 1./16,
                 2: 1./8,
                 3: 1./4,
                 4: 1./2,
                 5: 1}
```

Acquiring images from the camera Despite this camera being touted as UVC compatible, many of the traditional UVC driver queries are not behaving nicely with

the OpenCV driver interface. This includes querying frame rate.

In order to get our camera to read in images with the OpenCV library, changes need to be made in the OpenCV library source file: `opencv\modules\highgui\bin\cap_libv4l.cpp`. The lines to change and how to change them are outlined in Table

Generally, however, the OpenCV implementation of capturing images from a camera is illustrated below.

```
#instantiation
cap = CaptureManager(
        cv2.VideoCapture(captureSource),
        window,
        mirroredPreview,
        resolution)
#use
while window.isWindowCreated:
    cap.grab()
    frame = cap.retrieve()
    window.show(frame)
```

Writing images to video The video is written leveraging OpenCV's `VideoWriter` object. When writing frames, the FOURCC MJPG codec writes in color format (which is a little unfortunate) using a frame rate estimate calculated after each frame is 'exited'. This estimate, however, changes over time. Troubling, is the fact that the VideoWriter gets instantiated only once with a frame rate estimate at that time, which unfortunately is variable over the length of capture. A small example of instantiation of the `VideoWriter` object and the line used to write a frame is shown below.

```
#instantiation
videoWriter = cv2.VideoWriter( videoFilename,
                               videoEncoding,
                               fps,
                               size,
                               color = False )
#use
videoWriter.write(frame)
```

Calibration Depending on the distance between the camera and the petri dish, the field of view will change. The calibration procedure involves initializing the `easyEBB` object with the width of the frame in mm. There is a command line flag specified during launch of the tracker (`-m conf`) that will draw two dots on the image coming from the camera. One is at pixel location (200, 300) and the other is in the center

of the frame. To verify the measurement, a microscopic target should be positioned at the (200, 300) dot and the script `dotTest.py` is run. If the target lands in at the center dot, then calibration is complete. If not, a more accurate measurement of the width of the frame should be attempted.

3.3.4 Main Thread

The main thread connecting the video capture, processing, and recording methods is declared the `tracker.py` file as `run()` and was inspired from a project described in [25]. It controls and redirects the program flow depending on parameters such as the presence of motors or the color format of the input image stream (to toggle video and camera sources). To start recording, the TAB key is pressed once and then again to stop recording. The ESC key exits the program entirely (with safe guards to end recording and close the serial connection to the motors). A minimalist GUI is provided by OpenCV which displays images using the `show()` method. This allows display of an overlayed image in which `wormLocation` can be drawn and verified by a human watching the video.

4 Results and Discussion

This section reports some of the more challenging aspects of this project as well as important design and implementation considerations.

4.1 Real-time Processing Requirement

Since the worm under study is only 1 mm long, it's impossible to watch its locomotory behavior without the intermediary of a microscope or zoom-lens mounted camera. This forces a requirement that the recording should occur with a known frame rate in order to ensure fidelity of time scales. This will ensure that a given behavior is represented by the recording at the same speed at which it is occuring. A known, consistent frame rate will be important to quantify motion features dependent on time such as velocity and acceleration. For the tracking portion of this video acquisition system, the `wormFinder()` has to work fast enough to recenter the worm before it leaves the field of view.

Frame rate Given that this system needs to work real-time, the main development concern has been to acquire, process and record images from the camera in real-time at the frame rate delivered by the camera. However, 45 fps may supply many sequential frames in which the worm has barely moved a pixel. A consideration for whether or not to lower the frame rate lies in the length of videos we plan on acquiring. For our given problem, we will be recording video for a longer timescale than is traditionally done, between 45 min and 1h30 min. An important consideration for our system (both for acquisition and feature extraction) will be its robustness across a large data set. In the event we down-sample our recording (due to space or inability to capture 45 fps), we will need to weigh the computational and space benefit over the potential loss of image data (that will not be able to be later reconstructed).

Guidance in determining an appropriate frame rate is supplied by Table

Speed	Frame Rate (fps)								
	5	10	15	20	25	30	35	40	45
500 μm/sec	12.80	6.40	4.27	3.20	2.56	2.13	1.83	1.60	1.42
200 μm/sec	5.12	2.56	1.71	1.28	1.02	0.85	0.73	0.64	0.57

Table 1: Possible distance travelled between frames (in pixels) for different frame rates and maximal worm speeds (assuming 1280 by 960 resolution and 10mm f.o.v.

Our camera's maximal frame rate of 45 fps may be more appropriate for a published [4] maximal worm speed of 500 μ/sec. A worm traveling at that speed would have moved 1.42 pixels between frames, which may be an important motion to capture and record.

Image processing efficiency Early development of the system targeted using a Raspberry Pi as the controlling computer for both image processing and camera capture and recording. The memory available on the Raspberry Pi and the Python classes available to interface with the camera were very limiting in terms of processing. The experience, however, did inspire using computationally simple methods for finding the worm, over more common methods (and initially more robust methods) of finding the centroid of the worm from a binarized image.

The most expensive step in the image processing sequence is applying a Gaussian filter to the difference image. When initially developing the system, I was using a matrix data structure from the `numpy` Python package to hold the image. A Gaussian filter applied to a `numpy` image took much longer to process than using OpenCV's `CvMat` object. This would be attributed to OpenCV's efficient C++ implementation. The Gaussian step is vital in ensuring robustness to worm localization.

Another main component of processing images to keep up with real-time capture is to subsample the image stream. A bonus consequence is that this technique allows the worm to move a little bit before the next image is processed, limiting situations in which the image difference image subtracts a current worm image from its reference.

Recent tests on previously captured video have shown that when the search space is the entire image, worm finding can occur at 33 fps. This is improved when the search space is cropped to a minimum observed at 200 fps.

Limitations In the current version of the tracking software, the estimated frame rate is about 22 fps, well slower than the 45 fps promised by the camera manufacturer. This could be due to structure of the program and its inefficient handling of passing around states that need to be shared by different procedures. I am investigating using either a leaner Shared State design pattern or Observer design pattern to circumvent this challenge.

Another influence on frame rate is the interpreted nature of the Python language.

Investigation produced evidence that the EBB `centerWorm()` procedure required a second to return after being called in `decideMove()`. During this time, the program waits for the `centerWorm()` method to return, delaying the entire capture process. To circumvent this problem, the Callback design pattern was implememted, sending `centerWorm()` to its own process without waiting for it to return. This allows for a more timely execution of the subsequent methods.

Also, in the event that frame rate is never metronomic, a log file specifying time stamps of each written frame as well as motor instructions is generated, allowing the Feature Extractor to have true time and distance information to accurately generate time and distance features such as velocity and acceleration. These log files are generated using Python's logging library which is well worth learning. The timestamp for writing images is important given the non-metronomic nature of the image processing procedure. As fast as the image processing is and can be, the frame rate estimate changes over time given the different time it takes to find a worm in the whole image vs. a cropped search space.

A more optimal solution to securing the actual frame rate supplied by the camera would incorporate total separation of the capture/recording and finder threads, combined into a main thread. Furthermore, design patterns under consideration will ease the process adding a GUI with controls (instead of just image display) would allow for ease in testing of some of the decision parameters discussed in

4.2 Hardware Integration

Testing the tracker is made difficult by the usual absence of one or more hardware elements (motors and camera) at time of testing (the complete tracking system lives at RFUMS.) Algorithms are often tested on previously recorded video. Certain conditional boolean controls such as color format of the input or the presence or absence of motors were necessarily built into the programmatic structure to allow for testing on multiple inputs (video or live camera).

4.3 Motor Considerations

An important design decision is to have a very large field of view compared to other experiments. We posit that this will result in fewer (albeit longer) motor movements. This is an important point to consider, given that frames in which which the camera is moving are often dropped for analysis as the camera motion results in a blurred worm. It might still be more beneficial to the analysis step to have more frequent, shorter movements.

4.4 Availability

It is important to note that the software has been developed and tested on a Unix system (Xubuntu 14.04) and may not be compatible to a Windows environment. In general, this may be a deterrent to researchers, as few are comfortable in that environment. One of our major selling points is going to be price point relative to the current systems available from domain juggernauts. However, our program will be free and

open source while most others are open source, but dependent upon having MATLAB toolboxes that are costly for the image acquisition portion.

I would love to have this system be compatible with any PC so that any available existing computer in a laboratory could be used. The limiting factor, is going to be the camera. It may not be compatible in a Windows environment (though this hypothesis has never been specifically tested).

5 Conclusions

There may be quite a few programmatic changes to make to ensure good video (data) acquisition given many of the considerations discussed. I am hopeful that once this step is complete, more techniques of feature extraction independent of segmentation and skeletonization can be explored such as optical flow or using local invariant features such as SURF, SIFT and MSER which are more generalizable to other Computer Vision problems.

A Installation

The program code is available at `http://github.com/vsimonis/worm3/`

Successful development and running of the system can be achieved by running the following scripts.

A.1 Setup Machine: `setup-machine.sh`

Installs all of the authors favorited development tools.

```
#!/bin/bash

sudo apt-get -y update
sudo apt-get -y upgrade

#Support programs
echo "Getting favorite programs"
sudo apt-get install -y emacs24
sudo apt-get install -y texlive
sudo apt-get install -y guake
sudo apt-get install -y synaptic

echo "Browsers and multimedia"
sudo apt-get install -y chromium-browser
sudo apt-get install -y flashplugin-nonfree
sudo apt-get install -y libmono-wcf3.0a-cil
sudo apt-get install -y vlc

echo "Camera stuff"
sudo apt-get install -y v4l-utils
sudo apt-get install -y v4l-conf
sudo apt-get install -y guvcview
sudo apt-get install -y cheese


# Setup Git & project
sudo apt-get install -y git
echo "Setting up Git"
git config --global user.name "Valerie Simonis"
git config --global user.email valerie.simonis@gmail.com
git config --global color.ui true

mkdir w3
cd w3
git init
git remote add w3 https://github.com/vsimonis/worm3.git
```

```
git pull w3 master
echo "Git ready to use"

# Python libraries
sudo apt-get install -y python-skimage
sudo apt-get install -y python-sklearn
sudo apt-get install -y python-pandas

chmod +x setup-machine.sh
```

A.2 OpenCV Installation Script: `opencv.sh`

This has been adapted from multiple internet sources to allow a uniform installation of this program across development machines. It is also forwards compatible to the most recent OpenCV library. Current library in use is version 2.4.9
These files are availabe in the git repository under the `install` directory.

```
version="$(wget -q -O - http://sourceforge.net/projects/
   opencvlibrary/files/opencv-unix
| egrep -m1 -o '\"[0-9](\.[0-9])+' | cut -c2-)"

echo "Installing OpenCV" $version
cd
mkdir OpenCV
cd OpenCV
echo "Removing any pre-installed ffmpeg and x264"
sudo apt-get -qq remove ffmpeg x264 libx264-dev
echo "Installing Dependenices"
sudo apt-get -qq install libopencv-dev
sudo apt-get -qq install build-essential checkinstall cmake
    pkg-config yasm
sudo apt-get -qq install libtiff4-dev libjpeg-dev libjasper
   -dev
sudo apt-get -qq install libavcodec-dev libavformat-dev
   libswscale-dev libdc1394-22-dev libxine-dev
   libgstreamer0.10-dev libgstreamer-plugins-base0.10-dev
   libv4l-dev
sudo apt-get -qq install python-dev python-numpy
sudo apt-get -qq install libtbb-dev libqt4-dev libgtk2.0-
   dev
sudo apt-get -qq install libfaac-dev libmp3lame-dev
   libopencore-amrnb-dev libopencore-amrwb-dev libtheora-
   dev libvorbis-dev libxvidcore-dev
sudo apt-get -qq install x264 v4l-utils ffmpeg
sudo apt-get -qq install libgtk2.0-dev
```

```
echo "Downloading OpenCV" $version
wget -O OpenCV-$version.zip http://sourceforge.net/projects
   /opencvlibrary/files/opencv-unix/$version/opencv-"
   $version".zip/download

echo "Installing OpenCV" $version
unzip OpenCV-$version.zip
cd opencv-$version
rm -r build
mkdir build
cd build
cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/
   usr/local -D WITH_TBB=ON -D BUILD_NEW_PYTHON_SUPPORT=ON
   -D WITH_V4L=ON -D INSTALL_C_EXAMPLES=ON -D
   INSTALL_PYTHON_EXAMPLES=ON -D BUILD_EXAMPLES=ON -D
   WITH_QT=ON -D WITH_OPENGL=ON ..
make -j4
sudo make install
sudo sh -c 'echo "/usr/local/lib" > /etc/ld.so.conf.d/
   opencv.conf'
sudo ldconfig
echo "OpenCV" $version "ready to be used"
```

A.3 Changes to OpenCV source

724	`capture->form.fmt.pix.pixelformat = V4L2_PIX_FMT_BGR24;`
724	`capture->form.fmt.pix.pixelformat = V4L2_PIX_FMT_GREY;`
734	`if (V4L2_PIX_FMT_BGR24 != capture->form.fmt.pix.pixelformat) {`
734	`if (V4L2_PIX_FMT_GREY != capture->form.fmt.pix.pixelformat) {`
845	`IPL_DEPTH_8U, 3, IPL_ORIGIN_TL, 4 );`
845	`IPL_DEPTH_8U, 1, IPL_ORIGIN_TL, 4 );`
931	`capture->imageProperties.palette = VIDEO_PALETTE_RGB24;`
931	`capture->imageProperties.palette = VIDEO_PALETTE_GREY;`
943	`if (capture->imageProperties.palette != VIDEO_PALETTE_RGB24) {`
943	`if (capture->imageProperties.palette != VIDEO_PALETTE_GREY) {`
982	`IPL_DEPTH_8U, 3, IPL_ORIGIN_TL, 4 );`
982	`IPL_DEPTH_8U, 1, IPL_ORIGIN_TL, 4 );`
1258	`IPL_DEPTH_8U, 3, IPL_ORIGIN_TL, 4 );`
1258	`IPL_DEPTH_8U, 1, IPL_ORIGIN_TL, 4 );`
1271	`IPL_DEPTH_8U, 3, IPL_ORIGIN_TL, 4 );`
1271	`IPL_DEPTH_8U, 1, IPL_ORIGIN_TL, 4 );`

Table 2: Changes to `cap_libv4l.cpp`

B Code

see `http://www.github.com/vsimonis/worm3`

References

[1] Eviatar Yemini et al. "A database of Caenorhabditis elegans behavioral phenotypes". In: *Nat Meth* 10.9 (Sept. 2013), pp. 877–879.

[2] Daniel Ramot et al. "The Parallel Worm Tracker: a platform for measuring average speed and drug-induced paralysis in nematodes". In: *PLoS One* 3.5 (2008), e2208.

[3] George D Tsibidis and Nektarios Tavernarakis. "Nemo: a computational tool for analyzing nematode locomotion". In: *BMC neuroscience* 8.1 (2007), p. 86.

[4] Zhaoyang Feng et al. "An imaging system for standardized quantitative analysis of C. elegans behavior". In: *BMC bioinformatics* 5.1 (2004), p. 115.

[5] Jesse M Gray, Joseph J Hill, and Cornelia I Bargmann. "A circuit for navigation in Caenorhabditis elegans". In: *Proceedings of the National Academy of Sciences of the United States of America* 102.9 (2005), pp. 3184–3191.

[6] Wei Geng et al. "Automatic tracking, feature extraction and classification of C. elegans phenotypes". In: *Biomedical Engineering, IEEE Transactions on* 51.10 (2004), pp. 1811–1820.

[7] Kuang-Man Huang, Pamela Cosman, and William R Schafer. "Machine vision based detection of omega bends and reversals in C. elegans". In: *Journal of neuroscience methods* 158.2 (2006), pp. 323–336.

[8] Steven J Husson et al. "Keeping track of worm trackers". In: (2005).

[9] Christopher J Cronin et al. "An automated system for measuring parameters of nematode sinusoidal movement". In: *BMC genetics* 6.1 (2005), p. 5.

[10] Joong-Hwan Baek et al. "Using machine vision to analyze and classify Caenorhabditis elegans behavioral phenotypes quantitatively". In: *Journal of neuroscience methods* 118.1 (2002), pp. 9–21.

[11] Wei Geng et al. "Quantitative classification and natural clustering of Caenorhabditis elegans behavioral phenotypes". In: *Genetics* 165.3 (2003), pp. 1117–1126.

[12] Venkat Padmanabhan et al. "Locomotion of C. elegans: a piecewise-harmonic curvature representation of nematode behavior". In: *PloS one* 7.7 (2012), e40121.

[13] Daeyeon Kim et al. "The shallow turn of a worm". In: *The Journal of experimental biology* 214.9 (2011), pp. 1554–1559.

[14] Greg J Stephens et al. "Dimensionality and dynamics in the behavior of C. elegans". In: *PLoS computational biology* 4.4 (2008), e1000028.

[15] Greg J Stephens et al. "From modes to movement in the behavior of Caenorhabditis elegans". In: *PloS one* 5.11 (2010), e13914.

[16] André EX Brown et al. "A dictionary of behavioral motifs reveals clusters of genes affecting Caenorhabditis elegans locomotion". In: *Proceedings of the National Academy of Sciences* 110.2 (2013), pp. 791–796.

[17] Katsunori Hoshi and Ryuzo Shingai. "Computer-driven automatic identification of locomotion states in Caenorhabditis elegans". In: *Journal of neuroscience methods* 157.2 (2006), pp. 355–363.

[18] Evil Mad Scientist. *Parts Menu: Stepper Motor*. URL: `http://shop.evilmadscientist.com/productsmenu/partsmenu/187-stepper` (visited on 06/08/2014).

[19] Evil Mad Scientist. *Parts Menu: EBB Driver Board*. URL: `http://shop.evilmadscientist.com/productsmenu/partsmenu/188-ebb` (visited on 06/08/2014).

[20] e-con Systems. *See3CAM Camera 1.3 Mega Pixel USB3.0 Camera Developer Resources*. URL: `http://www.e-consystems.com/doc_Globalshutter-USB3-camera.asp` (visited on 06/08/2014).

[21] Intel Corporation. *Mini PC - Intel NUC Kit DN2820FYKH*. URL: `http://www.intel.com/content/www/us/en/nuc/nuc-board-dn2820fykh.html` (visited on 06/08/2014).

[22] itseez. *The OpenCV Reference Manual, Release 2.4.9.0*. 2014. URL: `http://docs.opencv.org/opencv2refman.pdf` (visited on 06/08/2014).

[23] Brian Schmalz. *EBB (EiBotBoard)*. URL: `http://www.schmalzhaus.com/EBB/` (visited on 06/08/2014).

[24] Windell. *eggbotcode*. URL: `https://code.google.com/p/eggbotcode/downloads/detail?name=eggbot_2013_08_11_2.3.4.zip&can=2&q=` (visited on 06/08/2014).

[25] Joseph Howse. *OpenCV Computer Vision with Python*. Packt Publishing, 2013.