The stages of my program are:

1. Load the channels, remove the slide, and equalize the histogram.
2. Compensate for uneven illumination.
3. Threshold and merge the resulting binary images.
4. Find the skeleton of the binary image. Detect endpoints and bifurcations (splits).
5. Search from the bifurcations to establish clusters.
6. Pair up the endpoints of the remaining non-cluster worms.
7. Build worms from clusters by growing them from endpoints, taking segments based on how similar they are to the rest of the worm.
8. Region filling in the binary image from the worms found in the skeletons.
9. Evaluation of the straightness and thicknesses of the worms to guess if they're alive. Comparison with the ground truth data by sizing the overlap.

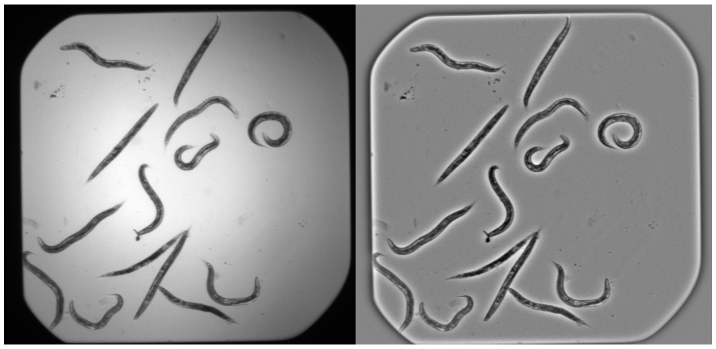
Fault cases:

1. Thresholding the first channel rarely fails. In this case we only use the second channel.
2. Border detection rarely fails. In this case we erase the outer portion of the image.
3. Incorrect segmentation of complex clusters.

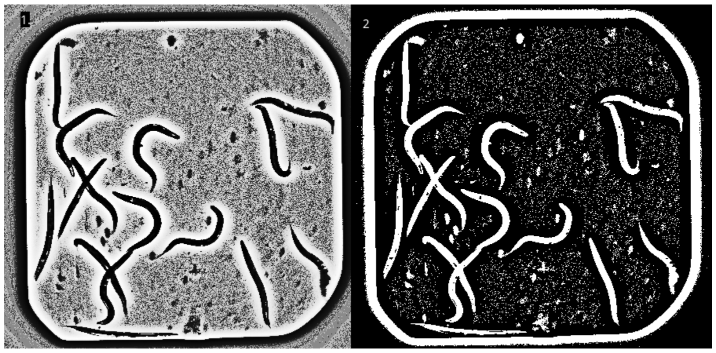
Some images are included below, however you can view the entire process live in the program itself.

Thresholding

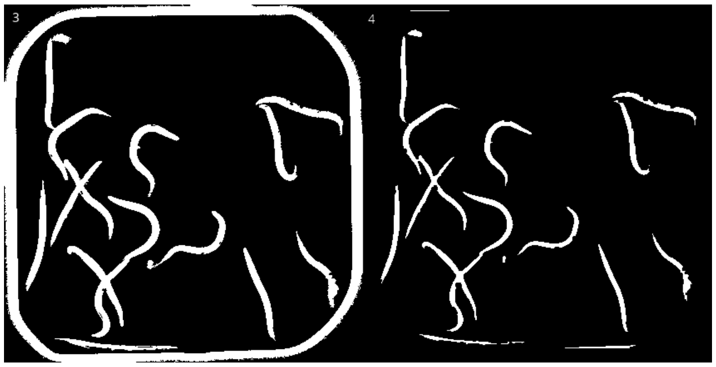
Firstly I apply some form of holomorphic filtering to get rid of uneven illumination. This is basically a high-pass filter applied to the logarithm of the image.



I then equalize the histogram. This is followed by simple thresholding, where pixels are set to 255 iff they're less than 30.



I use breadth-first search to remove noise - connected components under a certain size (both white and black) are removed.
Finally I detect and remove the slide border. Small remnants are later removed.



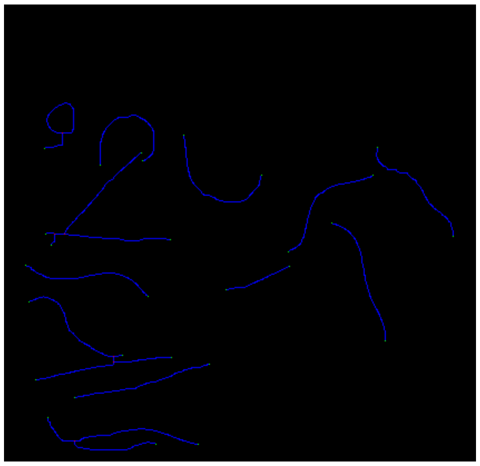
Skeletonization

Once a threshold has been found, I find the medial of the worms.
I first tried a simple approach as described here. This method was fast but lead to poor skeletons.

I then implemented the Zhang-Suen algorithm as described here. This gave me satisfactory results. Finally, I implemented the Guo-Hall algorithm described in this paper. It gave me similar results to the Zhang-Suen algorithm but took much longer.

Detection of endpoints and bifurcations

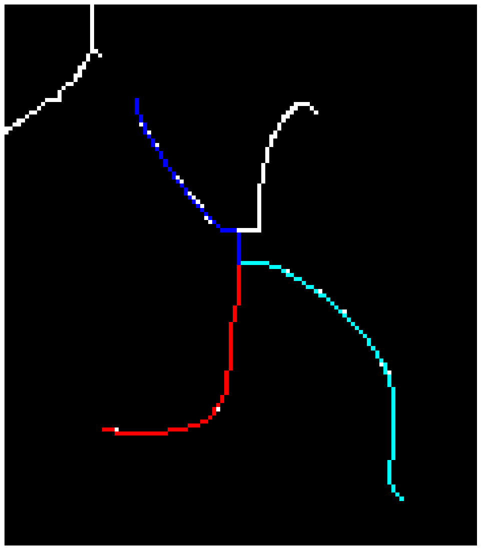
Detecting endpoints in the skeleton was easy - if the neighbourhood of a pixel has 6 consecutive blank spots it's an endpoint. Finding bifurcations is done by two manual rounds of BFS - if there are three or more points on the fringe it's a bifurcation.



Endpoints in green, bifurcations in red.

Graph searching to untangle clusters and build directional data

Normally shape fitting is used to untangle clusters. I thought I would try greedily growing worms from endpoints instead.
We search out from bifurcations to see which endpoints are connected in a cluster. This creates a graph, where the endpoints/bifurcations are nodes and the edges show which nodes are connected in the cluster. Whilst searching, we attach to each edge the directions that segment of the skeleton takes as points on the compass.
Once this is done, we pair the remaining endpoints (which are part of seperated worms). We then grow worms from each endpoint in the cluster. At each bifurcation the worm considers all potential segments and takes the one that agrees with its current shape the most. Contested edges are given to the best candidate.



A worm (dark blue) considers two possible edges. Its preference is shown in turquiose.

Region filling

We finish with a simple region filling algorithm, using the worms on the skeleton as markers and the binary image as regions. I implemented my own in order to merge small regions and preserve shape data.

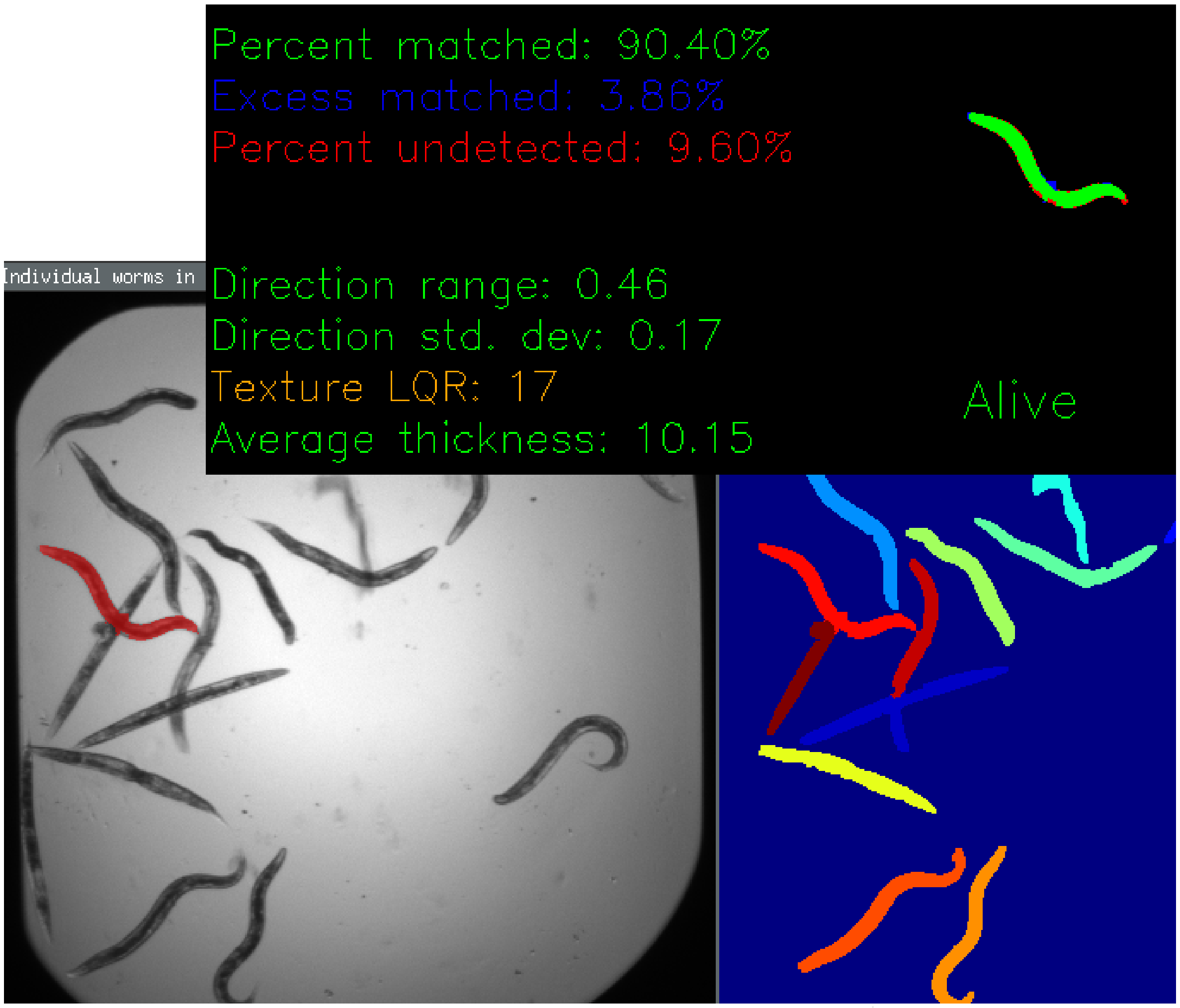
Detecting dead worms

I assign a score to each worm based on their curvature, thickness, and the LQR of intensities (texture). Dead worms are stained in the first channel, so an easy way of detecting the dead worms would just be to use that.

Programmatic comparison with ground truths

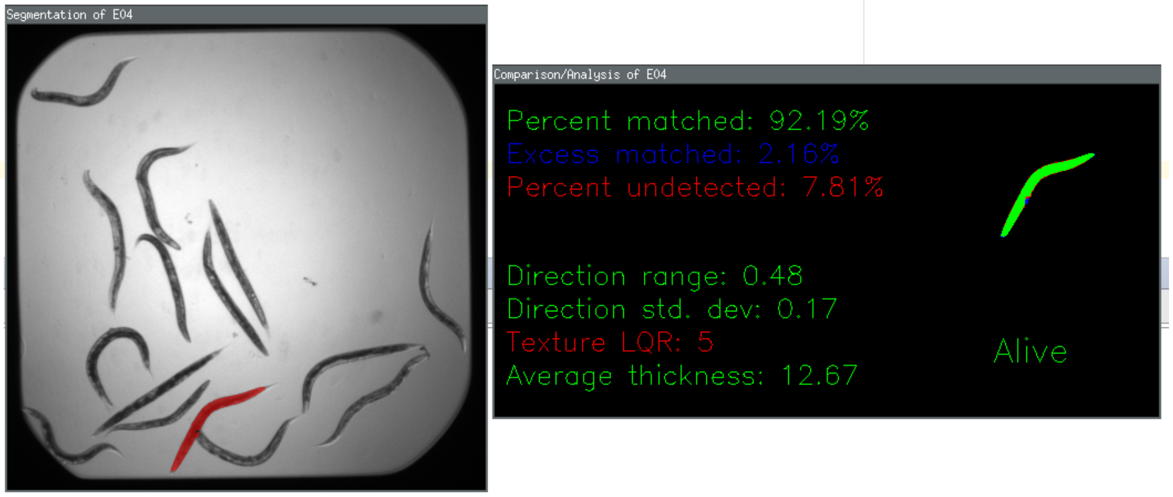
To evaluate the segmentation of individual worms we go through the 'eachworm' folder and see which ones overlap the most with our detected worms. The final thresholded result is compared with the foreground to evaluate how well my background seperation worked.
In both cases the overlap is coloured green, excess blue, and unmatched red. This, along with calculated percentages, is shown to the user so that performance is easily judged.

Some results



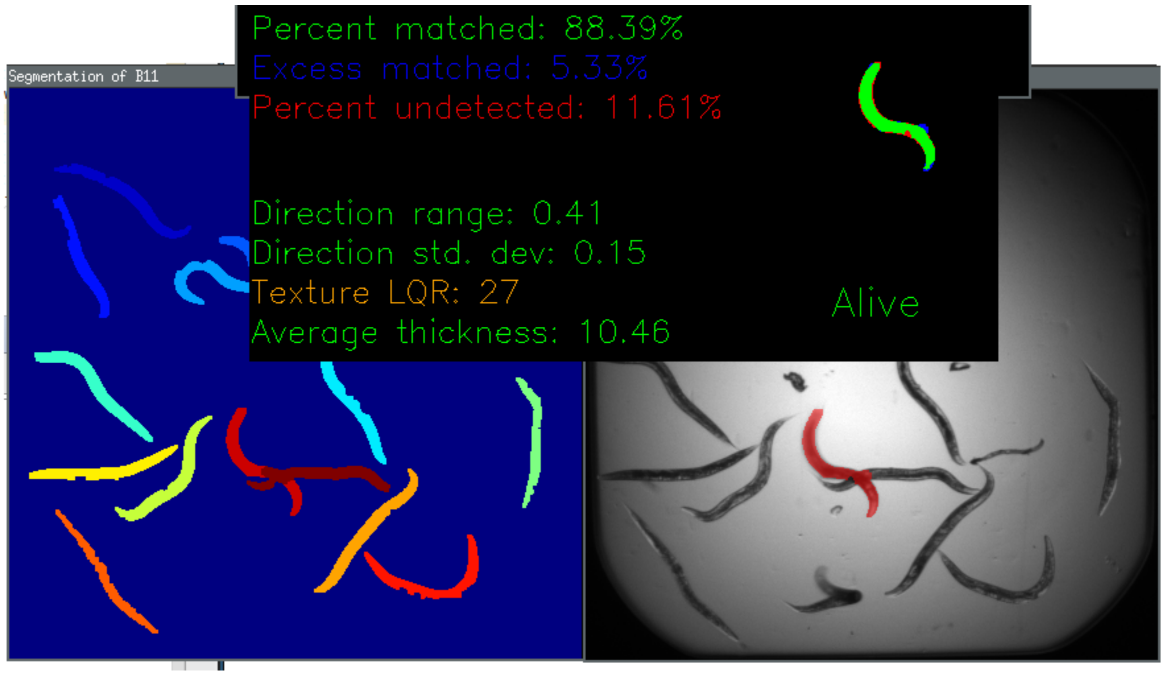
```
Percent matched: 90.40%
Excess matched: 3.86%
Percent undetected: 9.60%

Direction range: 0.46
Direction std. dev: 0.17
Texture LQR: 17
Average thickness: 10.15                     Alive
```



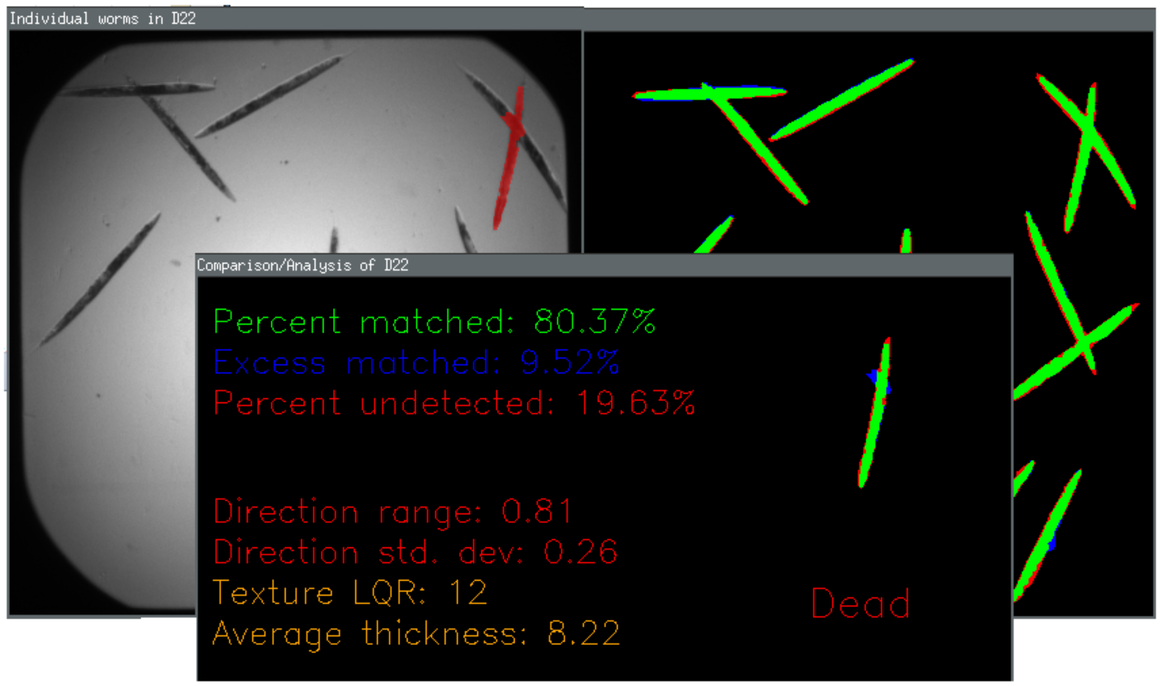
```
Percent matched: 92.19%
Excess matched: 2.16%
Percent undetected: 7.81%

Direction range: 0.48
Direction std. dev: 0.17
Texture LQR: 5
Average thickness: 12.67                     Alive
```



```
Percent matched: 88.39%
Excess matched: 5.33%
Percent undetected: 11.61%

Direction range: 0.41
Direction std. dev: 0.15
Texture LQR: 27
Average thickness: 10.46
                                        Alive
```



```
Percent matched: 80.37%
Excess matched: 9.52%
Percent undetected: 19.63%

Direction range: 0.81
Direction std. dev: 0.26
Texture LQR: 12
Average thickness: 8.22
                                        Dead
```

.