1 Methodology

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

1.1 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 1) are modular, meaning any housing can be used with any base. This also allows for easy transport.

The *base* (see Figure 2) module houses the hardware of the system. The *camera* is mounted on the *X-Y translation mechanism*, whose movement is controlled by two *stepper motors* connected to an *EiBotBoard (EBB)*.The *housing* module elevates the *dish stage* above the camera.

It is important to note that the *dish stage* (see Figure 3) needs to be co-planar with the *base* for a flat field (to maintain focus of the camera on all areas of the dish). This component can also be raised or lowered to allow for different field resolutions, allowing to worms take up more or less frame space (ie. zoom).

1.2 Hardware

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

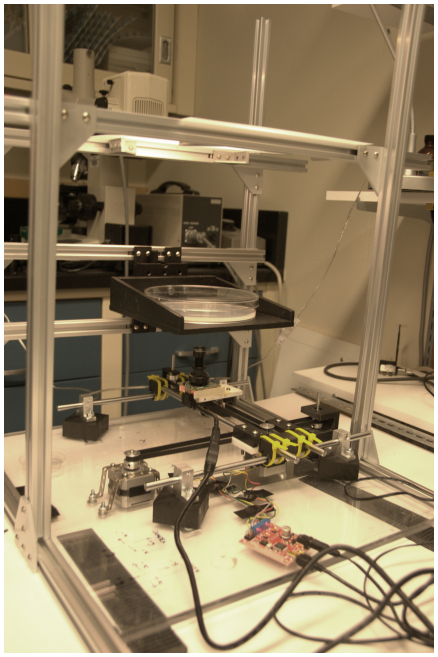

Figure 1: *housing* and *base*

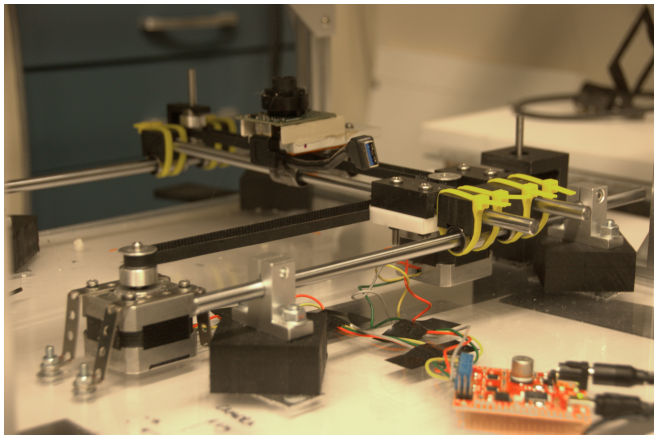

Figure 2: Close-up of *base*

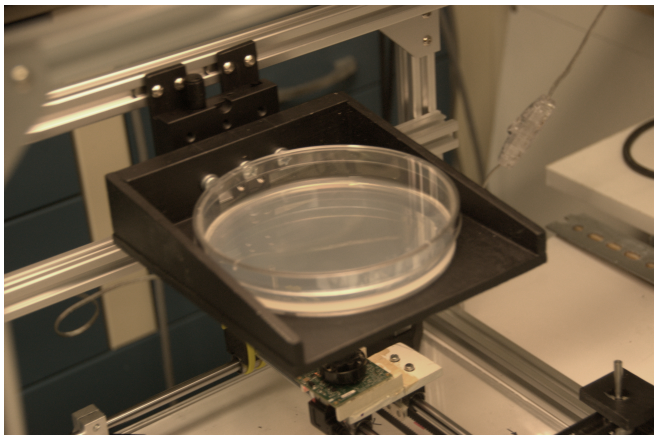

Figure 3: 15cm *dish* on *dish stage*

1.2.1 Motors and Motor Control Board

The stepper motors being used are easily available from the Evil Mad Scientist website [1]. The price point is $ 16/motor.

The motor control board is the `EggBotBoard` [2]. The price point is $50.

1.2.2 Imaging

The current camera being used is e-con System's `See3CAM_10CUG`. [3] It is a 1.5MP USB 3.0 camera capable of up to 45 fps at a resolution of 1280 × 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.

1.2.3 Computer

We are using a computer dedicated to the tracking system, though any laptop with similar configuration (outlined in Appendix **??**) should be compatible. The computer is the Intel NUC DN2820FYKH [4] with 8GB RAM, 500GB HD, and 2.4GHz dual-core Celeron cpu. It's quite small (4" x 4" x 2"). Most importantly, it has a USB 3.0 for the camera. The price point is $300.

1.3 Software

The tracking sofware was developed using the OpenCV 3.0.0 (dev) library [5] 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.

1.3.1 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 1.3.2).

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

1.3.2 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 4

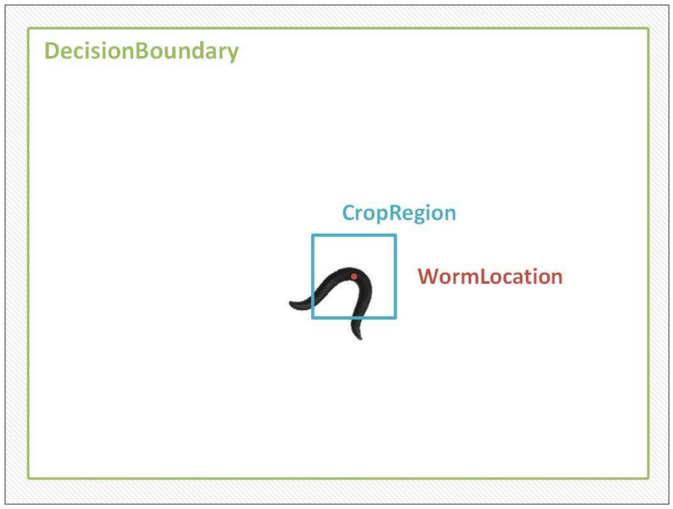

Figure 4: Geometry of the `findWorm` and `decideMove()` methods

The schematic, detailed overview of the system can be seen in Figure 5.

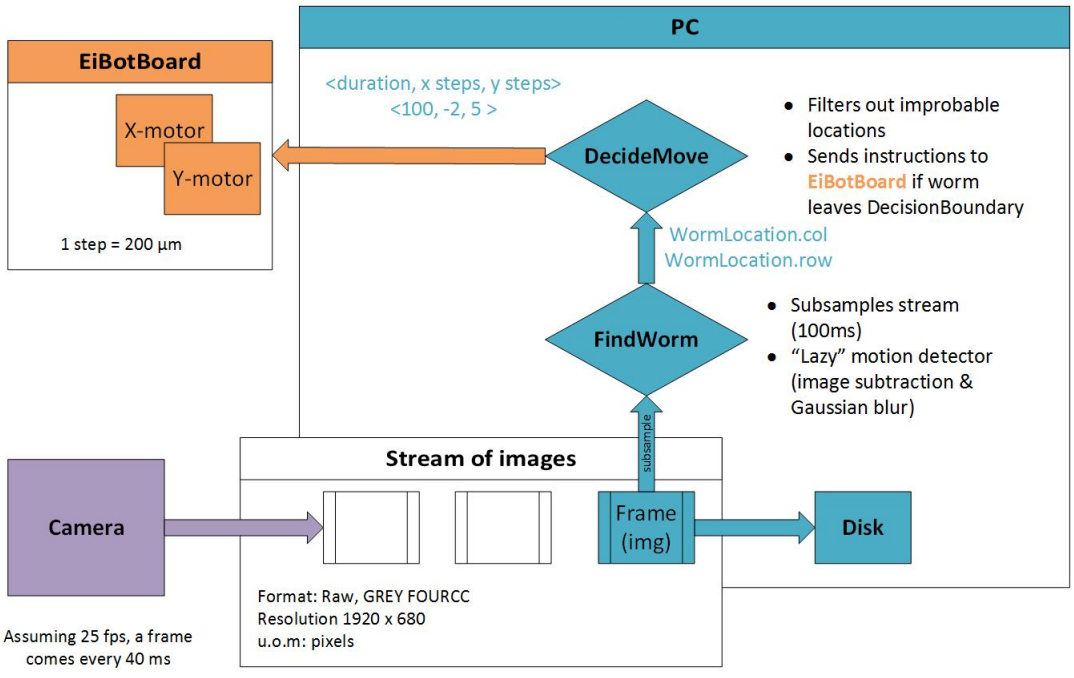

Figure 5: System diagram

1.3.3 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. [6, 7]. 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 ?? in Section ??. These changes currently make the system only compatible with grayscale cameras. The underlying issue here is that the conversion of BGR24 to GREY was assumed possible for every camera by the OpenCV authors, but it did not seem to be the case for our system. The developers of the camera in India make claim that they were able to use the OpenCV library with no issues but their resolution was never able to be replicated.

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.

1.3.4 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 [8]. 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.