ANALYSIS OF RHYTHM GENERATION IN THE CAENORHABDITIS ELEGANS MOTOR CIRCUIT

Anthony Demharter Fouad

A DISSERTATION

in

Bioengineering

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2018

Supervisor of Dissertation

____________________
Christopher Fang-Yen

Assistant Professor of Bioengineering and Neuroscience

Graduate Group Chairperson

____________________
Ravi Radhakrishnan

Professor of Bioengineering and Chemical and Biomolecular Engineering

Dissertation Committee
Vijay Balasubramanian, Cathy and Marc Lasry Professor of Physics, University of Pennsylvania
Brian Chow, Assistant Professor of Bioengineering, University of Pennsylvania
Gal Haspel, Assistant Professor of Biological Sciences, New Jersey Institute of Technology
Robert Kalb, Professor of Neurology, Northwestern University
ANALYSIS OF RHYTHM GENERATION IN THE CAENORHABDITIS ELEGANS
MOTOR CIRCUIT

COPYRIGHT
2018
Anthony Demharter Fouad

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License
To view a copy of this license, visit
https://creativecommons.org/licenses/by-nc-sa/3.0/us/
ACKNOWLEDGMENTS

I would like to thank my advisor, Chris Fang-Yen, for driving my intellectual growth over the past 5 years. Chris constantly challenges me to rethink my assumptions and challenges me to justify my interpretations of experimental results. Moreover, during my time in Chris’s group, I have expanded my mathematical, programming, and analytical skills tremendously. Finally, Chris was a supportive and empathetic advisor who nurtured a constructive working environment, in large part by allowing me the freedom to explore questions from angles of my choosing, even if such explorations turned out to be unfruitful. It has been a great privilege to be educated in this lab.

I also wish to thank my thesis committee, Professors Vijay Balasubramanian, Brian Chow, Gal Haspel, and Robert Kalb, for their expert guidance throughout my thesis work. I am similarly grateful to David M. Raizen and Michael P. Nusbaum, who repeatedly offered their support and advice despite having no obligation to do so. Not only did this support group evaluate the technical merits of my progress, but they also provided insightful perspectives on my professional development and path forward. I have been tremendously fortunate to learn from such an intellectually diverse group of experts during my time at Penn. It has also been a distinct privilege to work with collaborators Mei Zhen and Quan Wen, who provided helpful feedback and numerous transgenic strains.

Finally, I am grateful to my family, including my brother Edward, mother Cheryl, and father Ashraf for their constant support through the highs and lows of my graduate education. I thank them all for listening to my weekly briefings and offering advice or
encouragement, and even discussing my work at a technical level. I am perhaps most grateful to my wife, Wassana, for enthusiastically choosing to spend her life with me in the fascinating but uncertain world of science. Despite the many unknowns that lay ahead, she has quickly become the bedrock of my life, grounding me back to the real world and supporting me unwaveringly.
ABSTRACT

ANALYSIS OF RHYTHM GENERATION IN THE CAENORHABDITIS ELEGANS MOTOR CIRCUIT

Anthony Demharter Fouad
Christopher Fang-Yen

Understanding the neuronal control of movement has been a central goal of neuroscience for decades. In many organisms, chains of neural oscillators underlie the generation of coordinated rhythmic movements. However, the sheer complexity of spinal locomotor circuits has made understanding the mechanisms underlying rhythmic locomotion in vertebrates challenging. The roundworm C. elegans generates rhythmic undulatory movements that resemble those of swimming vertebrates, but using only a few hundred neurons. The relative simplicity of this organism has allowed a complete synaptic map of the nervous system to be developed. Moreover, C. elegans has a three-day life cycle and is amenable to a powerful battery of genetic techniques that allow the molecular basis of circuit functions to be probed much more rapidly than is possible in more complex organisms. Because of these advantages, C. elegans offers the possibility of understanding the network, cellular, and molecular principles of rhythmic locomotion in deeper detail than has been possible in any other model organism. However, it is currently unclear where in the C. elegans motor circuit rhythms are generated, and whether there exists more than one rhythm generator. I used optogenetic and lesioning experiments to probe the nature of rhythm generation in the locomotor circuit. I found that rhythmic activity in different parts of the body can be decoupled by both methods, implying that multiple sections of forward locomotor circuitry
are capable of independently generating rhythms. By perturbing different components of the motor circuit, I localized at least two rhythmic sources to a network of cholinergic motor neurons that are distributed along the body. Moreover, I used rhythmic optogenetic manipulations to show that imposed rhythmic signals in any portion of the motor circuit can entrain oscillatory activity in the rest of the body, suggesting bidirectional coupling within the motor circuit. This organization, in which distributed oscillating circuits exist along the body but are closely linked by bidirectional coupling, is found in wide range of vertebrate and invertebrate animals. My results show that the functional architecture of the *C. elegans* motor circuit is highly analogous to that of much more complex organisms.
TABLE OF CONTENTS

ACKNOWLEDGMENTS ..............................................................................................................III
ABSTRACT ................................................................................................................................V
TABLE OF CONTENTS ............................................................................................................VII
LIST OF TABLES ..................................................................................................................X
LIST OF ILLUSTRATIONS .....................................................................................................XI

CHAPTER 1: INTRODUCTION .................................................................................................1
Rhythmic locomotion in animals ..............................................................................................1
C. elegans has unique strengths as a model in neuroscience .................................................5
The source of locomotory rhythms in C. elegans is unknown ..............................................7
Objective and overview ........................................................................................................11

CHAPTER 2: EXPERIMENTAL TOOLS ..............................................................................12
Optical tools for acute manipulation of C. elegans neurons .................................................12
Investigating circuit function with targeted optogenetics ....................................................13
Lesioning at variable precision and depth is required for some circuit experiments ........20

CHAPTER 3: DISTRIBUTED RHYTHM GENERATORS UNDERLIE CAENORHABDITIS
ELEGANS FORWARD LOCOMOTION .............................................................................25
ABSTRACT ............................................................................................................................26
INTRODUCTION ...................................................................................................................26
RESULTS ................................................................................................................................34
Rhythmic posterior undulation persists despite anterior paralysis .....................................34
The head and tail are capable of simultaneous oscillations at different frequencies .........42
Most premotor interneurons are not essential for rudimentary forward movement or 2FU 50
Several classes of motor neurons are not required for forward locomotion or 2FU .........65
B-type motor neurons, as a class, are essential for 2FU ......................................................67
Multiple lesion-separated VNC segments are capable of independent rhythm generation 72
B class motor neurons are necessary for rhythmic wave generation in VNC-lesioned worms .................................................................77
Undulations can arise in arbitrary portions of the VNC motor circuit ................................85
Rhythmic motor entrainment is possible in both the anteriorward and posteriorward directions ...............................................................88
DISCUSSION .......................................................................................................................95
MATERIALS AND METHODS .............................................................................................104
Strains ....................................................................................................................................104
Construction and validation of the optogenetic targeting system ....................................108
The construction and validation of the optogenetic targeting system is discussed in Chapter 1 ..........................................................108
Optogenetic inhibition and stimulation .............................................................................108
Neuron and muscle photoablation using miniSOG .........................................................109
Ablation of premotor interneurons by ICE .........................................................................111
<table>
<thead>
<tr>
<th>File Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Images.cpp</td>
<td>280</td>
</tr>
<tr>
<td>PVcamAccessoryFuncs.h</td>
<td>289</td>
</tr>
<tr>
<td>PVcamAccessoryFuncs.cpp</td>
<td>290</td>
</tr>
<tr>
<td>SimpleSerial.h</td>
<td>299</td>
</tr>
<tr>
<td>SimpleSerial.cpp</td>
<td>300</td>
</tr>
<tr>
<td>StageAccessoryFuncs.h</td>
<td>301</td>
</tr>
<tr>
<td>StageAccessoryFuncs.cpp</td>
<td>301</td>
</tr>
<tr>
<td>stdafx.h</td>
<td>311</td>
</tr>
</tbody>
</table>

**BIBLIOGRAPHY**  ................................................................. 314
LIST OF TABLES

Table 1: Transgenic arrays acquired or generated for this study..........................105
Table 2: Additional strains generated by combining transgenes in Table 1.............107
Table 3: conditions and mutants shown in Figure A1.3........................................146
Table 4: conditions and mutants in Figure A1.S1..................................................148
LIST OF ILLUSTRATIONS

Figure 2.1. Real time optogenetic manipulation of moving C. elegans ..................16
Figure 2.2. Validation of the optogenetic targeting system .................................18
Figure 3.1. Overview of curvature analysis and models of rhythm generation ........33
Figure 3.2. Anterior undulation is not required for posterior undulation ...............36
Figure 3.2 Figure Supplement 1. Tail undulation after gross head lesioning .........40
Figure 3.3. Disruption of motor coupling in the neck desynchronizes head and tail oscillations .................................................................46
Figure 3.3 Figure Supplement 1. Additional disruptions to motor coupling cause 2FU ..48
Figure 3.4. VNC premotor interneurons, D-type motor neuron signaling, and individual B-type motor neurons are not required for 2FU .........................................................54
Figure 3.4 Figure Supplement 1. VNC premotor interneurons and several VNC motor neuron classes are not required for de-synchronized oscillations ..................57
Figure 3.4 Figure Supplement 2. Ablation of the AVB interneurons .......................58
Figure 3.4 Figure Supplement 3. Subsets of B-type motor neurons are not required for 2FU .........................................................................................61
Figure 3.4 Figure Supplement 4. B-type motor neurons posterior to the vulva are not required for 2FU .......................................................................................63
Figure 3.5. B motor neurons are required for 2FU ...............................................70
Figure 3.5 Figure Supplement 1. Behavior after ablation of VB motor neurons ....71
Figure 3.6. Undulations generated in the tail after severing the dorsal and ventral nerve cords ..............................................................................76
Figure 3.6 Figure Supplement 1. Body undulations after severing the VNC and DNC do not require the A motor neurons .................................81
Figure 3.6 Figure Supplement 2. B motor neurons are required for body undulations after severing the VNC and DNC ........................................83
Figure 3.7. Undulations are generated after VNC/DNC lesioning in arbitrary locations .87
Figure 3.8. Head undulation frequency can be entrained by mid-body optogenetic manipulation ...........................................................................91
Figure 3.8 Figure Supplement 1. Rhythmic activity in the mid-body B motor neurons is sufficient for posterior-to-anterior entrainment, and muscle-to-muscle gap junctions are not required. .............................................................93
Figure 3.9. A model for C. elegans forward locomotion ......................................101
Figure 4.1. Computational modeling of the leech swim CPG ..............................122
Figure 4.2. Overview of proposed methods of rhythm generation ......................124
Figure 4.3. Identifying a CPG or a reflex loop by its response to brief inhibition of movement. .................................................................126
Figure A1.1. Light microscope with dark field illumination ..............................138
Figure A1.2. Fat-staining structures appear bright in dark field images ..........139
Figure A1.3. Scattering density correlates with ORO staining density ..........144
Figure A1.S1: Validation experiment using other mutants and conditions ....147
Figure A1.4. Scattering density increases sharply between the L1 and L2 larval stages .................................................................................152
Figure A1.S3: Oil-Red-O stained, well-fed N2 worms at various larval stages .....153
Figure A1.5. Scattering in L1 worms corresponds to gut granules; scattering in older worms does not. .................................................................157
Figure A1.S2: Droplets visible in dark field images are not LROs .........................158
Figure A1.S4. Fat loss is not correlated with prior activity level. ..........................161
Rhythmic locomotion in animals

Nearly all animals exhibit rhythmic locomotion. Although the manner in which different species locomote varies from upright walking to limbless swimming, in all cases the observed motor output requires rhythmic signaling from the underlying motor circuit. Because of the ubiquity of rhythmic locomotor systems in animals, understanding the basis of rhythmic locomotion in animals has been a central goal of neuroscience for more than a century (Kristan et al., 2005).

Early studies of vertebrate locomotor circuits revealed that cats in which inputs from the brain have been eliminated are fully capable of generating well-formed rhythmic walking without input from the brain. Moreover, this movement did not require intact sensory or proprioceptive afferents, and in fact the regularity of such behavior was greater when afferents were also cut. This work thus localized central rhythm generating circuits for locomotion to the spinal cord (Brown, 1911), and helped form the foundation for a century of studies of the spinal locomotor circuits.

During the succeeding decades, rhythmic locomotion has been investigated in a wide variety of vertebrate and invertebrate animals. Invertebrates that have been studied in this capacity include leeches (Kristan et al., 2005) and crayfish (Mulloney and Smarandache-Wellmann, 2012). Vertebrate models have included rodents, zebrafish, tadpoles, and lampreys (Kiehn, 2006, 2016). Rhythmic locomotion in most of these species is encoded in the spinal circuits (for vertebrates) or ventral nerve cord circuits. In
each case, rhythmogenic capability is distributed along all or a portion of the spinal or nerve cord, and lesions at various positions can reveal multiple oscillating units (Kiehn, 2006; Mulloney and Smarandache-Wellmann, 2012).

An overview of rhythmic swimming control in the leech is shown in Figure 1.1A. The ventral nerve cord in each segment of the animal houses a ganglion that contains the oscillator interneurons and motor neurons necessary to generate rhythmic locomotion in the segment (Kristan et al., 2005; Marder et al., 2005; Mullins et al., 2011). When separated from the rest of the nerve cord, most of the ganglia are capable of generating rhythmic oscillatory signals, especially when given an appropriate stimulus (Weeks, 1981). When intact, oscillatory activities in each ganglia have a predictable phase lag relative to the anterior ganglia in order control wave propagation along the body (Marder et al., 2005).

A similar distributed rhythm generating architecture is found in vertebrates. In the lamprey, for example, chains of at least three out of the ~100 spinal segments produce fictive rhythmic swimming with appropriate phase lags (Figure 1.1B), and even single segments are capable of rhythmic activity (Mullins et al., 2011). The distributed nature of rhythm generation is not restricted to these swimming organisms; the spinal circuits for rhythmic hindlimb locomotion in mammals, including but not limited to the decerebrate cats discussed above, are also dispersed along the caudal portion of the spinal cord (Kiehn, 2006).
Figure 1.1. Organization of rhythm generators in the leech swim CPG. Reproduced from (Mullins et al., 2011). (A) Oscillatory circuits are found in each of the 18 ganglia along the body. Each ganglia, when surgically isolated from the others, is capable of producing rhythmic locmotor signals when provided with the appropriate stimulus. In the intact circuit, intersegmental coupling locks the phase of each oscillator such that each one slightly lags the one anterior to it, allowing a rhythmic wave to propagate. (B) A similar architecture governs lamprey swimming. Oscillators exist in each of the ~100 spinal segments. When intact, posterior segments (e.g. R19 and L19) maintain a fixed phase lag relative to anterior segments (e.g. R7 and L7) (Weeks, 1981; Kristan et al., 2005; Mullins et al., 2011).
In all of these systems, there are differences in the frequency, excitability, and other properties of the spinal segments or segmental ganglia. In the leech, for example, the observed frequencies of each circuit unit in isolation vary considerably, and may not necessarily match the natural frequency of locomotion (Zheng et al., 2007), and some of the ganglia produce weaker or more sporadic fictive swimming activities in isolation than others (Mullins et al., 2011).

How can a distributed network of rhythm generators, each with different rhythmic properties, organize itself into a single coherent locomotory pattern? This question has posed a challenge to the study of motor circuits. It can be broadly appreciated that the different oscillatory units are in segmented animals such as leeches and lampreys are linked by intersegmental coupling that can span significant fractions of the body to allow for tight coordination between all of the motor units (Kristan et al., 2005; Mullins et al., 2011). This communication, which mediated by intersegmental axons of oscillatory interneurons in the leech, has approximately the same strength in the anteriorward and posteriorward directions (Kristan et al., 2005), suggesting that intersegmental coordination is bidirectional.

Coordination is also controlled by proprioceptive feedback. The leech ventral nerve cord contains stretch receptor neurons that relay information about the current bending state of the animal to the swim CPG. Although such feedback is not essential to rhythm generation, it does heavily regulate it; injection of sinusoidal current into the proprioceptive neurons can entrain the swim CPG (Yu and Friesen, 2004), and severing the intersegmental axons in a live animal actually does not prevent propagation of a bending wave across the lesion due to physical bending triggering the posterior sensor (Yu et al., 1999). As is the case for other features of rhythmic circuits discussed here,
similar mechanisms for sensory control of locomotion also exist in a wide range of vertebrate models. In fact, rhythmic manipulation of body shape can entrain the underlying rhythm generators in lampreys and mammals (Andersson et al., 1981; Grillner et al., 1981).

The decades of pioneering studies in limbed and swimming model organisms summarized above have defined the fundamental principles of rhythmic locomotor control that are found to some extent in all animals (Kiehn, 2006; Goulding, 2009; Kiehn, 2016). Despite this impressive volume of work, an integrated understanding of how network, cellular, and molecular principles interact to generate and coordinate rhythmic behaviors has remained elusive. A key reason for this shortcoming is the sheer complexity of the animals being studied. Even in relatively simpler model organisms such as leeches, many neuronal components of the locomotor rhythm generator remain unidentified, and the mechanisms by which these components interact to generate and coordinate locomotion are unclear (Kristan et al., 2005; Kiehn, 2016). While it is not impossible to approach the fundamental principles of rhythm generation in leeches or any of the available vertebrate models, the timeframe and resources required to fully understand these systems would be considerable, verging on prohibitive. A more efficient way to approach the problem would be to study an even simpler organism that generates similar behaviors with far fewer components.

**C. elegans** has unique strengths as a model in neuroscience

In the late twentieth century, the nematode *C. elegans* emerged as a model organism in genetics and neuroscience (Brenner, 1974). This millimeter-long roundworm generates
rhythmic sinusoidal swimming and crawling behaviors using only a few hundred neurons (Altun and Hall, 2011), several orders of magnitude less than the number found in any vertebrate spinal circuit or even the leech motor circuit (Kiehn, 2006; Mullins et al., 2011; Kiehn, 2016). The 300 neurons have stereotypical locations and connectivity, and were mapped by electron microscopy to forming a nearly complete connectome, or map of all circuit connectivity in the nervous system (White et al., 1976; White et al., 1986). The relative simplicity of the system and availability of a cellular-level connectome have enabled decades of intensive study on the worm’s nervous system, allowing the structure function of numerous individual neurons and small networks to be dissected in detail (Altun and Hall, 2011).

In addition to the detailed investigations of cells and networks that are possible in this organism, C. elegans has a three-day life cycle and is amenable to a powerful battery of forward and reverse genetic techniques (Ahringer, 2006). Mutant or transgenic animals can be generated and analyzed in a matter of weeks, compared to months or not at all in mammalian models or leeches and lampreys, respectively (Evans, 2006; Kiehn, 2016).

Despite the worm’s outstanding simplicity and amenability to neurobiological experimentation, the C. elegans genome features extensive homology to the human genome, and its nervous system makes use of many familiar ion channels, neurotransmitters, and synaptic machinery (Lai et al., 2000; Brockie et al., 2001a; Brockie et al., 2001b; Brockie and Maricq, 2006; Chase and Koelle, 2007; Rand, 2007; Mellem et al., 2008).

The amenability of C. elegans to experimentation, combined with its high degree of biological relevance to more complex animals, positions the worm as the ideal organism
for achieving an integrated understanding of the network, cellular, and molecular principles that underlie rhythmic locomotion in animals.

The source of locomotory rhythms in *C. elegans* is unknown

In order to capitalize on the promise of *C. elegans* to deliver an integrated understanding of rhythmic locomotion, it is first necessary to understand the basic structural and functional architecture of the worm’s motor circuit. Despite the availability of a cellular level connectome, the origin of rhythmic signals that drive *C. elegans* locomotion has been a mystery (Von-Stetina et al., 2006; Gjorgjieva et al., 2014). However, laser ablation, calcium imaging, and microfluidic studies have revealed essential details about the motor circuit.

In the 1980s, Martin Chalfie *et al* sought to assign functional roles to many *C. elegans* sensory neurons, interneurons, and motor neurons by ablating them and assaying behavioral defects (Chalfie et al., 1985). *C. elegans* neurons can be ablated at the embryonic or early larval stage with a pulsed nanosecond or femtosecond laser (Fang-Yen et al., 2012). A more detailed discussion of ablation methods is presented in Chapter 2. Of the findings relevant to locomotor behaviors, Chalfie found that the interneurons AVB and PVC were required for normal forward crawling, although limited locomotion directed by the head was possible in their absence. Removal of B and A type motor neurons, albeit in young larvae, inhibited forward and reverse locomotion, respectively. Ablation of the D type motor neurons disrupted locomotion in both directions (Chalfie et al., 1985), as did genetic manipulations that block synaptic output from D motor neurons (McIntire et al., 1993a; McIntire et al., 1993c). There is one other
class of motor neurons, called AS motor neurons, that has not been investigated in detail.

The specificity of the B and A motor neurons to forward and reverse locomotion was later confirmed by neuronal calcium imaging. In these experiments, a fluorescent calcium sensor that increases in intensity or changes color upon calcium binding is used as a proxy for neuronal activation. The B motor neurons are preferentially active during forward locomotion (Haspel et al., 2010), and indeed are rhythmically active in phase with body bending (Kawano et al., 2011; Wen et al., 2012). An overview of the known circuit for forward locomotion is shown in Figure 1.2.

![Figure 1.2. Overview of the circuit for locomotion. Adapted from (Von-Stetina et al., 2006).](image)

Excitatory DB/VB and inhibitory DD/VD motor neurons are required for forward locomotion. D type motor neurons are wired to inhibit bends on the side of the animal undergoing relaxation at any point in the bending cycle. The PVC and AVB neurons activate these motor neurons to promote forward locomotion (Chalfie et al., 1985;
Although little is known about how these interneurons and motor neurons generate forward locomotion, it is known that the B type motor neurons propagate bending waves along the body at least in part by proprioception. Quan Wen et al used microfluidic channels to trap worms by the midsection in straight or bent postures, leaving the head and tail free to move (Wen et al., 2012). Under these conditions, bending waves originate in the head but do not pass through the trapped region of the body. The posterior region of the body remains paralyzed with the same curvature as the trapped region (Figure 1.3).

Figure 1.3. Proprioceptive coupling in the forward motor circuit.

Adapted from (Wen et al., 2012). When worms are trapped by their mid-body in a microfluidic device that is straightened (A,B) or curved (C), the posterior (p) adopts a
curvature similar to the curvature of the trapped region. Waves originating from the head
do not pass through the trapped region.
The B type motor neurons are required for this proprioceptive coupling to occur; inhibiting all B motor neurons prevents the tail from following the curvature of the trapped region. Inhibiting A or D motor neurons does not prevent coupling. It is, however, unclear whether the B motor neurons sense changes in curvature themselves, or whether they are signaled from another source (Wen et al., 2012).

Because partially trapped worms exhibit rhythmic behaviors in the head while the tail passively follows the curvature of the trapped region, this work lent support to a previous computational model postulating that the worm contains only a single central pattern generator for forward locomotion, located in the head (Karbowski et al., 2008; Wen et al., 2012). However, the work did not rule out the possibility that posterior motor circuits have rhythmogenic capability that was not visible in the trap assay. The work also did not investigate tail movement when a more anterior portion of the worm is immobilized. We sought to investigate the identities of the C. elegans rhythm generator(s).

**Objective and overview**

In this work I sought to identify the location(s) and identities of locomotor rhythm generator(s) that drive C. elegans forward locomotion. In Chapter 2, I introduce experimental techniques that I developed in pursuit of this goal. In Chapter 3, I describe experiments that led to our identification of multiple rhythm generators within the forward motor circuit. In Chapter 4, I provide my perspective on future directions for this work. In Appendix I, I describe an offshoot project on measuring worm fat levels by dark field imaging. In Appendix II, I present my C++ software for controlling the optogenetic targeting system.
CHAPTER 2: EXPERIMENTAL TOOLS

This chapter introduces two experimental tools that I developed to allow interrogation of the C. elegans motor circuit. The description of the optogenetic targeting system is reproduced with light modifications from a publication in eLife (Fouad et al., 2018b). The description of the infrared laser ablation system is adapted from a manuscript under preparation by Fouad et al.

Optical tools for acute manipulation of C. elegans neurons

Despite its advantages as a model organism in neuroscience, study of the C. elegans motor circuit has been limited by the tools available to manipulate the animal at micron-scale. Traditional electrophysiology, the workhorse of the classical experiments summarized above, is extremely difficult in worms due to their small size, poor accessibility to neurons, and hydrostatic skeleton. While some attempts to perform electrophysiology in worms have been reported, reports are limited to a sparse sampling of the available neurons and muscles (Liu et al., 2014a; Gao et al., 2015; Gao et al., 2017), and have yet to shed light on the origin or nature of rhythm generation in the worm.

Fortunately, the available C. elegans genetic toolkit allows the use of recently developed optical tools to manipulate or record neuromuscular activity. Optogenetics, the use of genetically encoded light sensitive ion channels to depolarize or hyperpolarize electrically excitable cells, has emerged as an especially potent tool for investigating circuit function in C. elegans and many other species. Manipulations can be performed
on command by supplying light of the appropriate wavelength. (Nagel et al., 2003; Nagel et al., 2005; Zhang et al., 2007; Chow et al., 2010; Fang-Yen et al., 2015).

Investigating circuit function with targeted optogenetics

Expressing optogenetic channels under the control of well characterized promoters allows manipulations to be performed on defined cell types. However, the specificity afforded by available promoters can be insufficient to answer certain questions. For example, as described in Chapter 2, I asked whether locomotory behavior in different parts of the worm can be decoupled by inhibition of motor neurons or muscles in the middle of the body. No promoter is known to drive expression in only certain subsets of these cells. One solution to this problem is to restrict illumination to only some of the cells by optical methods. This technique, while straightforward in principle, is complicated by the worm’s movement and requires rapid tracking of the animal’s changes in position and body shape to update the region of illumination.

Targeted optogenetic manipulation of moving *C. elegans* was developed by two groups (Leifer et al., 2011; Stirman et al., 2011). The centerpiece of both systems is a digital light processor (DLP), an array of 1024 by 768 independently movable mirrors that can be switched to the “on” state, reflecting light to a tiny portion of the field of view through a microscope objective, or the “off” state, reflecting it elsewhere. The mirrors that should be switched at any given time are determined by machine vision software that tracks the worm’s swimming across a microscope slide and defines the region of the animal to be targeted by laser illumination (Figure 2.1 A, B).
To complete the experiments described in Chapter 2, I constructed and programmed a modified closed-loop optogenetic illumination system that is adaptable to experiments ranging from the original body-region targeting described above to targeting of a specific, fluorescently labeled neuron (Figure 2.1 C, D).
Figure 2.1. Real time optogenetic manipulation of moving C. elegans.

(A) Schematic of an optogenetic targeting system, reproduced from (Leifer et al., 2011). Red light illuminates the worm to allow dark field tracking of its shape and position on a microscope stage. A laser, controlled by a DLP (also referred to as a Digital Micromirror Device or DMD), is projected onto specific regions of the animal to allow optogenetic manipulation of specific neurons or muscles.

(B) Illustration of the machine vision software, also reproduced from (Leifer et al., 2011). The worm’s contours are segmented, and centerline is drawn from the head to the tail, and a region of the body, defined in centerline coordinates, is selected for illumination.

(C), (D) Schematics of the optogenetic targeting system developed and applied in this work. I wrote extensive C++ software (Appendix II) to allow it to perform a wide variety of experiments. Variant (C) allows tracking and targeting of a specific neuron labeled by a red fluorescent protein, potentially increasing the resolution beyond the 30 um resolution reported for dark field targeting. Variant (D) is similar to the original design and targets body regions.
The dark field variant in Figure 1.1 D was used extensively in the experiments described in Chapter 2. The fluorescence targeting variant was not used in experiments described here, but is being used for new experiments arising from these studies.

The system was built around a Leica DMI4000B microscope. Dark field illumination was provided by red LEDs and worms were imaged with a sCMOS camera (QImaging optiMOS). I wrote custom written C++ software (Fouad et al., 2018a) to segment images of the worm. The user selects a targeting method (e.g. dark field body region) and region of the worm to illuminate in each experiment. The laser, gated by a DLP as described above, is focused onto the worm through a 10X objective. My software updates the laser position at 40 Hz to allow accurate tracking of the freely moving worm. A green laser (Shanghai Laser & Optics Century GL532T3-300, 532 nm wavelength, irradiance 10 mW/mm² at focal plane) was used for all activations of the inhibitory opsins (e.g. NpHR or Arch), and a blue laser (Shanghai Laser & Optics Century BL473T3-150, 473 nm wavelength, irradiance 4 mW/mm² at focal plane) was used for activation of the excitatory opsin ChR2. Worms were mounted in a solution of 17% dextran in NGMB (Fang-Yen et al., 2010) in an 80 µm thick chamber between a microscope slide and cover glass, separated by glass beads. NGMB is identical to NGM (Stiernagle, 2006) but lacks agar, peptone, or cholesterol.

To determine the accuracy of my illumination system, I studied transgenic worms expressing the excitatory opsin ChR2 in the RIS head interneuron under the control of the aptf-1 promoter (Fouad et al., 2018b). It was previously reported that these worms become immobile when RIS is stimulated by illumination with blue light (Turek et al., 2013). I used my system to target a thin band of blue light to various locations along the centerline of each worm, and measured the amplitude of head bending waves as an
output (Figure 2.2). The most slowing occurred at body coordinate 14 out of 100. The nearest coordinate with no slowing, body coordinate 20 out of 100, suggests that my optogenetic targeting system has a resolution of about 6% of a worm’s body length, similar to that reported for a the original system described above (Leifer et al., 2011).

Figure 2.2. Validation of the optogenetic targeting system.

A thin band of blue laser light was projected onto various body coordinates of the worm ranging from the head (body coordinate 0) to the tail (body coordinate 100). A more detailed description of the body segmentation method can be found in Chapter 3. The output, $M_2/M_1$, is the ratio of bending wave amplitude during illumination ($M_2$) over prior to illumination ($M_1$). As shown, bending amplitude was minimized when the band was
targeted to body coordinate 14 out of 100, and not affected when the band was targeted to body coordinate 20.
Lesioning at variable precision and depth is required for some circuit experiments

While optogenetics has been a fairly recent development in neuroscience, surgical techniques have been a mainstay of *C. elegans* circuit research for decades (Fang-Yen et al., 2012), and was even used to generate an initial characterization of the interneurons and motor neurons that drive locomotion (Chalfie et al., 1985). Briefly, these techniques involve using a nanosecond or femtosecond laser to induce local plasma formation to ablate cell nuclei.

The experiments described in chapter 2 required laser ablations neurons or severing the main nerve cords that run along the anterior-posterior axis of the body. While a nanosecond laser is routinely used for the former procedure, severing the ventral nerve cord (VNC) with this technique has not been reported, and my attempts to do so generally either failed to cut the nerve or damaged the outer cuticle.

I hypothesized that these limitations were due to the effect of optical aberrations interfering with the laser's ability to come to a focus at planes more than 3-5 microns deeper than the slide's coverglass. However, we found that a slightly different method, focusing a pulsed *infrared* laser through the objective to induce lesions by locally elevated temperature, was able to damage tissues at the depths required to lesion either the cells of nerve cords of adult worms.

The optical layout of the infrared laser system, which is adapted from one previously described for a different purpose (Churgin et al., 2013), is shown in Figure 2.3.
Figure 2.3. Infrared laser surgery system.

An infrared laser, modulated by a pulse generator, is focused through a microscope objective along with green fluorescent excitation light to excite fluorescence in the targeted neuron or nerve. Returning red fluorescence light is sent to a CMOS camera to visualize the targeted structure.
Figure 2.4. Precise ablation of VNC motor neurons by infrared laser irradiation.

(A) Ablation of VB9 leaving the neighboring motor neurons intact. (B) Fraction of animals in which VB9 and its neighbors remained alive after irradiation with a single pulse at the specified length. A pulse length of 0.8 ms reliably killed VB9 but not the neighboring cells. (C) Fraction of animals in which VA11 remained alive after irradiation with a single 0.8 ms pulse offset by the specified distance.
I found that I could kill a ventral nerve cord motor neuron in fourth stage larvae without apparent damage to its neighbors by targeting its cell body with a single 0.8 ms pulse of 400 mW infrared laser radiation (Figure 2.4 A, B). At this dosage, the radius of cell death around the target was about 5 microns. Although this resolution is not as precise as the existing technique, I found that lesioning at this depth within a fourth stage larvae was virtually impossible using a nanosecond laser. Lesions are normally induced in first stage larvae due to their small size, but developmental compensation for the missing neurons is an issue of some concern when using this method (Fang-Yen et al., 2012). By ablating at the final larval stage, I likely minimized that chances that developmental programs in the worm would have time to compensate for the missing cells by changing synaptic connections among the remaining neurons.

The infrared ablation technique, unlike traditional techniques involving plasma formation, is highly tunable allowing for larger or more resistant structures, like the VNC, to be lesioned. I found that 0.8ms pulses were completely ineffective at severing the VNC (not shown), and a pulse at 2.0 ms (more than double that sufficient to kill a cell) was only partially reliable in severing the VNC. However, a train of ten 2.0 ms pulses was highly effective at severing this nerve and left the circuits on both sides of the lesion independent but otherwise functional (Figure 2.5 and Chapter 2). This technique was invaluable in dissecting the function of the motor circuit as described in Chapter 2.
Figure 2.5. Severeing the VNC by infrared laser irradiation. (A) Depiction of the VNC, labeled by Pacr-5:wCherry, without application of infrared light or 4 hours after application of 10 pulses of 2 ms each. The cord is severed after 10 pulses. (B) A single pulse was only sometimes effective at severing the nerve cord, but trains of 10 or 20 pulses always severed the VNC.
CHAPTER 3: DISTRIBUTED RHYTHM GENERATORS UNDERLIE CAENORHABDITIS ELEGANS FORWARD LOCOMOTION

Anthony D. Fouad¹, Shelly Teng¹, Julian R. Mark¹, Alice Liu¹, Pilar Alvarez-Illera¹, Hongfei Ji¹, Angelica Du¹, Priya D. Bhigoo¹, Eli Cornblath¹, Asuka Guan², and Christopher Fang-Yen¹,³

1. Department of Bioengineering, School of Engineering and Applied Science, University of Pennsylvania, Philadelphia PA 19104, USA

2. Lunenfeld-Tanenbaum Research Institute, Mount Sinai Hospital, Toronto, Ontario M5G 1X5, Canada

3. Department of Neuroscience, Perelman School of Medicine, University of Pennsylvania, Philadelphia, PA 19104, USA

This chapter is a lightly edited version of my paper published in the journal eLife (Fouad et al., 2018b). All data in this chapter was collected either by me or by other students operating under my immediate supervision. I conceived of the project independently during my qualifying exam. Most transgenic strains were contributed by external collaborators (Table 1, especially A. Guan from the lab of Mei Zhen), but I performed all crosses and generated several strains myself (All YX strains in tables 1 and 2). The original optogenetic targeting system was designed by Christopher Fang-Yen, although the version that I constructed was modified to incorporate simultaneous two magnification imaging and I wrote custom software to run the system. Dr. Fang-Yen also helped with designing, troubleshooting, and interpreting experiments. I adapted the Infrared laser system previously constructed by Matthew Churgin (Churgin et al., 2013) to deliberately lesion tissue by rebuilding the beam expanding optic.
ABSTRACT

Coordinated rhythmic movements are ubiquitous in animal behavior. In many organisms, chains of neural oscillators underlie the generation of these rhythms. In *C. elegans*, locomotor wave generation has been poorly understood; in particular, it is unclear where in the circuit rhythms are generated, and whether there exists more than one such generator. We used optogenetic and ablation experiments to probe the nature of rhythm generation in the locomotor circuit. We found that multiple sections of forward locomotor circuitry are capable of independently generating rhythms. By perturbing different components of the motor circuit, we localize the source of secondary rhythms to cholinergic motor neurons in the midbody. Using rhythmic optogenetic perturbation we demonstrate bidirectional entrainment of oscillations between different body regions. These results show that, as in many other vertebrates and invertebrates, the *C. elegans* motor circuit contains multiple oscillators that coordinate activity to generate behavior.

INTRODUCTION

Oscillatory neural activity underlies rhythmic animal behaviors such as feeding and locomotion. Rhythm generating units are sometimes functional in isolated spinal cord and invertebrate nerve cord preparations, producing fictive rhythmic motor outputs that resemble *in vivo* patterns (Marder and Calabrese, 1996; Marder et al., 2005; Kiehn, 2006; Mullins et al., 2011; Grillner and El Manira, 2015). At the same time, sensory feedback and reflex loops have also been found to be important for motor rhythm
coordination and modulation (Wendler, 1974; Andersson et al., 1981; Yu and Friesen, 2004; Kristan et al., 2005).

How do cellular pacemakers, network oscillators, and sensory feedback interact to perform rhythmic motor generation and coordination? The identification and study of locomotor Central Pattern Generators (CPGs) in the mammalian spinal cord has been complicated by the system's complexity and the large numbers of neurons that are potentially involved. As a result, many components of the mammalian locomotor rhythm generator remain unidentified (Kiehn, 2006; Mullins et al., 2011; Kiehn, 2016). However, work on vertebrate and invertebrate models, such as swimming leeches and lampreys, has allowed the basic principles and components of neural oscillators to be identified (Goulding, 2009; Mullins et al., 2011).

Electrophysiological studies on the leech isolated ventral nerve cord (VNC) have found that individual ganglia distributed along the body can generate oscillatory patterns that mimic those of normal swimming (Weeks, 1981; Kristan et al., 2005; Marder et al., 2005). Stretch sensation and central control couple the oscillatory units in both the ascending and descending directions, such that the intact animal's entire circuit functions in synchrony during swimming (Mullins et al., 2011).

In the lamprey, excitatory interneurons proposed to be rhythm generators are also found throughout the approximately 100 spinal segments, which can generate oscillations when isolated (Mullins et al., 2011; Kiehn, 2016). The distributed nature of rhythm generation in swimming models bears some resemblance to that found in hindlimb locomotion in limbed vertebrates, for which rhythm generating capability is distributed along the caudal spinal cord (Kiehn, 2006). Moreover, analogues of many of the key
neuronal classes underlying these behaviors in lampreys and zebrafish are also found in the mouse spinal cord (Goulding, 2009; Kiehn, 2016).

Despite these findings, a clear understanding of how motor systems generate locomotory oscillations at the network, cellular, and molecular levels remains elusive. In both the leech and lamprey, identification of the neurons responsible for rhythm generation remains incomplete, and the mechanism(s) by which these neurons generate swim rhythms are unclear (Kristan et al., 2005; Mullins et al., 2011). In the lamprey, proposed oscillator neurons have not been directly shown to generate the swimming rhythm (Kiehn, 2016). Moreover, the paucity of genetic manipulations available in these organisms makes it difficult to describe molecular mechanisms that contribute to rhythm generation.

The roundworm *C. elegans* is a promising model for achieving an integrated behavioral, circuit, and molecular understanding of how locomotion is generated and coordinated. *C. elegans* has a compact nervous system containing a few hundred neurons, for which a nearly complete wiring diagram of synaptic connectivity has been mapped (White et al., 1986; Varshney et al., 2011). Worms’ optical transparency allows researchers to monitor neural activity with genetically encoded calcium and voltage sensors (Kerr et al., 2000; Kerr, 2006; Flytzanis et al., 2014), and manipulate neurons and muscles using optogenetics (Nagel et al., 2005; Zhang et al., 2007; Leifer et al., 2011; Stirman et al., 2011; Husson et al., 2012; Kocabas et al., 2012; Fang-Yen et al., 2015; Gao et al., 2015). *C. elegans* is readily amenable to a powerful set of genetic manipulations (Ahringer, 2006; Evans, 2006), and shares extensive genetic homology with humans (Lai et al., 2000). Classical neurotransmitters involved in *C. elegans* locomotion include
acetylcholine (Rand, 2007), GABA (Jorgensen, 2005), glutamate (Brockie and Maricq, 2006), and the biogenic amines dopamine and serotonin (Chase and Koelle, 2007).

*C. elegans* moves forward by generating sinusoidal dorso-ventral bending waves that propagate from anterior to posterior. The circuit for locomotion consists of interneurons, excitatory and inhibitory motor neurons, and body wall muscles (White et al., 1976; Chalfie et al., 1985; White et al., 1986; Altun and Hall, 2011). The majority of motor neuron cell bodies are located in the ventral nerve cord (VNC), which runs along the ventral side of the body from head to tail (White et al., 1986; Altun and Hall, 2011). The VNC motor neurons include A, B, VC, D, and AS cell types. Laser ablation studies have shown that the A-type neurons are essential for reverse locomotion, whereas the B-type are required for forward locomotion (Chalfie et al., 1985). The D-type (GABAergic) motor neurons are required for a normal amplitude of body bending waves but are not essential for locomotion itself (McIntire et al., 1993b). The function of the AS neurons is unknown. The VC neurons are involved in egg laying (Waggoner et al., 1998). These classes all form neuromuscular junctions with body wall muscles (BWMs).

While the basic architecture of the motor circuitry has been delineated by laser ablation studies, much less is understood about how its components interact to generate coordinated locomotory behavior. Perhaps most notably, it is not known which elements generate the worm’s dorso-ventral oscillations during forward movement, nor how many such rhythm generators may exist. Worms are capable of limited movement despite ablation of most premotor interneurons (Chalfie et al., 1985; Wicks and Rankin, 1995; Zheng et al., 1999). When all premotor interneurons are removed, animals did not generate directional movement, but retained the ability to generate local body bends (Kawano et al., 2011). However, forward locomotion was observed after ablation of all
premotor interneurons and A motor neurons (Gao et al; accompanying paper), suggesting that periodic bending during forward locomotion may be organized at the level of the non-A motor neurons and/or the body wall muscles.

Sensory feedback has been shown to play an important role in coordinating *C. elegans* motor behavior. The frequency of *C. elegans* undulation depends continuously on mechanical loading by its environment (Berri et al., 2009; Fang-Yen et al., 2010), and computational models based on proprioceptive feedback and coupling have recapitulated key aspects of locomotory behavior (Boyle et al., 2012; Wen et al., 2012). Experiments in which the worm’s body was partially immobilized in a microfluidic device showed that the posterior B-type motor neurons mediate anterior-to-posterior proprioceptive coupling (Wen et al., 2012). B-type motor neurons sense the body curvature and induce bending in the same direction (ventral or dorsal) posterior to the sensed bending.

These findings suggested a model for forward locomotion, similar to one proposed earlier (Karbowski et al., 2008), in which a single rhythm generator generates bending undulations in the head, and these undulations propagate through the body from anterior to posterior via proprioceptive coupling (Wen et al., 2012). This model successfully reproduced the continuous variation in locomotory characteristics observed in varied mechanical environments (Berri et al., 2009; Fang-Yen et al., 2010). This work, while demonstrating how a wave can be propagated along the body, did not directly address the identity of the rhythmic generator(s). Furthermore, it focused on coupling in the posterior of the worm and did not determine whether head and neck proprioception is similarly essential for bending wave propagation.
How might the locomotory circuit be organized? The circuit contains one or more oscillators (Figure 3.1). A model including a single oscillator with proprioceptive coupling (Figure 3.1C) predicts that a disruption in body bending at any location will inhibit posterior bending (Figure 3.1E). Alternative possibilities (Gjorgjieva et al., 2014; Zhen and Samuel, 2015) include the presence of multiple oscillators distributed along the motor circuit (Figure 3.1D), as in the vertebrate spinal cord (Kiehn, 2006; Mullins et al., 2011; Kiehn, 2016) and in the VNC of some invertebrates (Kristan et al., 2005). These oscillatory units could be capable of generating undulations in the posterior of the worm even if anterior neural activity or physical bending is interrupted (Figure 3.1F).
Figure 3.1. Overview of curvature analysis and models of rhythm generation

(A) Dark field image of worm shown with curvature segmentation. Dorsal bending is shown in blue and ventral bending in red. The dorso-ventral orientation is arbitrary unless otherwise specified. The worm’s centerline is used to define a coordinate system in which the head and tail are located at body coordinates 0 and 100, respectively. Scale bar: 200 μm.

(B) Curvature map from a normally swimming worm. The curvature at time t = 0 s corresponds to the image shown in (A).

(C) In a single-oscillator model of locomotion, an unknown oscillator causes rhythmic head bending, and a reflex-like coupling mechanism mediates propagation of these bends along the rest of the body.

(D) A multi-oscillator model (Gjorgjieva et al., 2014) posits the existence of additional circuit units outside the head capable of generating oscillations.

(E) Conceptual curvature map showing predicted worm behavior after paralyzing a small region of the body (dotted white box). The single-oscillator model predicts that all regions posterior to the paralyzed region will also become paralyzed.

(F) Conceptual curvature map predicting the outcome of the same manipulation applied to a multi-oscillator model. If additional oscillators exist posterior the paralyzed region, additional tail oscillations may arise, potentially with different amplitude, frequency, and/or phase.
In this work, we used spatiotemporally targeted optogenetic illumination (Leifer et al., 2011; Stirman et al., 2011) and lesion studies to show that the mid-body VNC motor circuit contains multiple units capable of independent oscillation. We found a fundamental architecture in the C. elegans motor circuit similar to that previously described in other vertebrate and invertebrate models.

RESULTS

Rhythmic posterior undulation persists despite anterior paralysis

We first sought to test a model in which there is a single oscillator in the head and proprioceptive feedback is the dominant organizer of bending waves along most of the body (Figure 3.1C). This model, supported by experiments showing that immobilization of the mid-body of worms induced the posterior to adopt the same direction of curvature as the immobilized region (Wen et al., 2012), predicts that paralysis of any region will eliminate undulations posterior to the paralyzed region (Figure 3.1E). In particular, we asked whether paralysis of the head and “neck” (a region immediately posterior to the head) would halt body bending posterior to these regions.

To manipulate neural and muscular activity in freely moving worms, we constructed an optogenetic targeting system similar to that previously described (Leifer et al., 2011). Briefly, this system uses real-time imaging processing and a digital micromirror device to project laser illumination onto arbitrarily specified regions of an unrestrained worm.
Figure 3.2. Anterior undulation is not required for posterior undulation

(A) Inhibition of anterior BWMs (via Pmyo-3::NpHR) increases tail frequency. Body coordinates 0-45 were illuminated with green light (532 nm wavelength) to trigger relaxation of the anterior muscles. The spatiotemporal extent of green laser illumination is indicated by the white dotted box.

(B) Inhibition of anterior cholinergic neurons (via Punc-17::NpHR; Punc-17::ChR2) does not prevent tail undulation. Body coordinates 0-33 were illuminated with green light to optogenetically inhibit anterior motor activity.

(C) Tail undulations persist despite paralysis of the anterior BWMs due to miniSOG-mediated lesion of muscle cells. Animals were subjected to mechanical stimulation to induce locomotion (see Methods). A total of 9 animals were illuminated with blue light (472 nm wavelength) in approximately their anterior halves. Of these, 5 displayed partial-body forward swimming as depicted here, 3 were immobile, and one was not sufficiently paralyzed in the head. Six control worms, which were mounted identically but not illuminated, all displayed waves propagating normally from head to tail (not shown).

(D) Inhibition of some anterior muscles (body coordinate 0-33, N=10 worms) significantly increases tail frequency. Inhibition of most anterior muscles (0-45, N=10 worms), or inhibition of anterior cholinergic neurons (N= 14 worms) produces mixed results; some animals generate high frequency tail oscillations while others slow down. Each colored circle represents one trial; worms may have multiple trials. Tail frequency is measured at body coordinate 85. Error boxes represent the mean and SEM.

(E) Amplitude of undulation in the head and tail before and during muscle or neuron inhibition. Head frequency is measured at body coordinate 15. Note sharp decreases in
head amplitude during all three manipulations. Amplitude here and henceforth is measured as the root mean square of the time derivative of the curvature times worm length \( \text{rms}(L \cdot \frac{dx}{dt}) \) and has units of s\(^{-1}\). (*) p<0.05; (**) p<0.01; (***) p<0.001; paired t-test.
To examine the effect of inhibiting anterior muscles, we first used this system to project 532 nm illumination onto worms expressing the inhibitory opsin halorhodopsin (NpHR/Halo) in all body wall muscles under the control of the \textit{myo-3} promoter (Zhang et al., 2007; Leifer et al., 2011). We quantified the movement of worms before and during optogenetic manipulation by measuring the curvature of the worm over time (Figure 3.1A, B) (Pierce-Shimomura et al., 2008; Fang-Yen et al., 2010; Leifer et al., 2011; Wen et al., 2012). We specify longitudinal positions via a body coordinate ranging from 0 at the tip of the head to 100 at the end of the tail.

Illuminating body coordinates 50-65 in \textit{Pmyo-3::NpHR} worms caused substantial paralysis in the tail (not shown), consistent with previous findings (Leifer et al., 2011; Wen et al., 2012). When we paralyzed the anterior 33\% or 45\% of the worm, however, we observed robust oscillations in posterior regions of the body. In addition, we found to our surprise that illumination of the anterior 33\% of the body caused the tail’s undulation frequency to increase (Figure 3.2A, D; Video 1).

Next, we asked whether oscillations in the posterior would persist under optogenetic inhibition of excitatory motor neurons instead of inhibition of muscles. We illuminated worms expressing NpHR in all cholinergic neurons (\textit{Punc-17::NpHR}), including the A-type and B-type motor neurons, head motor neurons, and several other neuronal cell types (Duerr et al., 2008). We found that while optogenetic inhibition of cholinergic neurons in the head and neck caused anterior paralysis, tail undulation often persisted (Figure 3.2B, D, Video 1).

During optogenetic muscle or neuron inactivation, the amplitude of the bending wave in the head decreased greatly but did not vanish (Figure 3.2A, B, E), leaving open the
possibility that a residual small amplitude wave allows propagation of the bending wave through the partially paralyzed region. We therefore sought means of paralyzing the head more effectively.

We hypothesized that regional paralysis could be induced by lesioning the anterior BWMs instead of hyperpolarizing them. To selectively lesion muscles, we used region-targeted illumination at 470 nm of *Pmyo-3::PH::miniSOG* worms in which the photosensitizing protein miniSOG is expressed in body wall muscles (Xu and Chisholm, 2016). The anterior portion of most treated animals was nearly immobile (Figure 3.2C, Video 1, especially the last 8 seconds). Nevertheless, undulation posterior to the region of illumination was routinely observed in these animals.

We also conducted thermal lesioning experiments in which touched the anterior half of the worm with a hot platinum wire attached to a soldering iron. After this treatment, the animal’s head and neck were again nearly motionless, yet rhythmic undulation routinely persisted in the tail (Figure 3.2 - Figure Supplement 1, Video 1).

Our finding that posterior undulation can persist despite anterior paralysis is consistent with a multi-oscillator model (Figure 3.1D) and not with a single oscillator model that relies on reflex-like signaling for wave propagation (Figure 3.1C).
Figure 3.2 Figure Supplement 1. Tail undulation after gross head lesioning

The approximate location of hot wire lesioning is indicated with a red arrow. Each panel presents data from a different worm. Scale bars: 200 μm.

(A) Slow rhythmic undulations are evident posterior to the head.

(B) Rhythmic undulations in the mid-body arise after substantial damage is applied to the head.
(C) Rhythmic undulations in the neck and mid-body arise after substantial damage is applied to the head.
The head and tail are capable of simultaneous oscillations at different frequencies

The finding that optogenetic inhibition of anterior muscles induces higher frequency oscillations in the tail suggests that an interruption of propagating activity in the motor circuit enables independent activity in a posterior oscillator. To test this idea further, we applied several optogenetic manipulations to inhibit motor coupling in the neck only, leaving the head and tail free to oscillate.

First, we optogenetically inhibited neck muscles in Pmyo-3::NpHR worms. In most trials, optogenetically inhibiting neck muscles prevented waves generated in the head from propagating through the neck. During the interruption of these waves the tail exhibited bending undulations at a higher frequency than that of the head, resulting in the animal simultaneously undulating at two distinct frequencies (Figure 3.3A, Video 2). We henceforth refer to this behavior, whether or not induced by any manipulation, as two-frequency undulation (2FU).

We observed 2FU upon inhibiting all neck cholinergic neurons (Figure 3.3B, Video 2) and also upon inhibiting neck B-type motor neurons (Pacr-5::Arch; Figure 3.3C, Video 2). These manipulations led to a large decrease in wave amplitude in the neck and a smaller decrease in wave amplitude in the tail (Figure 3.3F). Nevertheless, multiple animals in each experiment showed 2FU, with the highest ratios of tail frequency to head frequency seen in worms in which the neck muscles were inhibited (Figure 3.3E).

The bending amplitude of the tail generally decreased as the frequency increased (Figure 3.3E, F), consistent with the changes in bending frequency and amplitude previously observed when the viscosity of a fluid environment was varied (Fang-Yen et al., 2010). The opposite trends of amplitude and frequency may reflect a constraint to
the maximum absolute rate of change of curvature, which is proportional to the product of amplitude and frequency.

In some experiments, the optogenetic manipulation of motor neurons or muscles did not completely block wave transmission through the paralyzed region. In these trials, some tail waves appeared synchronized with head waves, whereas others did not (Figure 3.3C). To test whether 2FU can occur after stronger disruption of motor coupling, we lesioned mid-body muscles in Pmyo-3::PH::miniSOG worms. This manipulation indeed led to stronger decoupling between head and tail oscillations, but did still not prevent 2FU (Figure 3.3D, Video 2).

If the posterior motor circuit of B and AS type neurons contains additional oscillating units, we reasoned that localized undulations might occur after selectively activating small portions of the motor circuit while inhibiting the rest. We therefore examined worms in which both the inhibitory opsin NpHR and the excitatory opsin Channelrhodopsin-2 (ChR2) were expressed in the cholinergic neurons, after the A-type motor neurons were ablated by Punc-4::miniSOG.

We first illuminated these animals with 590 nm wavelength (yellow) light throughout the body to inhibit all cholinergic neurons. While maintaining this yellow illumination, we targeted small portions of the tail with 473 nm wavelength (blue) light, activating ChR2 and stimulating a few posterior B and AS neurons. Under these conditions, several animals generated high frequency localized undulations in the tail (Figure 3.3 - Figure Supplement 1A, Video 3). These findings further support the presence of additional oscillator(s) in this region.
If multiple independent oscillators underlie a worm’s forward movement under physiological conditions, we reasoned that independent head and tail oscillations might also be observable in animals without induced lesions or optogenetic perturbations.

Wave frequency depends strongly on the degree of mechanical loading from the environment, for example decreasing with viscosity of the fluid medium (Berri et al., 2009; Fang-Yen et al., 2010). We hypothesized that head and tail oscillations might be decoupled by placing the anterior and posterior of a worm in fluids of different viscosities. When we studied worms transitioning between regions of a low viscosity buffer into highly viscoelastic hydroxypropylmethylcellulose (HPMC) islands (see Methods), we observed 2FU in 6 of 41 worms (15%). In these animals, the tail continued oscillating at a high frequency for at least two full cycles even as the head frequency was sharply reduced (Figure 3.3 - Figure Supplement 1B, Video 3). Although these events were uncommon, they demonstrated that 2FU can occur in C. elegans with no internal perturbations.

Taken together, these results strongly suggest that the C. elegans forward motor circuit contains at least two units capable of independent rhythm generation, and that a partial breakdown in anterior proprioceptive coupling (for example by inhibiting neck BWMs) is sufficient to reveal the presence of the posterior oscillating unit(s).
Figure 3.3. Disruption of motor coupling in the neck de-synchronizes head and tail oscillations

(A) Inhibition of neck BWMs (via Pmyo-3::NpHR) increases tail frequency and decreases head frequency. We refer to this effect as two frequency undulation (2FU). Body coordinates 25-45 were illuminated with green light to induce relaxation of neck muscles. The spatiotemporal extent of green laser illumination is indicated by the white dotted box.

(B,C) Inhibition of neck cholinergic neurons (Punc-17::NpHR) or neck B-type motor neurons (Pacr-5::Arch) also induces 2FU behavior.

(D) Two frequency undulation after miniSOG-induced paralysis of the mid-body BWMs. Animal was subjected to mechanical stimulation to induce locomotion, but also displayed this behavior prior to stimulation. A total of 10 individuals were illuminated with blue light on approximately one-fifth of their body length, centered near the vulva. Of these, 7 displayed 2FU as depicted here, one was immobile, and two were not sufficiently paralyzed in the mid-body to disrupt bending waves. Color map data is scaled down by 50% because bends in this animal had higher amplitudes than those shown in A-C.

(E) Several optogenetic manipulations produced decoupled head and tail oscillation. 2FU is assayed by dividing tail frequency by head frequency in each worm. Before illumination, the head (body coordinate 15) and tail (body coordinate 85) usually oscillate at the same frequency. During illumination, tail frequency often exceeds head frequency. Each colored circle pair represents one trial; worms may have multiple trials. N= 11, 10, 12, and 10 worms per condition, respectively. Error boxes represent the mean and SEM.
(F) Amplitude of undulation in the neck and tail before and during neck muscle or neuron inhibition. Neck amplitude is measured at body coordinate 35. (*) p<0.05; (**) p<0.01; (***) p<0.001; paired t-test.
Figure 3.3 Figure Supplement 1. Additional disruptions to motor coupling cause 2FU

(A) Two examples of worms in which all cholinergic neurons are inhibited (Punc-17::NpHR; global yellow illumination) except for those within a small tail region (Punc-17::ChR2; blue illumination in a small tail region). The A-type motor neurons were killed at the L2 larval stage (Punc-4::MiniSOG). Green triangle: worm head; blue band: blue illumination region. The curvature map (lower pane) indicates the spatiotemporal...
windows of yellow illumination (yellow dotted box) and blue laser illumination (white dotted box). Scale bars: 200 μm.

(B) 2FU induced by an inhomogeneous mechanical environment. Red dotted line indicates the boundary between low viscosity buffer (NGM) and high viscosity HPMC. Around t = 11s, the tail continues oscillating at high frequency even as the head rapidly slows to a crawl inside the HPMC.
Most premotor interneurons are not essential for rudimentary forward movement or 2FU

To better understand the source of tail oscillations during 2FU, we used genetic analysis and lesion studies to ask which components of the motor circuit are required for this behavior. Almost all chemical or electrical synaptic connections to the VNC motor neurons are made by the premotor interneuron (IN) classes AVB, PVC, AVA, AVD, and AVE (White et al., 1986). Laser ablation studies have indicated that AVB, and to a lesser degree PVC, are essential for normal forward locomotion (Chalfie et al., 1985), although rudimentary forward crawling is possible in their absence if the reverse-driving A motor neurons are also removed (Gao et al., 2017). Therefore, we asked whether 2FU is possible in the absence of AVB, PVC, and all other premotor INs.

To determine if the premotor interneurons are required for 2FU, we first asked whether optogenetic muscle inhibition in the neck in worms lacking premotor interneurons would induce 2FU (c.f. Figure 3.3A, E). We used transgenic strains in which expression of the apoptosis-promoting interleukin-converting enzyme (ICE) was used to ablate premotor INs and some other neurons (Zheng et al., 1999). When ICE is expressed under the control of the nmr-1 or glr-1 promoters, the PVC, AVA, AVD, and AVE interneurons are removed. AVB, however, are present in both Pnmr-1::ICE (Kawano et al., 2011), and Pglr-1::ICE worms (Kawano, Po, and Zhen, personal communication). We generated the strains Pmyo-3::NpHR; Pglr-1::ICE and Pmyo-3::NpHR; Pnmr-1::ICE. We found that both strains were capable of 2FU during optogenetic inhibition of neck muscles (Figure 3.4A, E, Figure 3.4 Figure Supplement 1A, E). This result demonstrates that 2FU does not require most premotor interneurons, including the forward locomotory interneurons neurons PVC.
The interneuron AVB is coupled to the B-type motor neurons by an extensive network of gap junctions. Formation of these connections, as well as electrical coupling between the premotor interneurons and the motor neurons of the circuit for reverse locomotion, requires UNC-7 expression in AVB and UNC-9 expression in the B-type motor neurons (Starich et al., 2009; Kawano et al., 2011; Liu et al., 2017). UNC-9 also participates in electrical coupling between BWM cells (Liu et al., 2006). We asked whether animals lacking UNC-7 or UNC-9 could exhibit 2FU. We crossed unc-7 and unc-9 mutations into our Pmyo-3::NpHR strain and performed optogenetic experiments as before. The unc-7 and unc-9 worms are uncoordinated and exhibit significantly reduced spontaneous forward locomotion (Starich et al., 2009; Kawano et al., 2011). To initiate a short bout of forward locomotion, we used a cell phone motor to apply a mechanical stimulus in the form of a 3-5 s, ≈200 Hz vibration of the slide just before illumination. We found that both strains were capable of 2FU, although it appeared to occur less often than in PVC-ablated animals, and sometimes occurred prior to neck muscle inhibition (Figure 3.4B, E; Figure 3.4 Figure Supplement 1E; Figure 3.4 Figure Supplement 5A,B). This finding shows that the UNC-7/UNC-9 gap junctions, including those between AVB and the B-type motor neurons, are not required for 2FU.

Finally, we ablated AVB, labeled by Psra-11::D3cpv (Kawano et al., 2011), using a pulsed infrared laser ablation system (Churgin et al., 2013) that we modified to intentionally lesion tissue (see Methods). This procedure generally removed both AVB cell bodies and their associated processes, and possibly other head neurons, but not PVC. When subjected to the same experiment as described above, 2FU events were nearly diminished. Very rarely we observed uncoupled undulation events that were not correlated with neck muscle inhibition (Figure 3.4 Figure Supplements 1B, 1E, 2, 5C).
Taken together, these results suggest that most individual classes of premotor interneurons, including AVA, AVD, AVE, and PVC, are not essential for 2FU. However, AVB may play a key role in activating the rhythm generator(s) to allow oscillation.
Figure 3.4. VNC premotor interneurons, D-type motor neuron signaling, and individual B-type motor neurons are not required for 2FU

(A) 2FU occurs despite ablation of premotor interneurons (Pglr-1::ICE; Pmyo-3::NpHR). The spatiotemporal extent of green laser illumination is indicated by the white dotted box.

(B) 2FU occurs despite disruption of AVB:B gap junctions and BWM:BWM gap junctions (unc-9; Pmyo-3::NpHR)

(C) 2FU occurs despite ablation of a small subset of the B-type motor neurons (Pmyo-3::NpHR; Pacr-2::wCherry). VB8, VB9, and DB6 were ablated using our pulsed infrared laser system.

(D) 2FU occurs despite elimination of GABAergic signaling (unc-49; Pmyo-3::NpHR)

(E) When subjected to neck paralysis (Pmyo-3::NpHR; Pacr-2::wCherry), 2FU occurs reliably in Pglr-1::ICE animals and occasionally in unc-9 and unc-49 animals. Each colored circle pair represents one trial; worms may have multiple trials. N = 12, 8, and 10 worms per condition, respectively. Head frequencies are measured at body coordinate 15. Mid-body frequencies are measured at body coordinate 60. Error boxes represent the mean and SEM. (*) p<0.05; (**) p<0.01; (***) p<0.001; paired t-test.

(F) When subjected to neck paralysis (Pmyo-3::NpHR), 2FU occurs at least occasionally despite ablation of subsets of the B-type motor neurons by our pulsed infrared laser system. For each condition, data is only considered from worms that have all specified neurons missing; some worms in each group may have additional B-type or other neurons missing. N= 40, 30, 32, 27, 18, 18, and 16 trials from 10, 10, 10, 8, 7, 7, and 7 worms per condition, respectively. Mean mid-body/head frequency ratios during
illumination are significantly lower than mock controls for all ablation conditions except DB6, VB10 and VB11, DB7 (p<0.05 by one-way ANOVA with Bonferroni post-hoc comparisons).
Figure 3.4 Figure Supplement 1. VNC premotor interneurons and several VNC motor neuron classes are not required for de-synchronized oscillations.

(A) 2FU occurs despite ablation of the premotor interneurons AVA, AVD, AVE, and PVC, and other neurons (Pnmr-1::ICE; Pmyo-3::NpHR). The spatiotemporal extent of green laser illumination is indicated by the white dotted box.

(B) 2FU rarely occurs upon laser ablation of both AVB interneurons (See Figure 3.4 Figure Supplement 2).

(C) Robust 2FU occurs despite ablation of the A- and C-type motor neurons (Punc-4::MiniSOG; Pmyo-3::NpHR).

(D) C. elegans vab-7 mutants, in which DB motor neurons are broadly disrupted, have paralyzed tails and appear incapable of 2FU.

(E) When subjected to neck paralysis (Pmyo-3::NpHR), 2FU occurs reliably in Pnmr-1::ICE and Punc-4::MiniSOG animals, occasionally in unc-7 animals, and rarely animals in which AVB has been ablated. Each colored circle pair represents one trial; worms may have multiple trials (see methods). N= 11, 8, 14, 16, and 19 worms per condition, respectively. Error boxes represent mean and SEM. Head and mid-body data were measured at body coordinates 15 and 60, respectively.

(F) Neck paralysis leads to modest decreases in tail bending amplitude in most conditions.
Figure 3.4 Figure Supplement 2. Ablation of the AVB interneurons

(A) Mock control with both AVB neurons visible (Psra-11::D3cpv). Yellow outline, worm pharynx. Red arrows, AVB L/R in separate focal planes of the same worm. Yellow arrows, a pharyngeal neuron, possibly I4. AVB cell bodies were identified by their placement at the medial anterior edge of the terminal bulb of the pharynx, with processes directed ventrally. Scale bars: 50 μm.
(B) Worm in which both AVB interneurons were ablated at the L4 stage. Neither cell bodies nor processes are visible. Behavioral recording of this worm is shown in Figure 3.4 Figure Supplement 1.
Figure 3.4 Figure Supplement 3. Subsets of B-type motor neurons are not required for 2FU

Groups of 2-3 B-type motor neurons at a time were ablated using a pulsed infrared laser in worms expressing Pmyo-3::NpHR; Pacr-2::wCherry. See also Figure 3.4F.

(A) Curvature maps of worms that exhibited 2FU despite ablation of the indicated B-type motor neurons and potentially some other neurons.

(B) wCherry fluorescence images of each worm to confirm cell death. Blue labels indicate some or all B-type motor neurons. Red labels missing B-type neurons. White arrows indicate the vulva. Scale bars: 100 μm.
Figure 3.4 Figure Supplement 4. B-type motor neurons posterior to the vulva are not required for 2FU

We attempted to ablate all B-type motor neurons posterior to the vulva in worms expressing \textit{Pmyo-3::NpHR; Pacr-2::wCherry}.

(A) Curvature maps of worms that exhibited 2FU despite ablation of the indicated B-type motor neurons and potentially other neurons.

(B) wCherry fluorescence images of each worm to confirm cell death. Blue labels indicate some or all B-type motor neurons. Red labels indicate missing B-type neurons. White arrows indicate the vulva. Scale bars: 100 \textmu m.
In some disruptions to the motor circuit, 2FU was observed occurring without inhibition of neck muscles. Examples are shown here from an unc-9 mutant (A), an unc-7 mutant (B), an AVB-ablated worm (C), and an unc-49 mutant (D).
Several classes of motor neurons are not required for forward locomotion or 2FU

The premotor interneurons comprise the primary circuit connection between the VNC motor neurons and the worm’s other sensory and interneuronal circuits (White et al., 1986). The finding that most premotor interneurons are not necessary for forward locomotion and 2FU suggests that high frequency tail undulations during 2FU may originate from the motor neurons themselves. We asked, in the presence of all premotor interneurons, whether any classes of motor neurons are required for 2FU.

We first examined the A-type motor neurons. While the A class motor neurons are preferentially active during reverse locomotion (Haspel et al., 2010; Kawano et al., 2011), and are required for reverse locomotion (Chalfie et al., 1985), it is conceivable that they play a role in 2FU. We found that ablating the A- and VC-type motor neurons with a genetically targeted ROS generator (Punc-4::miniSOG) did not prevent 2FU induced by neck paralysis during forward locomotion (Figure 4 - Figure Supplement 1C, E). This result supports the idea that the A and VC motor neurons are not necessary for the posterior forward oscillator(s).

Next, we examined whether the GABAergic D-type motor neurons are required for 2FU. The D-type motor neurons release GABA onto the UNC-49 receptor to trigger contralateral muscle inhibition during a bend. Therefore, the putative null allele unc-49(e407) (Bamber et al., 1999; Liewald et al., 2008), should effectively block the functional output of D-type motor neurons. Indeed, unc-49 mutants exhibited simultaneous dorsal and ventral contractions when stimulated for forward and reversal movement, as did animals in which D motor neurons were ablated (McIntire et al., 1993a). We found that animals harboring an unc-49(e407) mutation, while very slow swimmers, were nonetheless capable of 2FU during neck muscle paralysis (Figure
3.4D, E). In one case we observed 2FU before inhibition of neck muscles (Figure 3.4 Figure Supplement 5D).

The B-type motor neurons are required for forward locomotion (Chalfie et al., 1985; Wen et al., 2012), and are rhythmically active during forward locomotion (Haspel et al., 2010; Kawano et al., 2011; Wen et al., 2012). We sought to determine whether any individual or small group of these neurons is essential for 2FU. We ablated groups of 2-6 B-type motor neurons at a time using our pulsed infrared laser system. 2FU was observed very rarely after ablating DB3, VB3, and VB4 or VB8 and VB9 (Figure 3.4C, F), although in both conditions there were additional instances of 2FU that occurred outside the time 3s window used for frequency analysis (Fouad et al., 2018a). 2FU occurred repeatedly, though not commonly, in all other conditions. (Figure 3.4F; Figure 3.4 Figure Supplements 3, 4). In nearly all cases, ablation of a DB motor neuron resulted in the disappearance of its commissural process, but we could not determine whether ablated VB neuronal processes were similarly removed. Therefore, we cannot completely exclude the possibility that a specific neuronal process can generate rhythms for 2FU in the absence of its associated cell body.

Taken together, these results suggest that in the presence of premotor interneurons, the B class motor neurons contribute to neck-paralysis induced 2FU, but no single member is essential for generating high frequency tail rhythms during 2FU. The A, VC and D motor neurons are not required for 2FU. The AS motor neurons were not investigated. Our results are consistent with a model in which posterior rhythm generation can arise from multiple subsets of B or AS motor neurons.
B-type motor neurons, as a class, are essential for 2FU

We next asked whether the B motor neurons as a class are required for 2FU. We first considered \textit{vab-7} mutants, in which the DB motor neurons have aberrantly reversed processes. These worms have disrupted wave propagation in the tail, which coils towards the ventral side (Wen et al., 2012). We found that \textit{vab-7} mutants had mildly or strongly paralyzed tails and were incapable of 2FU when neck muscles were inhibited, suggesting that \textit{vab-7} is essential for 2FU (Figure 3.4 Figure Supplement 1D, E).

The inability of \textit{vab-7} worms to generate the high frequency tail oscillation is consistent with the notion that broad disruption of the B motor neurons prevents 2FU. However, the behavioral deficit could also result from other effects of the mutation.

To ascertain whether the B motor neurons are required for 2FU, we performed broad ablations of the B motor neurons. We studied worms expressing \textit{Punc-17::PH::miniSOG} (Xu and Chisholm, 2016). In our integrated lines, we found that illumination of \textit{Punc-17::PH::miniSOG} worms preferentially eliminated the DA and DB over the VA and VB motor neurons (see Methods). Worms in which most DB motor neurons were eliminated were dramatically less likely to show 2FU than mock controls, but were not incapable of doing so (Figure 3.5, Video 4; N=9 out of 104 trials from 25 worms by blinded, randomized scoring). We performed the converse experiment – elimination of most VB motor neurons using our infrared laser system— and found the incidence of 2FU was again sharply reduced but not eliminated (Figure 3.5, Video 4, Figure 3.5 Figure Supplement 1). When we combined miniSOG and laser ablation to remove all DB and most VB motor neurons, animals were incapable of 2FU (Figure 3.5; 0 out of 102 trials from 27 worms by blinded, randomized scoring).
Taken together, these results suggest that the B motor neurons as a class are essential for independent tail undulation during forward movement. Our results are consistent with the hypothesis that the B motor neurons have a role in generating the high frequency locomotory rhythm observed in 2FU.
Ablated by *Punc-17β::PH::miniSOG* (DB) or IR Laser (VB):

Mock  DB (with 2FU - rare)  DB (no 2FU - typical)  DB and most VB

Curvature (kL)

Body coordinate

Fraction scored 2FU

Ablated:

Mock  DB only  VB only  DB+VB

Before illum.
During illum.

62/79
9/104 3/42 0/102
Figure 3.5. B motor neurons are required for 2FU

(A) Top panels: assessment of ablations using a Pacr-2::mCherry label. After removal of dorsal B motor neurons (Punc-17β::PH::miniSOG), pairs of motor neurons – each corresponding to one VA and one VB type neuron – are visible along the VNC (blue arrows). Neither the DNC (red arrows) nor the DB or DA commissures (red arrowheads in mock) are visible. Scale bars: 100 μm.

Bottom panels: Corresponding examples of worm locomotion after removal of DB (via miniSOG) and VB (via IR laser) motor neurons. Removal of DB always resulted in tail paralysis in a coiled position, but a minority of worms were able to generate a rhythmic midbody wave. Additional removal of VB motor neurons by laser ablation completely prevented 2FU and resulted in severe tail paralysis. (B,C) 2FU, as assayed by frequency measurements (B) or blinded, randomized scoring (C), is sharply reduced or eliminated by removal of DB, VB, or both. Head and mid-body frequencies were measured at body coordinates 15 and 60, respectively. Error boxes in (B) are the mean and 95% confidence interval of the mean. Error bars in (C) are standard error of the sample proportion. Numbers above each bar in (C) indicate the number of trials scored 2FU over the total number of trials for the condition; each individual worm contributed between 1 and 5 trials (3.6 on average). (*** p<0.001; paired t-test.)
Figure 3.5 Figure Supplement 1. Behavior after ablation of VB motor neurons

Example curvature maps from trials in which 2FU was (A) and was not (B) observed after IR laser ablation of VB motor neurons. VB3-VB11 were removed in both worms.
Multiple lesion-separated VNC segments are capable of independent rhythm generation

The observation that 2FU could persist despite disruptions to many components of the mid-body motor circuitry could also be explained by the hypothesis that additional rhythm generators located in the head are responsible for the observed high frequency tail undulations. This possibility is supported by the observations that premotor INs account for the majority of synaptic inputs to the VNC motor neurons (White et al., 1986), and removal of the AVB premotor interneurons nearly abolished 2FU (Figure 3.4 Figure Supplement 1E).

To ascertain whether the mid-body motor circuit is capable of independent rhythm generation, we developed a method for eliminating synaptic connections between the mid-body motor neurons and the head circuits. We used our infrared laser system to sever both the VNC and the dorsal nerve cord (DNC) just posterior to the pharynx. In many cases, this procedure also severed other fasciculated process bundles that run parallel to the VNC and DNC (Figure 3.6A).

Several hours after disruption of the nerve cords, most animals were inactive (data not shown), but active forward locomotion was induced by application of a mechanical stimulus. Most of these worms could generate robust oscillations posterior to the cut location (Figure 3.6A, E, Video 5). Moreover, the tail often undulated at a higher frequency than the mid-body (Figure 3.6E). In these worms, oscillations in the head were highly disrupted. In some cases, low amplitude waves propagating in the posterior-to-anterior direction occurred simultaneously with robust mid-body and tail waves propagating in the anterior-to-posterior direction (Figure 3.6A, Video 5), suggesting a deficit in coordination between circuits on either side of the lesion. These results suggest
that synaptic connections from the head circuits to the motor neurons may not be essential for wave generation posterior to the head.

We considered the possibility that under these conditions, mid-body undulations were being caused by small movements in the head rather than generated by a second oscillator. To minimize the small movements of the head, we introduced an additional manipulation to reduce head movement. Using our infrared laser, we applied thermal damage to the worm’s nerve ring (located in the head) in addition to cutting both nerve cords. Animals treated with these three lesions are henceforth referred to as “VNC-lesioned” worms. These worms exhibited very little movement in the head. However, they continued to generate robust oscillations in the mid-body and even higher frequency oscillations in the tail (Figure 3.6B, E, Video 5). The pattern of locomotion in VNC lesioned animals strongly resembled the 2FU induced by optogenetic perturbation, with the important difference that in our lesioned preparation, both frequencies were likely generated outside the head.

The emergence of multiple frequencies of undulation outside the head suggests that the VNC motor circuit itself may contain multiple units capable of independent oscillation. These units may exist in addition to any oscillating unit(s) in the head. To test this possibility directly, we cut the VNC and DNC in two locations: in the neck (anterior to VB3) and in the tail (posterior to VB8). We again thermally lesioned the head neurons to suppress head movement. Under these conditions, the VNC motor neurons between VB3 and VB8 are isolated from circuitry in both the head and tail, and the VNC motor neurons posterior to VB8 are isolated from both the head circuits and the anterior VNC motor neurons. As before, these animals could generate robust body oscillations posterior to the first cut and higher frequency oscillations posterior to the second cut
(Figure 3.6C, E, Video 5), suggesting that rhythms can arise independently from each of these portions of the VNC motor neurons. It should be noted that in all of our VNC lesion studies, the processes of premotor interneurons AVA, AVB, and PVC likely remained present in the VNC.
Figure 3.6. Undulations generated in the tail after severing the dorsal and ventral nerve cords

(A) The VNC (blue arrow) and DNC (red arrow) were severed in a *Punc-17::GFP* worm using a pulsed infrared laser. Several other dorso-ventral processes also appear cut. Nonetheless, robust bending waves are generated in the mid-body (lower pane). All scale bars: 50 μm.

(B) The VNC and DNC are severed, and arbitrary damage has been applied to the nerve ring to suppress head movements. Robust bending waves are generated in both the neck and tail (lower pane).

(C) The VNC and DNC are severed posterior to the head and vulva (*Pacr-2::wCherry*), and the nerve ring is lesioned to suppress head movements. Robust bending waves are generated in both the neck and tail (lower pane).

(D) The VNC and DNC are severed posterior to the head and vulva, but the nerve ring was not targeted. Low frequency head undulation and high frequency tail undulation were observed separately in this animal.

(E) Frequency of undulation in the mid-body and tail for all ablation conditions and mock controls. Each colored circle pair represents one bout of forward locomotion lasting at least 2 s. For each condition, data is only considered for worms in which the VNC and DNC are clearly severed in the indicated locations. Mid-body and tail frequencies were measured at body coordinates 45 and 85, respectively. Error boxes represent the mean and SEM. *N* = 3, 7, 3, 4, 5, and 7 worms per condition, respectively. (*) *p*<0.05; (**) *p*<0.01; (***) *p*<0.001; paired t-test.
B class motor neurons are necessary for rhythmic wave generation in VNC-lesioned worms

We next asked which motor neuron groups contributed to rhythm generation in worms in which the VNC and DNC were severed. The A motor neurons are not necessary for 2FU induced by neck paralysis (Figure 3.4 - Figure Supplement 1C). We hypothesized that they are similarly not required for body oscillation in VNC-isolated worms. We severed the VNC and DNC in either one or two locations after ablating the A motor neurons with Punc-4::miniSOG. Animals in which the VNC and DNC had been severed near the head were capable of robust wave generation and propagation posterior to the head (Figure 3.6 - Figure Supplement 1B). Animals in which the VNC and DNC were severed in two locations were capable of rhythmic activity in the mid-body or tail, although we did not observe any cases of simultaneous oscillation in each segment (N=20 worms, Figure 3.6 - Figure Supplement 1C). These results support the idea that the A motor neurons are not required for generation of rhythmic forward waves in surgically isolated segments of the VNC motor circuit.

The B motor neurons were required for neck-paralysis-induced 2FU in intact animals (Figure 3.5). We hypothesized that they are also required for forward undulatory rhythms in VNC-lesioned worms. To test this idea, we used VNC-lesioned worms in which the DA and DB motor neurons were ablated by miniSOG and the VB motor neurons were ablated by an infrared laser as before. Three to five hours after nerve cord surgery, worms in either B-ablation condition, like mock controls, were highly inactive. Application of a mechanical stimulus caused tight coiling, especially but not exclusively in the DB-only ablation condition (Figure 3.6 Figure Supplement 2C, Video 6). Despite this coiling, we observed at least one case of a VNC-lesioned, DB-ablated
worm appearing to move the very tip of its tail, possibly in a rhythmic fashion, suggesting that DB neurons may not be required for rudimentary oscillation in at least the most posterior portion of the tail. However, in VNC-isolated worms for which most DB and VB neurons were ablated, we never observed rhythmic movements (Figure 3.6 Figure Supplement 2D-F; N=19 mechanical stimulus trials from 9 worms). These results suggest that, as was the case for optogenetic 2FU, the B motor neurons are required for rhythm generation in VNC-lesioned worms.

The observation that anterior VNC/DNC cuts disrupt normal head undulation (Figure 3.6A) suggests that rhythm generation by the head circuit may require inputs from the VNC. To address this possibility, we studied additional worms in which the VNC/DNC were cut in 2 locations slightly more posterior to the head to reduce the likelihood of damage to the head motor neurons, but which were not subject to thermal damage to the head. As in our earlier experiment (Figure 3.6A), head movement was severely disrupted (not shown). However, we occasionally observed very slow head undulation in these animals, indicating that head undulation is still possible under these conditions (Figure 3.6D, Video 5).

One explanation for the low frequency of head undulations is that damage to the SMB or SMD neurons in the parallel tracts (Figure 3.6A) hampered head movement. Another possibility is that input from the posterior motor circuit is essential for the normal frequency of head undulation. The first hypothesis may be supported by our earlier observation that strong decoupling of head and tail oscillations by muscle ablation (i.e. without lesioning the nerve cord or parallel tracts) resulted in similarly slow head oscillations (Figure 3.3D), while weak decoupling resulted in moderately slower head
oscillations (Figure 3.3A). The possibility of posterior-to-anterior communication is discussed in detail in later sections.

These results suggest that the C. elegans forward locomotor circuitry possesses at least three units capable of independent oscillation, with two units located outside the head. Although we did not observe simultaneous 3-frequency forward undulation in any animal tested, we have shown that oscillations are possible in each segment when isolated from the others.
Ablated A, cut VNC & DNC anterior to VB3, and damaged head
Punc-17::GFP; Punc-4::miniSOG
Figure 3.6 Figure Supplement 1. Body undulations after severing the VNC and DNC do not require the A motor neurons

(A) Representative images of the VNC motor neurons in adult worms without (i) and with (ii) illumination with blue light at the L2 larval stage. After illumination, only B and AS neurons were visible along most of the body. In all worms tested however (N=20), at least one A motor neuron was visible at the posterior end of the VNC, corresponding to either VA12, DA8, or DA9. The behavior of the worm in (ii) is shown in C(i). Scale bars: 100 μm.

(B) Rhythmic waves visible posterior to the head after removal of the A motor neurons, severing the VNC and DNC in the neck, and damaging the head.

(C) Rhythmic waves visible in the mid-body (i) and tail (ii) after removal of the A motor neurons, severing the VNC and DNC in the neck, severing the VNC and DNC posterior to the vulva, and damaging the head.
**A**

![Image A]

**B**

Mock control
Punc-17β::PH::MiniSOG

C

Ablated DB
Punc-17β::PH::MiniSOG

D

Ablated DB & VB
Punc-17β::PH::MiniSOG,
IR Laser

Cut VNC & DNC posterior to pharynx, and damaged head

**E**

Frequency (Hz)

Ablated: Mock DB DB & VB

**F**

Fraction scored oscillating

Mock DB DB & VB

28/32
1/8
0/19
Figure 3.6 Figure Supplement 2. B motor neurons are required for body undulations after severing the VNC and DNC

(A) Fluorescence image of the worm in (D), for which DB have been removed by MiniSOG and VB have been removed by laser ablation. The remaining VA neurons and processes allow visualization of the VNC, including the cut posterior to the pharynx (blue arrow). DA and DB commissures are not visible. Scale bar: 100 μm.

(B-D) behavior during periods of mechanical vibration in worms for which the VNC and DNC were severed posterior to the pharynx. Control worms exhibit robust undulation. Worms lacking DB nearly always coiled in response to the stimulus, but in a minority of cases, very faint movements, potentially oscillations, could be seen in the tip of the tail. Worms lacking DB and VB also reacted to the stimulus by at least partially coiling, but rhythmic body undulations were not observed.

(E) Recorded frequency during bouts of well-segmented non-reversal lasting at least two seconds during periods of mechanical stimulation. Control worms routinely exhibited oscillations in the mid-body and tail. DB ablated worms had no mid-body oscillations, but in one case produced very faint high frequency tail oscillations (see C). DB and VB ablated worms lacked any rhythmic oscillations. N = 58, 7, and 16 bouts from 15, 4, and 9 worms, respectively Mid-body and tail frequency are measured at body coordinates 45 and 85, respectively.

(F) Subjective judgement worm behavior in each condition. Fractions indicate the number of trials blindly scored as oscillating during 20 s periods of mechanical stimulus (anywhere along the body) over the total number of trials. Oscillation was also not
observed in DB or DB & VB ablated worms outside periods of stimulus (not shown).

Each animal was subjected to 1-2 such stimulus trials. Error bars represent standard error of the sample proportion.
Undulations can arise in arbitrary portions of the VNC motor circuit

We next sought to identify the smallest portion of the VNC motor circuit that is capable of generating rhythmic behaviors, and whether any differences exist between the rhythmic properties of segments of various sizes. To address these questions, we again damaged the head (as in Figure 3.6B), then systematically varied the location of VNC/DNC lesions relative to the B motor neurons.

When we severed the VNC and DNC at any one of a number of different locations (anterior to VB3, VB5, VB6, VB7, VB9, VB10, and VB11), we found that undulations posterior to the damage arose near each cut location (Figure 3.7B-E). Bending amplitude similarly recovered just posterior to each cut location, rather than at a fixed body coordinate (Figure 3.7F). Oscillations in the tail usually had a higher frequency when the nerve cords were severed in the tail than when they were severed near the head (Figure 3.7G). We did not observe any bouts of locomotion with anterior-to-posterior waves posterior to the lesions when we severed the nerve cords anterior to VB11 (not shown), and bouts detected after lesioning at VB10 had waves of very low amplitude (Figure 3.7F), without clear rhythmicity (not shown). The smallest VNC segment that produced clear and robust rhythmic waves was the region between VB9 and the tail. These results suggest that the most compact rhythm-generating unit of the forward motor circuit is at least as small as the region containing VB9, VB10, VB11, DB6, and DB7.
Figure 3.7. Undulations are generated after VNC/DNC lesioning in arbitrary locations

(A) Schematic indicating all regions at which we severed the VNC and DNC in relation to the B motor neurons. Each animal’s nerve cords were severed at only one of these locations. The nerve ring of each worm was also damaged to restrict head movements as in Figure 3.5B.

(B-E) Representative curvature maps for worms subject to four of the tested conditions. Note that anterior-to-posterior waves begin progressively more posterior for each cut location. In some cases, the head and tail exhibited waves propagating in opposite directions (D).

(F) Amplitude of bending as a function of body coordinate after severing the VNC and DNC anterior to the indicated motor neuron. Only the portion of the curve posterior to the amplitude minima (the cut location) is shown. No bouts of locomotion with anterior-to-posterior waves were discernible posterior to the cut at VB11 either subjectively or by our analysis software. For cut locations at VB3, VB5, VB6, VB7, VB9, VB10, and VB11 we studied N = 15, 9, 6, 10, 14, 9, and 6 worms and observed 25, 16, 7, 12, 30, 19, and 14 bouts of forward locomotion (lasting at least 3 s), respectively. Shaded outline represents ±SEM.

(G) Frequency of undulation at body coordinate 75 for four cut conditions. Boxes represent mean and SEM. Each colored circle indicates the frequency during one bout of forward locomotion. *p<0.05, one-way ANOVA with Bonferroni post-hoc comparisons.
Rhythmic motor entrainment is possible in both the anteriorward and posteriorward directions

Although we have shown that the *C. elegans* forward motor circuitry contains multiple rhythm generating units, it remains unclear exactly how these oscillators are coupled together, or even if they are all active during normal locomotion. Previous work demonstrated proprioceptive coupling in the posterior direction (Wen et al., 2012), and we showed that a disruption to proprioceptive coupling, via optogenetic inhibition of neck muscles, could decouple body undulations from head movements (Figures 3.3A, 3.4).

One surprising feature of our results was that paralyzing the neck muscles appeared to decrease the head frequency. We found that during 2FU, head frequency decreases relative to the unperturbed swimming frequency (Figure 3.8A). Slowing was often even more dramatic when decoupling was stronger (Figures 3.3D, 3.6D). These observations suggest that anteriorward coupling, in addition to posteriorward coupling, may be present in the forward locomotor circuitry.

To test for anteriorward coupling between motor circuit elements, we asked whether an oscillating optogenetic perturbation in the mid-body can entrain the head to a new frequency. We used our optogenetic targeting system to rhythmically inhibit the mid-body BWMs (Figure 3.8B). Worms subject to this procedure exhibit a head bending frequency approximately one half that of the imposed frequency. The factor of one half is likely due to the presence of two phases during the rhythmic locomotory cycle of any single part of the body during which the curvature is close to zero (i.e. muscles are relaxed). In some cases, small head bends corresponding to individual mid-body pulses were evident as well (Figure 3.8C(i), Video 7).
We found the head frequency could be entrained to a range of imposed mid-body frequencies. Subjecting body coordinates 33-66 to pulsed illumination at frequencies from approximately 1 to 2 Hz caused an increase in the power spectrum of the worm’s oscillations at frequencies corresponding to half of the imposed frequency, and a decrease at other frequencies. Pulsing at frequencies below 1 Hz generally led to head oscillations near the imposed frequency (Figure 3.8C(ii)). These results show that a mid-body rhythmic signal can entrain head bending, and point to the presence of an anteriorward coupling mechanism within the motor circuit.

We asked whether the anteriorward coupling occurs via the VNC cholinergic neurons, which are electrically coupled to each other, the muscle-to-muscle gap junctions, or through another mechanism. We applied a rhythmic optogenetic inhibition pattern to the midbody cholinergic neurons in Punc-17::NpHR worms. Once again, worms subject to this procedure quickly adjust their head bending frequency to match one-half of the imposed frequency (Figure 3.8D(i), Video 7). Moreover, rhythmic illumination of the tail cholinergic neurons at 2 Hz similarly increased the magnitude of head bending at 1 Hz (Figure 3.8D(ii)). Selective rhythmic hyperpolarization of the mid-body B motor neurons also sufficed to increase the magnitude of head bending at one-half of the stimulus frequency (Figure 3.8 - Figure Supplement 1A), as did rhythmic hyperpolarization of the BWMs when muscle-to-muscle gap junctions were disrupted in only in the BWM by a mutation in the innexin unc-9 that was rescued in neurons but not muscles (Wen et al., 2012) (Figure 3.8 - Figure Supplement 1B). These observations suggest that the posterior to anterior coupling is mediated neuronally, possibly by the VNC motor neurons.
Figure 3.8. Head undulation frequency can be entrained by mid-body optogenetic manipulation

(A) Neck muscle hyperpolarization (Figure 3.3A) causes a significant decrease in head bending frequency. This decrease is not predicted by either model discussed in Figure 3.1.

(B) A multi-oscillator model of forward locomotion allowing for motor coupling in both the anterior and posterior directions. To test this model, we sought to impose a new frequency on the mid-body of a freely moving worm, and test whether head bending also adopts the new frequency.

(C) Head bending frequency can be entrained by rhythmically inhibiting the mid-body BWMs.

(i) A curvature map showing a representative trial. Green light was pulsed on coordinates 33-66 at a frequency of 2 Hz onto a Pmyo-3::NpHR worm. Note that the head frequency slows to half of the imposed frequency, although some instances of a 1:1 correlation between laser pulse and a head bend are also evident (e.g. around t = 13 s).

(ii) Mean head frequency power spectra of Pmyo-3::NpHR worms before manipulation (left bar, worms from all conditions are pooled) and while subject rhythmic mid-body paralysis. Frequencies tested were 0 (with laser on), 0.5, 0.85, 1.1, 1.25, 1.4, 1.55, 1.7, 1.9, and 2.0 Hz. Frequency data is interpolated between these points. N≥11 trials per condition, with each worm supplying at most two trials. For high frequency inhibition (f>1.1 Hz) the head is entrained to half of the inhibition frequency (bright peaks lie along
\( y = \frac{x}{2} \). For lower frequencies of inhibition (\( f \sim 0.85 \text{ Hz} \)) the head is entrained to the inhibition frequency (bright peaks lie along \( y = x \)).

(D) Head bending frequency can be entrained by rhythmically inhibiting the head, mid-body, or tail cholinergic neurons. (i) A curvature map showing a representative trial. Green light was pulsed on coordinates 33-66 at a frequency of 2 Hz onto a \textit{Punc-17::NpHR} worm.

(ii) Mean head frequency spectra before manipulation (black, all conditions pooled), and after rhythmically inhibiting the head (blue, body coordinates 0-33), mid-body (orange, 33-66), or tail (yellow, 66-99) neurons at 2 Hz. Rhythmic inhibition of the mid-body or tail increases the frequency power at 1 Hz and decreases the power at the original undulation frequency, mirroring (C). \( N \geq 16 \) trials per condition, with each worm supplying at most two trials. Shaded outlines are the SEM. Vertical red lines indicate the imposed frequency (solid) or one-half of the imposed frequency (dashed).
Figure 3.8 Figure Supplement 1. Rhythmic activity in the mid-body B motor neurons is sufficient for posterior-to-anterior entrainment, and muscle-to-muscle gap junctions are not required.

(A) Representative kymogram from a Pacr-5::Arch worm that was subject to rhythmic (1.7 Hz) hyperpolarization of the mid-body B motor neurons (top), and average frequency spectra of the head before and during rhythmic inhibition (bottom). Note an increase in amplitude at 0.85Hz, one-half of the imposed frequency. N=22 trials were
analyzed from 11 worms. Vertical red lines indicate the imposed mid-body frequency (solid) or one-half of the imposed frequency (dashed).

(B) Equivalent analysis for worms in which the mid-body muscles were rhythmically inhibited and muscle-to-muscle gap junctions are disrupted. Expression of UNC-9 in neurons was transgenically restored (Wen et al., 2012). Note an increase in amplitude at 0.85 Hz, one half of the imposed frequency. N=22 trials were analyzed from 11 worms.
DISCUSSION
In zebrafish and lampreys, rhythmogenic capability for swimming undulations is distributed along the rostro-caudal axis of the spinal cord (Kiehn, 2006; Mullins et al., 2011). When isolated from the rest of the cord, groups of lamprey spinal segments do not exhibit identical preferred frequencies (Cohen, 1987). In the swimming intact animal, oscillations in all segments are phase and frequency-locked by intersegmental coupling that spans broad swaths of the spine (Mullins et al., 2011).

The motor system of the leech, an invertebrate, also shows a distributed rhythm generating architecture. Individual ganglia of the leech VNC can generate crude bursting patterns that resemble their firing patterns during swimming. The most robustly oscillating ganglia are towards the middle of the leech’s body, and isolated midbody ganglia also have a higher frequency than isolated ganglia near either the head or tail. In the intact animal, extensive, bidirectional intersegmental coupling drives the system to adopt a single locomotor frequency (Pearce and Friesen, 1985; Kristan et al., 2005).

Our results reveal a picture of forward locomotor control in C. elegans similar to that found in lamprey and leech. We found that rhythmogenic capability in the worm is distributed along the VNC motor circuit. As in other swimming models, the rhythm-generating capability of posterior circuits is only detectable when coupling is disrupted. The rhythm-generating capability of posterior circuits was demonstrated in several ways: optogenetic inhibition of anterior neurons, optogenetic inhibition of anterior muscles, an inhomogeneous mechanical environment, or a lesion to the nerve cords. The incomplete nature of our optogenetic decoupling method yielded evidence that an anterior rhythm can entrain the higher-frequency posterior rhythms. For example, we found that during
2FU, some but not all waves in the tail were continuous with waves in the head (Figure 3.3B, C). Even in these cases, the difference in locomotory frequency between the two body regions is inconsistent with single oscillator models.

We found that neither the head nor tail frequency during 2FU matched the natural (unperturbed) frequency of locomotion. Instead, the normal locomotory frequency in the environment of the assay was generally intermediate between the head and tail frequencies exhibited during 2FU. This finding shows another key similarity with models of swimming in other vertebrates and invertebrates: during locomotion, multiple rhythm generating units, each with different rhythmic properties, are combined by strong inter-unit coupling to form one functional unit (Friesen and Hocker, 2001; Kristan et al., 2005; Kiehn, 2006; Mullins et al., 2011). Such a whole-body oscillating unit will generally have rhythmic properties different from that of any subunit in isolation. Indeed, modeling studies of the leech swim CPG have suggested that the overall fictive locomotor frequency of a 17-ganglion portion of the VNC lies within the range of frequencies of the individual ganglia (Zheng et al., 2007). The gradient in intrinsic frequencies appears to set the wavelength during fictive swimming. The variation in rhythmic properties between different parts of the body may play a similar role in C. elegans.

During forward locomotion, the C. elegans motor circuit is coupled by posteriorward proprioception (Wen et al., 2012), and additional anteriorward and posteriorward coupling mechanisms (Figure 3.8). This bidirectional coupling appears to allow the entire circuit to operate as one unit. This picture is similar to descriptions of leech and lamprey motor circuits. In the leech, oscillatory interneurons that comprise the swim CPG send axons along the anterior-posterior axis of the animal, and mediate bidirectional coupling between midbody ganglia (Friesen and Hocker, 2001; Kristan et
The coupling between ganglia appears to result from a mixture of proprioceptive feedback and central control (Yu et al., 1999; Kristan et al., 2005). Injection of sinusoidally varying current into the leech stretch receptor cells was sufficient to entrain swimming activity (Yu and Friesen, 2004) at the injection frequency. Imposed rhythmic movements are also capable of entraining fictive swimming in isolated preparations of the lamprey spinal cord (Grillner et al., 1981; McClellan and Sigvardt, 1988) and several other vertebrate and invertebrate systems (Wendler, 1974; Andersson et al., 1981; Robertson and Pearson, 1983).

Our results include several lines of evidence implicating the cholinergic B and AS motor neurons as a likely source of posterior rhythmogenesis during C. elegans forward locomotion. First, disruption or removal of most or all B motor neurons virtually abolished independent tail undulations induced either optogenetically or by severing the nerve cords, but elimination of other motor neurons and premotor interneurons did not eliminate the ability of the posterior to oscillate independently (Figures 3.4, 3.5, and 3.6). Second, stimulation of select B and AS neurons after paralyzing most of the body led to local, high frequency oscillations in the tail that mimicked 2FU (Figure 3.3 Figure Supplement 1). Third, imposing a rhythmic signal on mid-body B motor neurons sufficed to entrain whole-body locomotor frequency, as did imposing rhythms on the broader class of cholinergic neurons and the body wall muscles; changes in body posture are sensed by the B motor neurons (Wen et al., 2012), providing a likely explanation for how this manipulation was able to entrain locomotion. The latter argument is somewhat comparable to one of the key criteria used to classify leech interneurons as part of the swim CPG; that injection of a pulse of current resets the phase of the locomotor rhythm (Mullins et al., 2011). In our experiments, the swimming frequency, and thus phase, was
reset almost immediately after the first inhibitory stimulus, suggesting that the B motor neurons share the phase shifting property with oscillatory interneurons in the leech. Lastly, the B motor neurons are rhythmically active in phase with locomotion (Kawano et al., 2011; Wen et al., 2012), satisfying another key criterion set for candidate oscillator neurons in the leech (Mullins et al., 2011).

Descriptions of swimming CPGs in vertebrates have tended to exclude motor neurons as members of the CPG, with most oscillatory function attributed to interneurons (Kiehn, 2016). Leech excitatory motor neurons have not been shown to be members of the swim CPG, as injection of a pulse of current fails to reset the cycle phase (Kristan et al., 2005). However, leech inhibitory motor neurons are ascribed a role in swimming rhythm generation using the phase-resetting criterion (Mullins et al., 2011), motor neurons comprise the crab stomatogastric CPG (Marder and Bucher, 2007), and recent work has shown that motor neurons are key components of several locomotory pattern generators. In leech crawling, which consists of cyclic elongation and contraction phases, current pulses to the CV elongation motor neurons do indeed reset the phase of fictive crawling. However, the CV neurons were not concluded to be necessary components of the CPG because tonic hyperpolarization failed to abolish the crawling rhythm. Anatomical removal, arguably the more relevant test of necessity, was not reported (Rotstein et al., 2017). In zebrafish, motor neurons for swimming are bidirectionally coupled to locomotion-driving interneurons by gap junctions, and influence their recruitment, synaptic transmission, and firing frequency during locomotion (Song et al., 2016). Hence, there is a growing recognition that motor neurons are not limited to conveying oscillatory signals from interneurons, but may themselves participate in rhythm generation.
One difference between our results and previous findings in leeches is in the effect of severing the VNC. When we severed the VNC and DNC of *C. elegans*, we found that independent, generally higher frequency undulations occurred posterior to the severed region (Figures 3.6 and 3.7). Disruption of the leech VNC, by contrast, was not sufficient to prevent wave propagation from head to tail (Yu et al., 1999), suggesting that proprioceptive information suffices to propagate the wave. However, severing the VNC intersegmental coordinating neurons in *in vitro* preparations induced uncoordinated fictive swim oscillations at different frequencies occurring on either side of the cut (Weeks, 1981). This difference in results may arise due to differences between our thermal ablation method in *C. elegans* and physical severing of the leech VNC, or the relative span of proprioceptive signals in each system.

Our laser lesioning of the VNC likely did not remove the severed processes of premotor interneurons, nor did it prevent nonsynaptic neurotransmission, for example through neuropeptides, from potentially regulating rhythm generation across the lesion. These possibilities may account for the apparent difference in posterior rhythmogenic capability between worms in which AVB had been ablated versus severed. When the ventral nerve cord was isolated from the head ganglia, including the soma of AVB, rhythmic tail undulation was reliably evoked by a mechanical stimulus (Figures 3.6 and 3.7). However, independent tail undulations were observed only rarely after ablating AVB, even when the mechanical stimulus was applied (Figure 3.4 Figure Supplement 1). The AVB process that likely remain in the severed VNC segments may continue to provide excitation to the B motor neurons to promote rhythm generation. The other apparent discrepancy, between disrupting AVB-B gap junctions by *unc-7* or *unc-9* genetic mutations and ablating AVB, may be a consequence of other effects of the
mutation, including changes in other connections between the premotor interneurons and the B motor neurons (Starich et al., 2009), that may compensate for the loss of AVB:B gap junctions to activate the forward circuit. In any case, the observation that tail undulation was reliably evoked after eliminating all synaptic inputs from the head is inconsistent with the notion that independent oscillations from the tail require synaptic input from head circuitry.

Taken together, our results point to a new working model of C. elegans forward locomotion (Figure 3.9). Three oscillator units are depicted: an unknown head CPG, the VNC motor neurons between VB3 and AS7, and the VNC motor neurons between VB9 and AS11. The two VNC units are justified by our data from worms in which the VNC and DNC were cut in multiple locations (Figure 3.6C). We include AS in the model because it is the only class of VNC motor neurons that we have not directly investigated in our 2FU assay, and could be important for rhythm generation in the tail. The premotor interneurons, especially AVB, are important for activating the oscillatory circuit and may have additional unexplained roles in posterior rhythm generation. These are not the only circuit units capable of generating oscillations; when we severed the VNC and DNC at arbitrary locations we found that oscillations resume closely posterior to each cut over a wide range of circuit sizes (Figure 3.7). Moreover, we cannot rule out the possibility that even smaller circuit units, perhaps even individual motor neurons, can generate rhythmic outputs.
Figure 3.9. A model for *C. elegans* forward locomotion

Two units of the VNC motor neurons (and potentially more subsections) are capable of independent rhythm generation. However, all oscillating units are coupled by proprioceptive coupling (Wen et al., 2012) and another unknown, likely non-proprioceptive coupling mechanism that allows signaling in the anteriorward direction, and potentially also in the posteriorward direction. Pre-motor interneurons activate or suppress this circuit. AVB may have an additional, unexplained role in rhythm generation.
Dissecting the relative contributions of cellular pacemakers, network oscillators, and reflex loops to rhythmic motor generation and coordination has been a longstanding challenge in vertebrates and invertebrates (Kristan et al., 2005; Kiehn, 2006). Our finding that the architecture for rhythm generation in *C. elegans* locomotion shares key properties with other vertebrate and invertebrate models sets the stage for molecular, cellular, and network-level investigations of motor coordination in a uniquely tractable model organism.
MATERIALS AND METHODS

Strains

We maintained *C. elegans* on 6 cm NGM plates seeded with *E. coli* OP50 at 20°C using standard methods (Sulston and Hodgkin, 1988b). For all optogenetic experiments, we added 100 mM all-trans retinal (ATR) in ethanol at 0.8% by volume to the bacteria suspension before seeding the plates, and kept plates in darkness. All strains were synchronized by hypochlorite bleaching and allowed to hatch on an NGM plate without food. L1 arrested larvae were transferred to OP50 or OP50+ATR plates and allowed to grow to the appropriate stage. Unless otherwise specified, all experiments were performed using day 1 adult hermaphrodites.

All strains used in this study are listed in Tables 1 and 2. All transgenic strains were outcrossed a minimum of 3 times against N2.
Table 1: Transgenic arrays acquired or generated for this study

<table>
<thead>
<tr>
<th>Transgene</th>
<th>Plasmid or Reference</th>
<th>Description</th>
<th>Purpose</th>
<th>Strain</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsIs48</td>
<td>(Chase et al., 2004)</td>
<td>Punc-17::GFP</td>
<td>Identification of VNC and DNC</td>
<td>LX929</td>
</tr>
<tr>
<td>aklIs11</td>
<td>(Zheng et al., 1999)</td>
<td>Pnmr-1::ICE; lin-15(+)</td>
<td>Ablation of INs</td>
<td>VM4770</td>
</tr>
<tr>
<td>kylIs36</td>
<td>(Zheng et al., 1999)</td>
<td>Pglr-1::ICE; lin-15(+)</td>
<td>Ablation of INs</td>
<td>VM4771</td>
</tr>
<tr>
<td>wenIs001</td>
<td>pJH2918</td>
<td>Pacr-5::ArchT::RFP; lin-15(+)</td>
<td>Inhibition of B motor neurons</td>
<td>WEN001</td>
</tr>
<tr>
<td>qhIs1</td>
<td>(Leifer et al., 2011; Husson et al., 2012)</td>
<td>Pmyo-3::NpHR::ECFP; lin-15(+)</td>
<td>Inhibition of muscles</td>
<td>YX9</td>
</tr>
<tr>
<td>qhIs2</td>
<td></td>
<td>Pacr-2::wCherry; dpy-20(+)</td>
<td>Identification of B</td>
<td>YX10</td>
</tr>
<tr>
<td>qhIs4</td>
<td>pJH1841</td>
<td>Pmyo-3::PH::miniSOG(Q103L) + Pmyo-3::mCherry + Pttx-3::RFP</td>
<td>Ablation of BWM</td>
<td>YX203</td>
</tr>
<tr>
<td>qhIs5</td>
<td>(Xu and Chisholm, 2016)</td>
<td>Punc-17(beta)::PH::miniSOG(Q103L) + Pacr-2::mCherry + Pttx-3::RFP</td>
<td>Ablation of B</td>
<td>YX234</td>
</tr>
<tr>
<td>hpEx803</td>
<td>(Wen et al., 2012)</td>
<td>unc-9(fc16); hpl3; Prgef-1-unc-9cDN) + odr-1 Punc-17::ChR2(H134R)::YFP; lin-15(+)</td>
<td>Neuronal unc-9 rescue; Stimulation of cholinergic neurons</td>
<td>ZM2509</td>
</tr>
<tr>
<td>zxIs6</td>
<td>(Liewald et al., 2008)</td>
<td>Pglr-1::chop-2(H134R)::YFP; lin-15(+)</td>
<td>Identification of INs</td>
<td>ZM4624</td>
</tr>
<tr>
<td>hpl166</td>
<td>(Gao et al., 2015)</td>
<td>Punc-17::NpHR::ECFP; lin-15(+)</td>
<td>Inhibition of cholinergic neurons</td>
<td>ZM5016</td>
</tr>
<tr>
<td>hpl178</td>
<td>(Leifer et al., 2011)</td>
<td>Psra-11::D3cpv</td>
<td>Identification of AVB</td>
<td>ZM5132</td>
</tr>
<tr>
<td>hpl179</td>
<td>(Kawano et al., 2011)</td>
<td>Punc-4::tomm-20::miniSOG::urSL::wCherry; lin-15(+)</td>
<td>Ablation of A and VC</td>
<td>ZM7690</td>
</tr>
<tr>
<td>hpl366</td>
<td>pJH2843</td>
<td></td>
<td></td>
<td>ZM7691</td>
</tr>
<tr>
<td>hpl371</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strain</td>
<td>Transgenes</td>
<td>Description</td>
<td>Purpose</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>-------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>YX119</td>
<td>qhls1; unc-49(e407)</td>
<td>Muscle::NpHR, unc-49</td>
<td>2FU with D function impaired</td>
<td></td>
</tr>
<tr>
<td>YX126</td>
<td>qhls1; hpls371</td>
<td>Muscle::NpHR, A/VCA::miniSOG Cholinergic Neurons::NpHR&amp;ChR2, A/VCA::miniSOG</td>
<td>2FU with A and VC removed</td>
<td></td>
</tr>
<tr>
<td>YX127</td>
<td>hpls178; hpls371; zxls6</td>
<td>Muscle::NpHR, DB disrupted</td>
<td>Inhibition or excitation of Cholinergic neurons with A removed</td>
<td></td>
</tr>
<tr>
<td>YX135</td>
<td>qhls1; vab-7(e1562) III.</td>
<td>Muscle::NpHR, DB disrupted</td>
<td>2FU with AVB::B gap junctions disrupted</td>
<td></td>
</tr>
<tr>
<td>YX137</td>
<td>qhls1; unc-7(e5)</td>
<td>Muscle::NpHR, unc-7</td>
<td>Entrainment with BWM::BWM gap junctions disrupted</td>
<td></td>
</tr>
<tr>
<td>YX139</td>
<td>qhls1; unc-9(fc16); hpEx803</td>
<td>Muscle::NpHR, UNC-9 disruption in muscles only</td>
<td>2FU with AVB::B gap junctions disrupted</td>
<td></td>
</tr>
<tr>
<td>YX140</td>
<td>qhls1; unc-9(fc16)</td>
<td>Muscle::NpHR, unc-9</td>
<td>2FU with some B removed OR undulation with nerve cords severed</td>
<td></td>
</tr>
<tr>
<td>YX148</td>
<td>qhls1; qhls4</td>
<td>Muscle::NpHR, AB::RFP</td>
<td>Assessment of ICE ablations</td>
<td></td>
</tr>
<tr>
<td>YX152</td>
<td>hpls166; akl11</td>
<td>IN::ICE&amp;YFP</td>
<td>Assessment of ICE ablations</td>
<td></td>
</tr>
<tr>
<td>YX153</td>
<td>hpls166; kyls36</td>
<td>IN::ICE&amp;YFP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>YX159</td>
<td>qhls2; akl11</td>
<td>Muscle::NpHR, IN::ICE</td>
<td>2FU with PVC removed</td>
<td></td>
</tr>
<tr>
<td>YX160</td>
<td>qhls2; kyls36</td>
<td>Muscle::NpHR, IN::ICE</td>
<td>2FU with PVC removed</td>
<td></td>
</tr>
<tr>
<td>YX177</td>
<td>hpls366; vls48</td>
<td>A/VCA::miniSOG, Cholinergic Neurons::GFP</td>
<td>Undulation with nerve cords severed and A removed.</td>
<td></td>
</tr>
<tr>
<td>YX223</td>
<td>qhls1; qhls9</td>
<td>Muscle::NpHR, DB ablated</td>
<td>2FU with B removed OR undulation with nerve cords severed and B removed.</td>
<td></td>
</tr>
<tr>
<td>YX230</td>
<td>qhls1; hpls179</td>
<td>Muscle::NpHR, AVB labeled for ablation</td>
<td>2FU with AVB removed</td>
<td></td>
</tr>
</tbody>
</table>
Construction and validation of the optogenetic targeting system

The construction and validation of the optogenetic targeting system is discussed in Chapter 1.

Optogenetic inhibition and stimulation

For head and neck optogenetic inhibition experiments, YX9 (Muscle::NpHR), ZM5016 (Cholinergic Neurons::NpHR), WEN001 (B::Arch), YX127 (Cholinergic Neurons::NpHR&ChR2, A/VC::miniSOG), YX119 (Muscle::NpHR, unc-49(e407)), YX135 (Muscle::NpHR, vab-7(e1562), YX137 (Muscle::NpHR, unc-7(e5)), or YX140 (Muscle::NpHR, unc-9(fc16)) larvae were transferred to ATR plates and allow to grow to first day of adulthood. Up to 20 adult worms were mounted at a time on our optogenetic targeting system. Worms were sampled with replacement from the dextran chamber, and illuminated with a green (532 nm wavelength) laser a total of 1-3 times in the indicated region, with at least 10 s between successive illuminations of the same animal.

For activation of posterior B and AS motor neurons, YX127 (Cholinergic neurons::NpHR&ChR2, A/VC::miniSOG) worms were allowed to grow to the second larval stage on ATR plates, and then illuminated en masse with blue light (470 ± 17 nm wavelength) at 3 mW/mm² for 20 minutes to ablate the A-type motor neurons. Worms were transferred to a fresh ATR plate to grow for 2 more days. To confirm a loss of reversal capability in each day 1 adult, worms were individually prodded with a platinum wire worm pick. All worms tested failed to move backwards during this assay. Worms were then mounted on the optogenetic targeting system as before. Global amber (580 ± 15 nm wavelength) illumination was applied through the transmitted light port of the microscope and varied in irradiance until just strong enough to paralyze the worm,
presumably through the action of Punc-17::NpHR. Once paralyzed or nearly paralyzed, 473 nm laser light was applied to the indicated region of the tail using the DMD.

For rhythmic inhibition of muscles or cholinergic neurons, synchronized YX148 (Muscle::NpHR, AB::RFP) L4 larvae, YX139 (Muscle::NpHR, BWM gap junctions disrupted) L4 larvae, YX127 day 1 adults, or WEN001, day 1 adults grown on ATR plates were mounted on the optogenetic targeting system and illuminated periodically at the indicated frequency and location through our custom software. For this experiment, L4 qhls1 larvae were used because expression of myo-3::NpHR::ECFP in the BWMs appeared to be weaker in adults. YX127 animals were not exposed to blue light prior to this experiment.

**Neuron and muscle photoablation using miniSOG**

For optogenetic 2FU experiments, YX126 (Muscle::NpHR, A/VC::miniSOG) larvae were allowed to grow for 2 days on OP50 plates until the L4 stage, at which RFP was visible in the both the A- and VC-type motor neurons. Worms were bulk illuminated with blue light (wavelength 470±17 nm) at 3.5 mW/mm² for 20 minutes of total illumination time using 0.5 s on / 1.5 s off pulse train (Qi et al., 2012) from a Leica EL6000 light source. After illumination, the larvae were transferred to OP50+ATR plates and incubated for 2 additional days, but did not appear to grow past the L4 stage. During the optogenetic experiments, worms were observed swimming forward and stopping, but never in reverse. In addition, worms were handled individually and recovered from the dextran chamber for fluorescence imaging. We mounted each worm on a separate 4% agar pad and acquired RFP images on a compound fluorescence microscope (Leica DMI6000B). In control worms, which grew into adults, RFP was visible in the VC-type motor neurons, some A-type motor neurons (RFP was visible in all A-type motor neurons at the L2-L4
stages), and the posterior intestine. The normal RFP expression pattern in A- and VC-
type motor neuron was absent in all illuminated worms. However, neurons in the tail
corresponding to VA12, DA8, or DA9 were usually visible (see also Figure 3.6 - Figure
Supplement 1A).

For VNC/DNC cauterization experiments, YX177 (AB::RFP, A/VC::miniSOG) worms
were exposed to blue light at the L2 stage using 0.5 s on / 1.5 s off pulsing and surgically
manipulated at the day 2 adult stage as described below. Ablation of the A motor
neurons was assessed by recording GFP fluorescence images of each animal after
behavioral imaging. In all animals, only B and AS motor neurons were visible along most
of the VNC, with the exception of the three posterior A motor neurons noted above.

For body wall muscle ablation experiments, YX203 (BWM::miniSOG) adults were
immobilized on agar pads using polystyrene beads (Kim et al., 2013) and the indicated
region of the body was exposed to light with wavelength 470 ± 20 nm at an irradiance of
75 mW/mm² through a 40x objective on a Leica DMI6000B or DMI4000B microscope for
5.5 minutes with a 0.5 s on / 0.5 s off pulse protocol (Xu and Chisholm, 2016). Spatial
selectivity was achieved by restricting the diameter of illumination by adjusting the
microscope’s epifluorescence field diaphragm. After illumination, worms were recovered
to an unseeded plate and immediately transferred to a 17% dextran chamber for
behavioral imaging.

For B ablation experiments using Punc-17::PH::miniSOG (YX223), we found that
illumination at any larval stage preferentially killed the DA and DB motor neurons, but left
most VA and VB alive, as determined by the loss of all dorsal commissures and DNC
labeling by Pacr-2::mCherry (not shown). Hence, we illuminated all YX223 animals at
the L1 stage for 20 minutes with 0.25 s on / 0.25 s off pulsing (Xu and Chisholm, 2016) to kill DB. Worms were recovered to ATR (for optogenetic 2FU) or regular seeded (for VNC surgery) plates. Additional laser ablation of VB neurons was performed at the L4 stage, and lesioning of the VNC and DNC was performed at the day 2 adult stage (both described below).

Ablation of premotor interneurons by ICE

Prior to conducting optogenetic experiments, we generated strains YX152 (Pnmr-1::ICE; Pglr-1::ChR2::YFP) and YX153 (Pglr-1::ICE; Pglr-1::ChR2::YFP) to test whether the interneurons were appropriately ablated. Control ZM4624 (Pglr-1::ChR2::YFP) L1 larvae and adults showed bright YFP labeling of many neurons, including many head neurons and PVC, the only pair of labeled neurons in the tail.

In L1 arrested YX152 and YX153 worms, many neurons were also easily visualized by YFP, although normal locomotion was impaired. By the adult stage, YFP signals in all head and tail neurons had nearly vanished in all YX153 worms (N=18); some small and faint fluorescent puncta, similar in appearance to intestinal birefringent granules, were visible in the nerve ring and tail. These dim puncta did not have visible processes, and we were unable to identify any head neurons or PVC in any of these worms. In YX152 adults, PVC was similarly not identifiable in any worm (N=10), although many brightly YFP-labeled neurons were visible in the nerve ring. These results suggest that all or most Pglr-1 positive interneurons are ablated in Pglr-1::ICE worms and that PVC (and likely other Pnmr-1 positive neurons) are ablated in Pnmr-1::ICE worms. AVB are likely
present in both *Pglr-1::ICE* and *Pnmr-1::ICE* worms. (Kawano et al., 2011; Kawano, Po, and Zhen, unpublished).

For optogenetic experiments, we generated strains YX159 (Muscle::NpHR, IN::ICE) and YX160 (Muscle::NpHR, IN::ICE) and performed optogenetic illuminations as in our original optogenetic 2FU experiments.

**Pulsed infrared laser cauterization of neurons and nerve cords**

For ablation of B motor neurons or AVB interneurons, YX148 (Muscle::NpHR, AB::RFP), YX223 (Muscle::NpHR, DB killed by miniSOG, see above), or YX230 (Muscle::NpHR, AVB labeled) worms were raised on ATR plates until most animals were at the L3 to L4 stage, and then immobilized on 4-10% agar pads using 50 nm polystyrene beads (Kim et al., 2013). Each pad was mounted on a pulsed infrared laser system (Churgin et al., 2013) that had been modified with increased power to deliberately kill cells. Each neuron, visualized by RFP or GFP fluorescence optics, was irradiated with a single 0.8 to 1.6 ms pulse of the 400 mW laser through a 63X oil immersion objective. We determined that a single 0.8 ms pulse reliably kills a targeted VNC motor neuron, has a 50% chance of killing a VNC neighbor within 2.5 μm, and has a 10% chance of killing a VNC neighbor within 5 μm from the target (A. D. F. and C. F.-Y., unpublished data). After ablation, worms were recovered and transferred to a fresh ATR plate to resume development for 1 additional day. During optogenetic experiments, worms were handled individually and recovered to agar pads after illumination.

To sever the VNC and DNC, day 2 adult YX148 or LX929 (Cholinergic neuron::GFP) worms were immobilized with polystyrene beads and mounted on our infrared laser
system as before, and the indicated area of the wCherry- or GFP-labeled cord was illuminated with a train of 10-25 IR laser pulses with 2 ms duration. Worms were transferred to fresh unseeded plates and allowed to recover for at least four hours before behavioral and fluorescence imaging. For behavioral imaging, worms were mounted individually in 20% dextran chambers and recorded swimming for at least 1 minute under dark field illumination. Most worms were inactive 4 hours after surgery, especially when the VNC and DNC were lesioned in two locations (not shown).

In many other systems, mechanical, electrical, or chemical stimuli can be applied to induce swimming or fictive swimming in an otherwise quiescent preparation (Kristan et al., 2005). To agitate C. elegans, we mechanically vibrated each worm using a 200 Hz cell phone motor for periods of 10-20 s to induce locomotion at least twice during each recording. After behavioral imaging, each worm was transferred to a pad and imaged for red or green fluorescence imaging.

**Fluorescence imaging and identification of neurons and nerve cord lesions**

For all neuron ablation or nerve cord lesioning experiments, we acquired RFP or GFP fluorescence images of each animal through a 40X objective on a compound fluorescence microscope (Leica DMI 6000B). Through examination of images, A and B type motor neurons or AVB interneurons were labeled as present or missing based on the location, stereotypic ordering, commissural orientation, and the presence or absence of commissures from dorsal A- or B-type motor neurons. In ablation conditions labeled “most VB” in Figure 3.5 and Figure 3.6 Figure Supplement 2, we only used data from animals in which at most 3 out of the 9 VB neurons between VB3 and VB11 inclusive were visible. In DB ablation conditions in the same two figures, all DB motor neurons appeared to be missing, and occasionally some VA/VB were missing as well. However,
we generally could not identify B motor neurons anterior to VB3 because of clustering and the presence of very bright AiY::RFP in qhs9. In each category in Figure 3.4F, we included only worms for which all indicated B motor neurons were missing. Some individuals in each category had additional missing neurons. In Figure 3.4 – Figure Supplements 3 and 4, all ablated B motor neurons are indicated for each individual worm example. In AVB ablation conditions, we only analyzed data from worms in which both AVB cell bodies and their associated processes were removed. Because cell killing occurs over a ~3 μm radius volume in our system, it is highly likely that other head neurons were also damaged or killed by this procedure. For nerve cord lesioning experiments, we only analyzed data from animals in which all indicated VNC/DNC targets were clearly severed.

**Head lesions using a heated wire**

To broadly lesion the head and inhibit anterior bending, four freely crawling adult N2 worms were gently touched on or near the head with a platinum wire attached to a soldering iron. The worms appeared to crawl backwards after the initial touch, so we applied a second touch to the agar near the tail to induce forward locomotion. We recorded behavior immediately after lesioning.
**Heterogenous mechanical environment experiments**

Day 1 adult N2 worms were transferred to a slide containing 3 to 5 µL islands of solutions of high viscosity 3% hydroxypropylmethylcellulose (HPMC, Ashland Benecel K200M) in NGMB, surrounded by NGMB without HPMC. A second slide, spaced by 125 µm thick plastic spacers, was placed on top to form a 2-dimensional chamber similar to those used for optogenetics experiments, but with an inhomogeneous mechanical environment. We imaged each slide under dark field illumination on a Nikon TE2000-S microscope and recorded worms transitioning from low viscosity to high viscosity regions.

**Curvature segmentation, analysis, and statistics**

For experiments with our optogenetic targeting system, the real-time segmentation for body targeting was recorded to disk along with each video frame. We wrote custom MATLAB codes (Fouad et al., 2018a) to compute the curvature of the worm in each frame using the recorded centerline coordinates. All analysis codes, codes for the optogenetic targeting system, and source data for figures are freely available (Fouad et al., 2018a). Frequencies and amplitudes in optogenetic experiments were measured over either a 3 s non-illuminated interval ending at the start of illumination, or a 3 s illuminated interval beginning at the start of illumination. Trials were excluded if, during the period of analysis, (1) the worm showed reverse locomotion at any time, or (2) the segmentation algorithm was disrupted, for example if the worm touched a bubble, another worm, or the edge of the chamber.
For all other experiments, worm segmentations were generated from dark field videos using WormLab software (MBF Bioscience, Williston, VT). The centerline coordinates were exported and curvature maps constructed as before. To identify bouts of forward locomotion in 1-2 min worm recordings, we computed the activity level and wave direction in the kymogram as a function of time and body coordinate. Bouts of forward locomotion in body segments were identified when the activity level was higher than a fixed threshold and the local direction of wave propagation was anterior-to-posterior for longer than the amount of time specified (typically 2-3 s).

To measure frequencies of undulation at any body coordinate, we computed the Fourier transform of time derivative of the curvature, and identified the frequency corresponding to the maximum amplitude within a 0 to 2.5 Hz window. For trials in which no clear peak emerged (i.e. the maximum amplitude was less than a fixed threshold), no undulations were considered to have occurred and the frequency was treated as zero. The same threshold was used for every frequency measurement presented. In ratiometric measurements, a small number (<2%) of ratios with infinite values were excluded. Bending amplitudes were calculated as the root mean square of time differentiated curvature. Changes in the mean frequency, frequency ratio, or bending amplitude of all paired data were evaluated using a paired t-test.

For blinded, randomized scoring of trials shown in Figure 3.5 and Figure 3.6 Figure Supplement 2, kymograms were scored manually without prior examination of the data. Trials were scored as “2FU” or “oscillating” if the following criteria were met: (1) bending waves were visible in the tail that did not appear to arise from the head, and (2) at least two complete undulatory cycles occurred in the tail, in any portion of the optogenetic inhibition or vibration stimulus time windows, which were 8 and 20 seconds, respectively.
Trials were marked for exclusion if the windows contained exclusively reversal or incorrectly segmented behavior. The manual scoring method allowed detection of 2FU occurring outside the 3 second window used for quantitative analysis, which is why more 2FU trials were found in the DB ablation condition by manual scoring than by quantitative frequency analysis.
**Video titles** (see Fouad, 2018 for video files)

**Video 1:** Posterior undulations after optogenetic inhibition of anterior body wall muscles or cholinergic neurons

**Video 2:** Two-frequency undulation during optogenetic inhibition of neck BWM, cholinergic neurons, or B motor neurons.

**Video 3:** Additional manipulations that evoke two-frequency undulation (2FU):
Stimulation of B and AS during tail paralysis; inhomogenous mechanical environment.

**Video 4:** Removal of B motor neurons by miniSOG and laser ablation eliminates 2FU.

**Video 5:** Posterior undulations occur after severing the ventral and dorsal nerve cords with an infrared laser (VNC-isolated animals).

**Video 6:** Removal of B motor neurons by miniSOG and laser ablation eliminates undulations in VNC-isolated animals.

**Video 7:** Head undulation frequency can be entrained by rhythmic mid-body optogenetic manipulation of the muscles or cholinergic neurons.
ACKNOWLEDGEMENTS

We thank Mei Zhen, Quan Wen, Min Wu, Michelle Po, Yishi Jin, Andres Villu Mariq, and Alexander Gottschalk for providing strains. Some strains were provided by the CGC, which is funded by NIH Office of Research Infrastructure Programs (P40 OD010440). C. F.-Y. was supported by the National Institutes of Health, Ellison Medical Foundation, and Sloan Research Foundation. A. D. F. was supported by the National Institutes of Health. S. T. was supported by an Abraham Noordergraaf Research Fellowship and a Littlejohn Fellowship. J. R. M. was supported by a Holtz Undergraduate Research Fellowship. We thank Mei Zhen, Michael Nusbaum, David Raizen, Vijay Balasubramanian, Robert Kalb, Gal Haspel, Brian Chow, and Edward Fouad for helpful suggestions and discussions, Matthew Churgin for technical assistance, and Wassana Techadilok for assistance with figure preparation.
CHAPTER 4: CONCLUSIONS AND FUTURE DIRECTIONS

The functional architecture of the C. elegans motor circuit is highly analogous to that of more complex invertebrate and vertebrate models.

In Chapter 2, I showed that the C. elegans circuit for forward locomotion contains multiple subunits capable of independent rhythm generation. These units, whether isolated by surgical or optogenetic manipulations, invariably showed higher frequency oscillations in the posterior of the animal than in the anterior. I further showed that extensive, bidirectional rhythmic coupling exists within the circuit and that the animal’s normal frequency of locomotion in our assay was generally intermediate between the frequencies of head and tail oscillations isolated by optogenetic relaxation of the neck muscles. These findings suggest that rhythmic forward locomotion is not driven by one oscillator overruling and entraining the others, but by the network as a whole operating as a large rhythmic unit with properties that may be different from any individual unit in isolation.

Despite the fact that C. elegans is a non-segmented, millimeter-long animal with only 300 neurons, this functional architecture for locomotion is highly analogous to that shown by more complex species. Leech swimming is driven by central pattern generators located in each of the 18 ganglia along the body segments (Kristan et al., 2005; Mullins et al., 2011). As shown in Figure 4.1, a gradient exists among the rhythm generating units in isolation, with the lowest frequency oscillators found closest to the head and the highest frequency closer to the mid-body. Moreover, the oscillatory frequency of the intact chain of oscillators is always intermediate between the frequencies of the
highest and lowest frequency ganglia in isolation (Zheng et al., 2007). Coordination between the oscillatory units to allow them to synchronize and reach an intermediate frequency requires extensive, bidirectional inter-oscillator coupling, and indeed such a mechanism exists in the leech (Weeks, 1981), just as I found that it exists in C. elegans.

The analogy between motor function in C. elegans and other animals even extends to vertebrates. The swimming lamprey and zebrafish also contain distributed rhythm generating circuits along the segments of the spinal cord, with their own gradients in excitability and rhythmic properties, but linked by substantial intersegmental coupling (Kiehn, 2006; Mullins et al., 2011).

Taken together, my findings suggest that the functional architecture of the C. elegans forward motor circuit is highly analogous to those found in other vertebrate and invertebrate models.
Figure 4.1. Computational modeling of the leech swim CPG.

Reproduced with modifications from (Zheng et al., 2007). Upper panel: periods of the CPGs within the body ganglia in isolation show a gradient from head to tail. Lower panel: the overall swim CPG in the intact circuit is intermediate between the highest and lowest frequency ganglia.
Mechanisms of rhythm generation

The extensive functional similarity between *C. elegans* and more complex animals offers the hope of understanding a highly conserved neuronal strategy in deeper detail than has been possible in more complex model organisms. Because of the unique experimental tractability of *C. elegans* in genetics and neuroscience, it is possible to investigate rhythm generating circuits at the network, cellular, and molecular levels to understand the neuronal basis of locomotion.

The first objective of future studies should be to discover the network-level mechanisms that allow the whole-body rhythmic unit to generate locomotory rhythms. Historically, there has been a great debate over whether rhythms arise from central pattern generators intrinsic to the nervous system, which can generate oscillations even without the physical execution of movement, or from sensory reflex loops that drive oscillations by detecting and reacting to each body bend (Kristan et al., 2005; Mullins et al., 2011) (Figure 4.2 A, B). Although central pattern generators have been demonstrated in many organisms, it is also well appreciated that even where they exist, sensory feedback is highly capable of entraining central clocks, suggesting that its contribution to rhythmic movement cannot be ignored (Figure 4.2C). For example, injection of sinusoidal current into leech stretch sensitive VSR neurons was found to completely entrain rhythmic activity in the VNC motor circuits (Yu et al., 1999; Yu and Friesen, 2004).
Figure 4.2. Overview of proposed methods of rhythm generation.

(A) A central pattern generator (CPG, green circle) is capable of generating rhythmic signals even if the execution of movement, the bending of the muscles (tan ovals), is interrupted. (B) A reflex loop generates oscillations concomitant with the execution of body bends. Stretch sensors on each side sense bending of the muscle and then activate motor neurons to stimulate bending in the opposite direction. This creates a reflex loop that drives oscillations. (C) The leech contains a central pattern generator, but injection of sinusoidal current into the VSR stretch sensitive neurons can entrain the CPG and define the locomotory behavior.
To determine which mechanism(s) explain *C. elegans* forward locomotion, Chris Fang-Yen and I have designed new experiments that were performed by Pilar Alvarez-Illera and Shelly Teng. We briefly interrupted normally progressing forward locomotion by hyperpolarizing neurons or muscles for a fraction of a second. If the rhythm generator is not attuned to sensory feedback – a blind central pattern generator – the rhythm should continue in phase after only a brief disruption (Figure 4.3A). If, on the other hand, sensory reflexes play a key role in rhythm generation or modulation, a brief interruption of forward movement should permanently shift the phase of locomotion after the manipulation. Our results (not shown), strongly support the latter possibility.

Moreover, I have attempted to search for *C. elegans* CPGs by conducting calcium recordings of the B-type motor neurons in mechanically immobilized worms. These experiments were performed with the help of Alice Liu. Our results showed no clear pattern of oscillation in these motor neurons, except for some rhythmic activities in two of the anterior-most B motor neurons. These results do not rule out the possibility of CPGs existing along the remainder of the motor circuit, but as of yet no evidence for that possibility has emerged.

Taken together, my preliminary results show that proprioception is closely associated with the generation of body rhythms, but do not support or disprove the idea that CPGs such as cellular pacemakers exist within the circuitry. Hence, much more work needs to be done before we will have a clear picture of how the motor circuit generates rhythms. The starting point for these studies will likely be additional calcium imaging of the motor neurons combined with simultaneous optogenetic stimulation of motor neurons or interneurons.
Figure 4.3. Identifying a CPG or a reflex loop by its response to brief inhibition of movement.

(A) An internal CPG that runs without concern for the execution of body movements would be only transiently effected by the manipulation, and then resume its ongoing course. (B, C) A reflex loop – or a CPG that is highly sensitive to the execution of movement – would be permanently delayed by the inhibition, leading to a phase shift.
Depending on where in the cycle the inhibition occurred, the phase shift may be an advance or a delay.
Towards an integrated network, cellular, and molecular understanding of rhythmic locomotion

As described in Chapters 1 and 2, substantial progress towards understanding the network and cellular principles that govern rhythmic locomotion has been made in other organisms such as the leech and the lamprey. Although this highly fruitful work can and should continue, it must be appreciated that not only are the pictures of rhythm generation in these animals incomplete, but integrating these findings with their associated molecular mechanisms is restricted by the lack of genetic access to these species. Even genetically accessible model organisms such as the mouse and zebrafish also do not provide an ideal platform for attaining an integrated understanding of locomotion because their spinal networks are fantastically complex, and genetic manipulations in longer life-cycle animals are extremely costly.

*C. elegans*, with its 3 day life cycle, genetic manipulability, and amenability to rapid genetic screening, provides the most tractable platform to develop a multilevel understanding of rhythmic locomotion. The most obvious targets for genetic efforts should be screens to identify the molecules involved in stretch sensation in the B motor neurons, which have yet to be found (Wen et al., 2012). Not only would characterization of these molecules help explain how wave propagation and communication between oscillatory units is achieved, but the preliminary results described above suggest that stretch sensitive molecules may play a key role in generating rhythmic bends in *C. elegans*. A genetic screen for these molecules can be approached by looking for animals with defective tail movement during forward swimming. Because of the high degree of homology between the *C. elegans* genome and that of humans (Lai et al., 2000), it is also quite possible that the molecular machinery uncovered may have a role in human
movement as well. The ability to dissect these molecular components and pathways, and study them in the context of individual cells or small networks of neurons, makes *C. elegans* perhaps the most promising model organism available today for a multilevel understanding of rhythmic locomotion.
APPENDIX I: QUANTITATIVE ASSESSMENT OF FAT LEVELS IN CAENORHABDITIS ELEGANS USING DARK FIELD MICROSCOPY

Anthony D. Fouad†, Shelley H. Pu†, Shelly Teng†, Julian R. Mark†, Moyu Fu†, Kevin Zhang†, Jonathan Huang†, David M. Raizen‡, and Christopher Fang-Yen†,†

†Department of Bioengineering, University of Pennsylvania, Philadelphia, PA, 19104
‡Department of Neuroscience, University of Pennsylvania, Philadelphia, PA, 19104
§Department of Neurology, University of Pennsylvania, Philadelphia, PA, 19104

This appendix is a lightly edited version of a paper published in the journal G3 (Fouad et al., 2017). During the course of my studies on the C. elegans motor circuit, Dr. Fang-Yen and I realized that dark field microscopy, which we use extensively, can also provide a simple metric for evaluating worm fat levels noninvasively and without exogenous stains. Most data in this appendix was collected by me or by other students working under my immediate supervision. However, the data in Figures A1.4 and A1.5 was collected by Shelley H. Pu working independently and under my daily supervision, and half of the data in Figure A1.3 was collected by Shelly Teng and Julian R. Mark working independently under my remote supervision.
ABSTRACT

The roundworm *Caenorhabditis elegans* is widely used as a model for studying conserved pathways for fat storage, aging, and metabolism. The most broadly used methods for imaging fat in *C. elegans* require fixing and staining the animal. Here, we show that dark field images acquired through an ordinary light microscope can be used to estimate fat levels in worms. We define a metric based on the amount of light scattered per area, and show that this light scattering metric is strongly correlated with worm fat levels as measured by Oil Red O staining across a wide variety of genetic backgrounds and feeding conditions. Dark field imaging requires no exogenous agents or chemical fixation, making it compatible with live worm imaging. Using our method, we track fat storage with high temporal resolution in developing larvae, and show that fat storage in the intestine increases in at least one burst during development.

INTRODUCTION

The roundworm *C. elegans* has been an important model for understanding basic mechanisms of metabolism and energy storage. Most of the approximately 400 *C. elegans* genes known to regulate fat storage have homologues in mammals, and many of these homologues have also been found to regulate metabolism (Lai et al., 2000; Ashrafi et al., 2003; Kniazeva et al., 2003; McKay et al., 2003; Jia et al., 2004; Kniazeva et al., 2004; Ludewig et al., 2004; Mak et al., 2006; Ashrafi, 2007; McKay et al., 2007; Jones et al., 2009; Soukas et al., 2009).
Critical to these studies are methods for measuring worm fat storage. A broadly suitable tool for this task should satisfy three criteria. First, it should be capable of measuring fat stores with high spatial and temporal resolution in live worms, allowing changes in fat storage in response to genetic or exogenous manipulations to be investigated longitudinally. Second, it should be scalable to provide such detailed information for a large number of animals. Finally, an ideal tool would be technically simple and inexpensive for labs to implement. Although a wide variety of methods for measuring fat in *C. elegans* are available, none are able to satisfy all three requirements.

Quantitative lipid biochemistry assays, which directly measure triglyceride content in a large number (thousands) of worms, are widely considered the gold standard for measuring *C. elegans* fat content (Ashrafi et al., 2003; O’Rourke et al., 2009; Soukas et al., 2009). This assay is usually performed by gas chromatography / mass spectroscopy (GCMS), but can also be conducted using colorimetric kits (Schulz et al., 2007). In both methods, amounts of triglyceride are normalized to amounts of protein or phospholipid to obtain relative measurements useful for comparing groups. However, neither method is feasible for live worms, small numbers of worms, or for determining the spatial distribution of fat in the worm body.

Some of the limitations of biochemical methods can be addressed by optical techniques for imaging fat distributions in individual worms. Fluorescence or absorption microscopy of lipid-staining dyes such as Oil Red O (ORO) has been validated against lipid biochemistry as a stain for major fat stores in *C. elegans* (O’Rourke et al., 2009; Wahlby et al., 2014). However, this method still requires fixed animals and a laborious staining procedure. While *in vivo* use of the lipid-staining dye Nile Red has been reported (Ashrafi et al., 2003; Mak et al., 2006), this method has been shown to produce data that
fail to correlate with triglyceride levels, for example increasing rather than decreasing in fluorescence upon starvation of animals (O'Rourke et al., 2009).

The unsuitability of lipid staining in live animals makes it difficult to record worm fat levels longitudinally. This technical limitation motivated the development of alternative optical methods for visualizing fats in *C. elegans*. Coherent anti-Stokes Raman scattering (CARS) microscopy, which uses intrinsic molecular vibrational modes as a contrast mechanism (Cheng and Xie, 2003; Evans and Xie, 2008), has been used to evaluate *C. elegans* fatty tissues without exogenous labels (Hellerer et al., 2007; Le et al., 2010; Yen et al., 2010). Stimulated Raman Scattering (SRS) has also met success in quantifying fat levels (Wang et al., 2011). However, CARS and SRS are technically complex and prohibitively expensive for most groups (Wählby et al., 2014). Finally, a more recent technique involves genetically modifying worms to express a GFP-labeled, lipid-droplet associating protein (Liu et al., 2014b), although this may perturb the natural properties of lipid stores (Hellerer et al., 2007).

Many researchers have reported an association between high fat accumulation and a darker intestine under bright field illumination (Kenyon et al., 1993; Apfeld and Kenyon, 1998; McKay et al., 2003; Avery and You, 2005). Worms lacking dark, fatty intestinal granules appear pale or transparent under bright field optics (McKay et al., 2003). Under dark field illumination (Figure A1.1A and methods), in which contrast is inverted in comparison to bright field illumination, differences in optical scattering between starved and well fed worms are plainly visible (Figure A1.1B). These observations are consistent with models, based on light scattering theory, which predict that micron-sized spherical lipid droplets are the dominant scatterers of light in soft tissues (Jacques and Prahl, 1998).
Here we show that differences in optical scattering can be exploited as a quantitative proxy for fat levels in *C. elegans*. We define a simple metric for evaluating dark field images and show that it strongly correlates with ORO staining intensities. We further show that this technique is easily adaptable to high temporal resolution tracking of fat mass during periods of growth and fasting. We propose that scattering analysis will prove useful in screens for mutants that store or deplete fat at unusual rates.
MATERIALS AND METHODS

Worm culture and preparation

C. elegans were maintained on 6 or 10 cm NGM plates seeded with OP50 bacteria, according to standard methods (Sulston and Hodgkin, 1988a). All worms were cultured at 20°C except for temperature sensitive daf-2 mutants, which were grown at 15°C and in some cases shifted to 25°C after the fourth larval stage. To generate synchronized experimental cohorts, we placed gravid hermaphrodites on 10 cm diameter plates for approximately 3 hours to lay eggs, and then removed the hermaphrodites. Progeny were allowed to develop until they reached the stage of interest, typically day 1 of adulthood (within 8-24 hours after reaching adulthood). For each experiment, we culled approximately 20 worms from the plate and imaged them directly. For validation experiments, we culled 20 additional worms from the same plate and stained them with Oil-Red-O as described below.

The following strains were used in this study: N2 (wild type), CB1370 [daf-2(e1370)], DA1113 [eat-2(ad1113)], FQ77 [tph-1(n4622)], JJ1271 [glo-1(zu391)], GH403 [glo-3(kx94)], RB811 [glo-4(ok623)], CE541 [sbp-1(ep79)], HY520 [pod-2(ye60)], and NL1142 [gpa-8(pk345)].

Oil-Red-O Staining

We stained worms with ORO as previously described (O'Rourke et al., 2009). Briefly, worms were collected in a 1.5 mL microcentrifuge tube filled with chilled 1X PBS, washed to remove bacteria, and fixed for 1 hour in 2X MRWB-PFA. Care was taken to ensure that worms were well mixed and not stuck to the sides of the tube. Worms were
washed to remove PFA and dehydrated for 15 minutes in 60% isopropanol. During this time, a portion of stock ORO solution was freshly diluted to form 60% ORO in deionized (DI) H$_2$O, equilibrated by rocking for one hour, and passed through a 0.22 µm syringe filter just before use. Isopropanol was removed from the worms and replaced with the ORO solution, which was allowed to stain for 12-16 hours while gently rocking the tube. After staining, worms were washed with 0.01% Triton in 1x PBS for 15 minutes to remove unbound dye.

**Dark field microscopy (validation experiment)**

Multiple *C. elegans* worms were transferred to a freshly prepared pad consisting of 2% agarose in either deionized water or NGM buffer (NGMB), immobilized in 2 µl of 20 mM NaN$_3$, and placed under a coverslip (Driscoll, 2016). NGMB consists of the same constituents as NGM agar but without peptone, cholesterol, or agar. The slide was mounted on the stage of a compound light microscope (Leica DM2500P), and surrounded by four 4.7 inch long red LED light strips (Oznium, LLC), which were arranged in a square for dark field imaging (*Figure A1.1A*) and powered by a 12V DC power supply. We acquired images through a 10x, 20X, 40X, or 63X microscope objective using a cooled CCD camera (Photometrics CoolSNAP K4).

To adjust for variations in lighting conditions, we used a 1 mm thick scattering phantom composed of 1.5% BaSO$_4$ in PDMS. We marked the surface with a small scratch, mounted it on a microscope slide, and imaged it immediately prior to all imaging sessions. The lighting intensity was manually adjusted to ensure that the mean pixel intensity around the mark was within approximately 10% from a fixed value. The
remaining differences in illumination were corrected during post processing by linearly scaling the pixel intensities of each image according to the ratio between the image's corresponding phantom image, and one phantom image used as a common reference (see detailed protocol in Supplementary Information).

**Oil Red O imaging**

We acquired ORO images via standard bright field microscopy using the same microscope and CCD camera, again using our intensity phantom to regulate and correct bright field lighting differences. ORO strongly absorbs green (510 nm) light, causing it to appear red under transmitted light (Ramírez-Zacarías et al., 1992; Yen et al., 2010). Accordingly, we used monochrome images acquired through a 510 nm fluorescence emission filter (**Figure A1.2C**) to quantify ORO staining levels. By this procedure, heavily stained regions appeared dark. We also acquired a small number of RGB color images using a color CMOS camera (Leica DFC290) (**Figure A1.2B**).
Figure A1.1. Light microscope with dark field illumination.

(A) Red LED strips placed on the microscope stage illuminate the worms from the side. Scattered light is collected by the objective and recorded by a camera.
(B) Dark field images of the reference strain (N2) and feeding defective (eat-2) day 2 adult *C. elegans*, shown with identical lighting conditions and gray scaling. Both animals are oriented with head at lower left. The N2 worm displays strong scattering from its intestine and embryos. Both worms display weak scattering in the head. Scale bar: 250 µm.

Figure A1.2. Fat-staining structures appear bright in dark field images
(A) Dark field image of a live N2 adult worm. (B) True color and (C) monochrome (acquired through a 510 nm filter) images of a fixed, ORO stained adult worm from the same cohort and time point. Scale bar: 250 µm.

**Dark field imaging of developing larvae**

We prepared synchronized, L1-arrested N2 worms by treating gravid hermaphrodites with an alkaline bleach solution and allowing embryos to hatch in M9 buffer without food. Larvae were transferred to agar plates containing streptomycin and seeded with DA837 (Davis et al., 1995), a streptomycin-resistant strain of *Escherichiae coli*, and incubated at 20°C. Starting at 4 hours post feeding, we culled groups of 10-15 worms from these synchronized cohorts and imaged them on a compound microscope (Leica DMI6000B) equipped with red LED light strips and CCD camera. A total of 6 synchronized cohorts was used; one each for hours 4-9, 10-17, 17-24, 24-33, 34-42, and 42-50. To obtain high-resolution images of small larvae, we acquired data from hours 4-31 through a 40x objective (blue points in Figure A1.4). Data from hours 31-50 were acquired through a 20x objective (orange points), since worms at these times were larger. Worms at age 31 hours were imaged through both objectives, providing a basis for combining the data.

**Dark field imaging of worms before and after fasting**

To track worm fat loss as a function of activity level on a per-worm basis, we first imaged 55 synchronized adult worms (in 3 separate experiments) on our dark field microscope, and then loaded them into individual PDMS ‘WorMotel’ microwells filled with agar and
NGM for observation overnight (Churgin and Fang-Yen, 2015). 18 hours later, we retrieved the worms from the wells and imaged them again on our dark field microscope.

**Nile Red staining**

Worms were stained with Nile Red as described (Mak et al., 2006). A 0.5 mg/ml Nile Red stock was diluted to 1 µg/ml in PBS for each experiment. 0.5 ml of the freshly diluted Nile Red was added to plates seeded with DA837 and allowed to equilibrate for a minimum of 2 hours. Subsequently, L1s obtained from bleaching were transferred to the plates and allowed to grow for the desired length of time. In order to account for each developmental stage (L1 through adulthood), selected time points for imaging included 4 hours, 8 hours, 24 hours, 30.5 hours, 44 hours, and 67.5 hours after re-feeding. A separate synchronized cohort was grown for each time point. For each time point, control worms were grown simultaneously without Nile Red addition to the plates. Stained and control worms were imaged through a 63x objective under dark field and red fluorescence illumination.

**Image post processing and analysis**

We wrote custom MATLAB routines to semi-automatically segment and analyze all images. These codes, as well as sample data from our study and a detailed experimental protocol, are available in File S1. In each experiment, we chose a gray scale intensity threshold that would allow reliable identification of the whole body of each worm. Because of variations in worm brightness (for example, eat-2 mutants are darker), this threshold varied between strains. The gray scale intensities of all pixels within each
worm contour were summed and divided by the area (in pixels) to generate a scattering density, in units of arbitrary units (a.u.) per pixel, or simply a.u. ORO images were inverted to form pseudo-dark field images, and then analyzed for pseudo-scattering (absorption) density in the same way. Accordingly, animals with weak or intense ORO staining produced low or high staining densities, respectively. We hypothesized that there would be a positive correlation between dark field scattering density and ORO staining density.

To compare the staining patterns of Nile Red-stained worms to high magnification dark field scattering images, we first manually segmented the boundaries of the worm within the field of view, excluding the head and cuticle. Each image was filtered to extract the high frequency components (e.g. puncta and edges). We then selected pixels above the 80th percentile of gray intensity within each Nile Red or dark field image for comparison. We used the Sorenson-Dice coefficient QS (Sørenson, 1948) to compute the degree of overlap between segmented pixels in dark field and Nile-Red images.
RESULTS

Optical scattering is correlated with ORO staining

Under dark field illumination, we observed the highest intensities (greatest light scattering density) within the intestine and eggs (Figures A1.1B and A1.2A), structures known to be rich in lipid stores (Ashrafi, 2007; O’Rourke et al., 2009). These structures also stained most intensely for lipids by ORO (Figure A1.2B, C), and appeared much larger and brighter in high fat mutants daf-2 and daf-7 (Figure A1.3A), suggesting that optical scattering is correlated with fat content.

To test this idea, we first measured the scattering density, and corresponding ORO staining density, of individual worms in low fat (fasted and gpa-8 mutant), and high fat (daf-2 and daf-7 mutant) conditions (Figure A1.3B). Mean scattering density in fasted worms was significantly lower than in the reference condition, while scattering density in high fat worms was significantly higher than in the reference condition. Mean scattering density in gpa-8 mutants was only slightly less than wild-type (<10% lower; p<0.05), and corresponding ORO measurements did not detect a decrease.
Figure A1.3. Scattering density correlates with ORO staining density.

(A) Dark field images of N2, *daf-2*, and *daf-7* animals. All worms were day 1 adults. Images are shown under identical lighting conditions and gray scaling (not the same as in Figure A1.1B). Scale bar: 250 µm. All animals are oriented with the head on top.
(B) Scattering density for several conditions. Each dot represents one worm. Black lines and boxes are the mean and 95% confidence interval. All dark field measurements (blue dots) were significantly different from the N2 reference by one-way ANOVA with Bonferroni pairwise tests (p<0.05). Food (-) indicates that the animals were fasted for 1 day prior to imaging. Data was pooled for each condition in which multiple experiments were conducted.

(C) ORO staining density is correlated with dark field scattering density. The height of each ellipse denotes the SEM of scattering density and the width denotes the SEM of ORO staining density. Each point represents 5-21 (median 19) worms imaged by dark field and 8-34 (median 15) separate worms from the same cohort imaged after ORO staining. $r^2=0.85$ for the linear fit and $p<10^{-4}$ for the null hypothesis that the slope equals zero. Conditions for each numbered group are given in Table 3.
Table 3: conditions and mutants shown in Figure A1.3.

<table>
<thead>
<tr>
<th>Number</th>
<th>Genotype</th>
<th>Age</th>
<th>Food</th>
<th>Temperature (°C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N2</td>
<td>Adult (day 2)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>N2</td>
<td>Adult (day 3)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>N2</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>N2</td>
<td>L4 larvae</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>N2</td>
<td>Adult (day 2)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>N2</td>
<td>L4 larvae</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>N2</td>
<td>Adult (day 2)</td>
<td>Fasted 24h</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>N2</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>N2</td>
<td>Adult (day 2)</td>
<td>Fasted 24h</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>sbp-1</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>sbp-1</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>pod-2</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td>pod-2</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>14</td>
<td>gpa-8</td>
<td>Adult (day 2)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>gpa-8</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>16</td>
<td>gpa-8</td>
<td>Adult (day 2)</td>
<td>Fasted 24h</td>
<td>20</td>
</tr>
<tr>
<td>17</td>
<td>gpa-8</td>
<td>L4 larvae</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td>daf-2</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>15</td>
</tr>
<tr>
<td>19</td>
<td>daf-2</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>daf-2</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>15 until L4, then 25 overnight</td>
</tr>
<tr>
<td>21</td>
<td>daf-7</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>15 until L4, then 25 overnight</td>
</tr>
<tr>
<td>22</td>
<td>daf-7</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>15 until L4, then 25 overnight</td>
</tr>
<tr>
<td>23</td>
<td>tph-1</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
</tbody>
</table>
Figure A1.S1: Validation experiment using other mutants and conditions

As in Figure A1.3B, the height of each ellipse denotes the SEM of scattering density and the width denotes the SEM of the ORO staining density. All animals imaged on DI H₂O agar pads (as opposed to NGM). N = 16-31 worms (mean 23) per dark field measurement and 12-42 worms (mean 24) per ORO measurement. Descriptions of each condition are given in Table 4 and the text. Linear fit $r^2=0.78$. 
Table 4: conditions and mutants in Figure A1.S1.

<table>
<thead>
<tr>
<th>Number</th>
<th>Genotype</th>
<th>Age</th>
<th>Food</th>
<th>Temperature (°C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><em>daf-2</em></td>
<td>Adult (day 1)</td>
<td>+</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td><em>daf-2</em></td>
<td>Adult (day 2)</td>
<td>+</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td><em>eat-2</em></td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td><em>eat-2</em></td>
<td>Adult (day 2)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td><em>glo-1</em></td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td><em>glo-1</em></td>
<td>Adult (day 2)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td><em>glo-1</em></td>
<td>Adult (day 3)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td><em>glo-4</em></td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td><em>glo-4</em></td>
<td>Adult (day 2)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td><em>glo-4</em></td>
<td>Adult (day 3)</td>
<td>Fasted 24h</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td><em>glo-3</em></td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td><em>glo-3</em></td>
<td>Adult (day 2)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td><em>glo-3</em></td>
<td>Adult (day 3)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>14</td>
<td>N2</td>
<td>Adult (day 2)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>N2</td>
<td>Adult (day 1)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>16</td>
<td>N2</td>
<td>Adult (day 2)</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>17</td>
<td>N2</td>
<td>L4 larvae</td>
<td>+</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td>N2</td>
<td>Adult (day 3)</td>
<td>Fasted 24h</td>
<td>20</td>
</tr>
</tbody>
</table>
We sought to determine the extent to which worm scattering density correlates with fat levels measured by ORO staining. We assayed synchronized worms with a variety of mutations and nutritional states that may affect fat stores (Ashrafi, 2007; Elle et al., 2010; Witham et al., 2016). We found the mean scattering density for a condition to strongly correlate with mean ORO staining density for that condition (Tables 3 and 4, Figures A1.3C and A1.S1).

We included glo-1, glo-3 and glo-4 mutants (Figure A1.S1) because they lack birefringent gut granules (lysosome related organelles, LROs) that are stained by Nile Red and have been mistaken for major fat stores (Hermann et al., 2005; Grill et al., 2007; Rabbitts et al., 2008; O’Rourke et al., 2009). Scattering densities for these mutants were correlated with ORO staining. Moreover, glo-1 and glo-3 day 3 adults yielded some of the highest measurements for both scattering and ORO staining, suggesting that birefringent gut granules are not principally responsible for light scattering in the adult worm intestine.

These results demonstrate that under most conditions, light scattering density is strongly correlated with ORO staining density in C. elegans.

**Scattering measurements reveal a rapid increase in fat content after the first larval stage**

Since we observed a large increase in both fat storage and scattering density between L4 and adult worms (points 3 and 6 in Figure A1.3C), we asked how fat content changes during development. Dark field imaging is compatible with live worm imaging and requires very little time compared to traditional staining or biochemistry procedures, making it well suited for gathering data with high temporal resolution.

By imaging groups of developing larvae culled at various times, we found that scattering density increases non-uniformly between 4 and 50 hours after re-feeding the animals. In
particular, we observed that during the first 24 hours of development, which corresponds primarily to the L1 stage, scattering density only increased by a small amount. Starting at the L1-L2 transition, however, a rapid increase in fat levels occurs for about seven hours, followed by a period of modest and variable increase in the L3 stage (Figure A1.4).
Figure A1.4. Scattering density increases sharply between the L1 and L2 larval stages

(A) Scattering density as a function of time after re-feeding L1 arrested larvae. Each point represents a single worm. Scattering density is normalized to the density at time t=4hr. N=11-16 worms per point (mean 15). Blue points represent data acquired through a 40X objective. Orange points represent data acquired through a 20X objective.

(B) Mean worm length at each time point. The approximate times of the L1-L2 and L2-L3 larval transitions, estimated by measurements of worm length, are indicated by vertical bars.
To confirm that young worms store very little fat, we also stained worms from various larval stages with ORO (Figure A1.S3). Indeed, L1 worms accumulated almost no ORO, with significant accumulation beginning only in later larval stages.

**Figure A1.S3**: Oil-Red-O stained, well-fed N2 worms at various larval stages.

Stages were determined by worm length and vulval morphology. Scale bar is 50 µm.
Taken together, these results show that fat storage rapidly accelerates during or after the L1-L2 larval transition, but remains approximately constant during the L1 and L3 stages.
Light scattering in first stage larvae is dominated by gut granules

We sought to determine which cellular or intracellular structure(s) within the worm were primarily responsible for light scattering during development. We observed that dark field images of L1 larvae, unlike those of adult worms, featured small, bright puncta surrounded by dark areas. The discrete nature of these puncta, and the low scattering density of L1 worms (Figure A1.4), suggested that these objects were gut granules (lysosome-related organelles, or LROs), not fat stores. To test this hypothesis, we stained larvae at several larval stages with Nile Red, which stains gut granules and not lipid droplets (O’Rourke et al., 2009). We then imaged Nile Red stained worms and unstained control worms for both scattering and red fluorescence (Figure A1.5A).

We observed that the bright puncta in dark field and fluorescence images of the L1 gut were highly co-localized. However, for older larvae the similarity between these images decreased as puncta become overshadowed by a more spatially uniform scattering in the gut. We found that the quantitative degree of overlap between bright pixels in dark field and Nile Red images, as measured by the Sorenson-Dice coefficient QS, decreased during development (Figure A1.5B). The steepest drop in Nile Red-dark field co-localization occurred between 24 and 30 hours post re-feeding, corresponding to the sharp increase in scattering density near the L1-L2 transition (Figure A1.4).
Figure A1.5. Scattering in L1 worms corresponds to gut granules; scattering in older worms does not.

(A) Representative dark field and Nile Red images from individual worms of all four larval stages, acquired through a 63x objective. Images were manually segmented to isolate the worm. The merge images show the overlap between the brightest pixels in each image. Red corresponds to dark field, green to Nile Red fluorescence, and yellow to overlapped regions. The corresponding Sorenson-Dice Coefficient (QS) value is shown at right. All scale bars are 20 µm.

(B) Mean QS, the average overlap between bright pixels in dark field and Nile Red images, decreases as a function of age. QS was also computed for control worms that were unstained, but imaged under the same fluorescence parameters (not shown in A). N=9-13 worms per point; error bars represent SEM.

(C) High resolution dark field and Nile Red images of an adult (69 hr) intestine. Layers of lipid droplets are plainly visible in the dark field image, and most do not co-localize with LROs. Similar droplets are also visible in some L2 and L3 dark field images (see A), and in mutants lacking LROs (see Figure A1.S2). White arrows, droplet that does not colocalize with Nile Red. Red arrows, droplet that colocalizes with a non-puncta Nile Red signal.
To confirm that dark field images of L1 larvae show gut granules, we obtained dark field and Nile Red fluorescence images of stained L1 glo-1 worms, in which these gut granules are missing (Hermann et al., 2005). Images of these larvae lacked bright puncta in the gut (Figure A1.S2A), confirming that dark field images of L1 larvae are dominated by gut granules. Interestingly, glo-1 mutants did stain for Nile Red in the gut, although bright puncta (LROs) were not visible.

Figure A1.S2: Droplets visible in dark field images are not LROs

(A) Dark field and Nile Red images of L1 larvae. Note glo-1 mutants are missing gut granules. In N2 larvae, Nile Red staining gut granules co-localize with scattering puncta. See also: Figure A1.5. (B) High magnification dark field and fluorescence images of Nile Red stained adults. Individual droplets are visible in dark field images of wild-type and
glo-1 adults. Gut granules are visible in wild-type worms only. Scale bar for both A and B is 20 µm. Consistent gray scales are used for each image type.
Individual lipid droplets are visible in dark field images

Our high-resolution dark field images of worm intestines, especially those of adults, often revealed readily discernable small circular droplets (Figures A1.5C and A1.S2B). Most of these droplets did not co-localize with LROs stained by Nile Red (Figure A1.5C). These circular droplets were also plainly visible in LRO-lacking glo-1 mutants. To our surprise, we found that many of droplets in the glo-1 intestine were weakly stained by Nile Red despite the absence of bright LRO puncta in this mutant. This observation suggests that scattering droplets that co-localized with a dim, non-puncta Nile Red circle (Figure A1.5C) are also not LROs.

In dark field images, the edges of each droplet appeared brighter than the interior, indicating that the scattering occurred at boundaries of these droplets. Most scattering in soft tissues occurs at the boundaries between lipid droplets and their aqueous surroundings (Jacques and Prahl, 1998), suggesting that these objects are individual lipid droplets.

Scattering density decreases during worm starvation may not correlate with activity

We expected fat levels to decrease in the absence of food due to animals’ expenditure of energy stores to meet metabolic demands. Indeed, we observed that mean scattering density decreased during periods of starvation (Figure A1.3C, points 7 and 9). We asked whether the amount of fat lost during these periods is correlated with worm locomotory activity levels. Dark field imaging is well suited for addressing this question, since it allows multiple measurements to be made on the same animal.
We imaged adult animals before and after 18 hours of fasting. During starvation, we recorded low-magnification videos of all animals to assess their activity levels (see methods). In 54 of 55 animals, we observed decreases in scattering density. However, no correlation emerged between average activity level and the amount of fat loss (Figure A1.S4). These results are consistent with a calculation based on allometric scaling suggesting that the worm’s power expenditure associated with locomotion represents a very small fraction of its overall metabolic rate (Lee, 2002).

Figure A1.S4. Fat loss is not correlated with prior activity level.

The relative change in scattering density ($\Delta S/S_0$) after 18 hours of fasting vs. the mean activity level. Each worm is represented by one point; 55 worms were analyzed in total. Points with the same color (red, blue, or black) belong to the same experiment.
DISCUSSION

We sought to develop a fat measurement technique that provides readings from live worms with high spatiotemporal resolution, is readily scalable to individual or large groups of worms, and is technically simple to implement. Dark field imaging relies on a simple and inexpensive setup compatible with any optical microscope. Its simple and non-invasive nature allows fat levels and distribution to be rapidly estimated with high spatial and temporal resolution in groups of growing worms (Figures A1.4-5), and in individual animals before and after a treatment on a per-animal basis (Figure A1.S4).

Our results show that scattering density can be used to estimate relative fat levels in *C. elegans*. We defined a simple and intuitive metric that can be semi-automatically computed from worm images, and showed that it correlates with relative levels of ORO staining across a wide range of conditions (Figures A1.3 and A1.S1). These results do not necessarily indicate the maximum linear range of the technique, since we did not observe saturation of the signal at either the high or low ends. Saturation probably did not occur because imaging metrics were used for both measurements; the same non-fatty tissue that provides a low baseline of scattering may also provide a low baseline of 510 nm absorbance. Nonetheless, the dynamic range over which scattering density is linearly correlated with ORO staining density, and thus fat levels, is at least as broad as the difference between a fasted worm and a high fat mutant. We did, however, observe temporal saturation in the dark field measurements of young larvae, which do not appear to store any fat at all until the second larval stage (Figure A1.4A).

For many scattering measurements shown in Figure A1.3B-C, the sample size of ~15 worms was sufficient to constrain the SEM to less than 10%. Other measurements had
errors of almost 20%, indicating substantial uncertainty in the mean. However, similar variation in error was also observed for the ORO measurements, making it unclear how much error results from the measurements themselves and how much error results from true variation in the population.

We found that scattering density is a valid approximation of ORO staining density in worms with substantial levels of fat stores, and is useful for measuring increases or decreases in fat storage. Scattering density may not be appropriate for comparing fat levels between two conditions when the worms have very few lipid stores in both conditions (e.g. first stage larvae). Our method may also not be suitable for assessing very small differences in fat levels.

Our study indicates that light scattering in the embryo and adult gut is principally due to lipid droplets and not Nile Red staining lysosome-related organelles (O’Rourke et al., 2009). The scattering densities of mutants lacking these lysosome-related organelles still correlated well with ORO staining levels during periods of starvation or growth, as shown in Figures A1.3 and A1.S1. Nile Red, which stains LROs, often fails to decrease in fluorescence during starvation. Moreover, individual lipid droplets that did not co-localize with LROs were readily visible in high-resolution dark field images (Figures A1.5C and A1.S2).

Because dark field imaging is technically simple, we were able to image worms during development at much higher temporal resolution than has been reported using CARS microscopy (Hellerer et al., 2007). The CARS data indicated that the volume fraction of lipids in developing worms approximately doubles sometime between the L1 and L2 stages, a finding supported by our measurements of dark field scattering density (Figure
However, our results reveal that fat storage does not increase uniformly during that period; rather, a rapid burst in fat storage appears to take place near the L1-L2 transition, perhaps even during lethargus. The CARS data also suggests that the volume fraction of lipids in wild type worms decreases after the L2 stage and into adulthood. Neither our ORO staining data (Figures A1.3, A1.S1, and A1.S3), nor our scattering data (Figures A1.3, A1.4, and A1.S1) matches this result. At least one other report of ORO staining in developing larvae also suggests that fat stores increase between the L3 and L4 stages (Yen et al., 2010).

Dark field imaging augments the toolbox of fat measurement techniques by dramatically reducing the cost and expertise needed to measure fat levels in live worms. We anticipate that it will be particularly useful for experiments in which fat levels need to be tracked in individual animals over time.

**ACKNOWLEDGEMENTS**
We thank Jianting Wang and Yu Chen of the University of Maryland, College Park, for providing the scattering phantom raw material. Some strains were provided by the CGC, which is funded by NIH Office of Research Infrastructure Programs (P40 OD010440). A. D. F. was supported by the National Institutes of Health (5R01NS084835-03). D. M. R. was supported by the National Institutes of Health (R01NS088432 and R21NS091500). C. F.-Y. was supported by the National Institutes of Health (5R01NS084835-03), Ellison Medical Foundation, and the Alfred P. Sloan Research Foundation.
APPENDIX II: C++ SOFTWARE FOR THE OPTOGENETIC TARGETING SYSTEM

This appendix contains the C++ software that drives the optogenetic targeting system described in Chapters 1 and 2. The original software files, along with compilation instructions in Microsoft Visual Studio 2012, are also available from an online repository (Fouad et al., 2018a). This software was written entirely by me, except for small sections adapted or replicated from other sources with attribution.

StageBoost.cpp

/*
  * StageBoost.cpp
  * Anthony Fouad
  * Fang-Yen Group, developed and used 06/2014 - 11/2017
  * 
  * Optogenetic control and fluorescence+darkfield imaging of freely moving C. elegans.
  *     - Stage controlled using the BOOST libraries implementation of Serial.
  *     - Camera control and image processing achieved using OPENCV.
  *     - sCMOS fluorescence camera (Qimaging optiMOS) controlled using PVCAM SDK 3.0
  *     - DMD control to illuminate worm regions achieved using ALP 4.1
  * 
  * Stageboost - Region specific laser targeting of freely moving C. elegans
  * Written in 2014-2018 by Anthony Fouad et al <afouad@seas.upenn.edu>
  * 
  * To the extent possible under law, the author(s) have dedicated all copyright and related
  * and neighboring rights to this software to the public domain worldwide. This software is
* distributed without any warranty.
* 
* You should have received a copy of the CC0 Public Domain Dedication
along with this software.
* If not, see <http://creativecommons.org/publicdomain/zero/1.0/>
* 
* StageBoost is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* CC0 License for more details.
* 
* NOTE: If you use any portion of this code in your research, kindly
cite:
* 
* Anthony D. Fouad et al., eLife 7, e29913
(2018).
* 
*/

*/

// Standard include files
#include "stdafx.h"
#include <thread>
#include <iostream>

//OpenCV includes
#include "opencv2/highgui/highgui.hpp"

//Boost includes
#include "boost\thread.hpp"

//PVcam includes
#include "master.h"
#include "pvcam.h"
#include "Common.h"
#include "PVcamAccessoryFuncs.h"

//Anthony's includes
#include "SimpleSerial.h"
#include "AnthonysCalculations.h"
#include "AnthonysColors.h"
#include "AnthonysCamFuncs.h"
#include "AnthonysTimer.h"
#include "StageAccessoryFuncs.h"
#include "DlpAccessoryFuncs.h"
#include "Images.h"

using namespace std;

int main(int argc, char** argv)
{
   /*----------------------Begin startup sequence----------------------*/

   // Load the script instructions, which define the experiment. Return if invalid script loaded. Load a different script if requested.
   char loadother[100] = "y";
   struct Script* pScript;

   while(strchr (loadother, 'y')>0){
      system("CLS");
      pScript = import_script();
      int ps = print_script(pScript);
      if(ps<1){printf("nINVALID SCRIPT DETECTED\n\n"); system("PAUSE"); return -9;}
      cout << "Reload the script? (y/n)" << endl;
      cin >> loadother;
   }

   // Select targeting scheme to use
   system("CLS");
   cout << "Which targeting scheme to use?\n"
   << "0 - global illumination\n"
   << "1 - Simple puncta\n"
   << "21 - Pharynx (either bulb)\n"
   << "22 - Pharynx (either edge)\n"
   << "23 - Pharynx (either edge + wings)\n"
   << "24 - Pharynx (relative position)\n"
   << "25 - Pharynx (Entire pharynx box)\n"
   << "26 - Pharynx (one edge + wings + all non pharynx on)\n"
   << "31 - Darkfield (longitudinal segment)\n"
   << "4 - blind dummy puncta\n";

   int tracktypeflag = pScript[0].arg[1];
   cout << "\nSpecified in script: " << tracktypeflag << endl;

   //boost::this_thread::sleep(boost::posix_time::milliseconds(2000))
};
if(!is_tracktype(tracktypeflag,17,0,1,11,12,21,22,23,24,25,26,27,
28,31,32,33,34,4,51,52))
{
    tracktypeflag = 1;
    printf("Incorrect type of tracking selected. Stageboost will now exit...
\n\n");
    system("PAUSE");
    return -99;
}

// Initialize setup
system("CLS");
printf("Starting initialization sequence\n");
printf("--------------------------------\n");

// Select the calibration data to use (surface fit coefficients)
    int objective = 20;
    char dlpcalibfile[300];

    sprintf(dlpcalibfile,"YourDirectory/StageBoost/Parameters/DlpCalibrationValsSurf.txt");

    // Setup the PVcam
    int PVCAM_ACQUIRE_MODE = PVCAM_ACQUIRE_MODE_DEF; // -1 use imagingsource, 0=standard , 1=CircularOverwrite , 2=CircularNoOverWrite
    int exp_time = pScript[0].arg[0];
    // The first entry in the script must be the exposure time of the PVCAM
    rgn_type region = { 0, 999, 3, 0, 999, 3 };
    region.p1 = pScript[1].arg[0];
    // The second entry in the script must be the region definition.
    region.p2 = pScript[1].arg[1];
    region.pbin = pScript[1].arg[2];
    region.s1 = pScript[1].arg[3];
    region.s2 = pScript[1].arg[4];
    region.sbin = pScript[1].arg[5];
    verify_region(&region);
    uns16* circBufferInMemory = new uns16[CIRC_BUFF_FRAMES*((region.s2+1)/region.sbin *
    (region.p2+1)/region.pbin)];
    /* Most PVcam setup occurs inside Images.cpp>
CreateImageStruct */

    // Setup some variables by loading a script
    int key;
char command[10];
rs_bool pvstatus = 0;
struct Images* pImg1 = CreateImageStruct(circBufferInMemory,region,&pvstatus,exp_time,PVCAM_ACQUIRE_MODE,640,480);  // Structure holds most data about the image and segmentation
struct DlpCalib* pDlp1 = CreateDlpCalibStruct(dlpcalibfile);
pImg1->NumSteps = pScript[0].arg[2];
pImg1->wingPct = pScript[0].arg[3];

//if(pvstatus == 0){return -99;}

// Setup the framerate to a custom value?
if(tracktypeflag==12){
pImg1->FrameRateFixed = 18;
}

// Record which objective and tracking scheme are in use
pImg1->objective = objective;
pImg1->tracktypeflag = tracktypeflag;

// Setup the stage serial control
printf("Attempting establish stage serial port control...\n");
SimpleSerial mySerial("COM5",9600);

Establish serial connection
printf("Successfully established stage serial port control.\n\n");

// Communicate with the darkfield camera
printf("Attempting to communicate with USB camera(s)...\n");
if(PVCAM_ACQUIRE_MODE_DEF<0){
pImg1->Cam2ID = find_system_cameras(-1);
pImg1->CamID = find_system_cameras(pImg1->Cam2ID+1); }
else{
pImg1->CamID = find_system_cameras(1); // -1 if webcam disconnected
}

// Load GUI image (for displaying system status only)
printf("Attempting to load the GUI interface template...\n");
pImg1->StatusImgBlank = cvLoadImage("YourDirectory/StageBoost/UI5.png",1);
if(pImg1->Status ImgBlank == 0){ printf("FAILURE TO LOAD USER INTERFACE 'UI5.png\n\n\n"); system("PAUSE"); return -99; }

printf("Successfully loaded the GUI interface template.\n\n\n");

// Load background subtraction image (for darkfield tracking, and only if the size matches)
if(1){
    if(exists("temp_BGavg.png")){
        IplImage *tempimg = cvLoadImage("temp_BGavg.png",CV_LOAD_IMAGE_ANYDEPTH | CV_LOAD_IMAGE_ANYCOLOR);
        if(tempimg->width == pImg1->BGavg->width && tempimg->height == pImg1->BGavg->height){
            cvCopy(tempimg,pImg1->BGavg);
            cvCvtColor(pImg1->BGavg,pImg1->BGavgColor,CV_GRAY2RGB);
        }
    }
}

// Setup the ALP/DMD micromirror system
printf("Attempting connect to the ALP mirror system...\n\n\n");
pDlp1->alpid = T2DLP_on();
if(pDlp1->alpid == -1){printf("------FAILED TO CONNECT TO ALP.\n\n\n");} else{printf("Successfully connected to the ALP mirror system...\n\n\n");} 

// Always start the focus point at 1/3 of the length, if applicable
pImg1->FocusPtIdx = pImg1->NumSteps * 33/100;
pImg1->StartPtIdx = pImg1->NumSteps * 0/100;

printf("---------------------------\n\n\n");
printf("Initialization sequence completed.\n\n\n");

system("PAUSE");

/**************End startup sequence**************/

/**************Begin main menu loop**************/

system("cls");
cout << "Welcome to Stageboost 1.0 by Anthony Fouad!" << endl;
int whilect = 0;

while(1){
    whilect+=1; if(whilect>25){printf("Exiting due to failsafe counter\n"); break;}

    cout << "Main Menu (please enter command):\n";
    cout << "c : view camera\n"
        << "  \t: record frames to disk\n"
        << "  o : overlay segmentation\n"
        << "  a : automate stage\n"
        << "  \tf : show fluorescence\n"
        << "  \tt : target worm with DLP\n"
        << "  \tz : target worm with DLP script (overrides 't')\n"
        << "  \tv : validation image saving\n"
        << "b : Find background location\n"
        << "p : set PVCAM parameters\n"
        << "d : calibrate DLP - to - image\n"
        << "u : verify DLP calibration\n"
        << "p : Stage console\n"
        << "l : Calibrate stage - to - pixel size\n"
        << "i : Display a sample image on the DLP\n"
        << "e : exit\n";

    cin >> command;

    // Reset defaults on each loop
    pImg1->OlFlag = 1; // By default, do not overlay segmentation
    pImg1->AutoFlag = 0; // By default, do not automate the stage
    pImg1->FluoFlag = 0; // By default, do not show the fluorescence (PVCAM) frame
    pImg1->RecordFlag = 0; // By default, do not save images to disk.
    pImg1->DlpFlag = 0; // By default, do not aim the DLP at the worm.
    pImg1->targetFlag = 1; // By default, target with the fluorescence image.
    pImg1->scriptFlag = 0; // By default, do not run any targeting scripts.
pImg1->startExptFlag=0; // By default, do not run the experimental script or stage automation. They will be run after the user selects a tracking point.
pImg1->startAutoFlag=0; // By default, don't start automation right away. Wait until user clicks or right clicks.
pDlp1->dlpscriptstep=2; // By default, the DLP script starts at step 2, since 0 and 1 are setup steps.
pDlp1->dlptime = 0; // By default, set the DLP timer at 0 to begin with.
pDlp1->dlprepeatcount = 0; // By default, start all the way at the beginning of the script

// Search within the input string to parse the action that the user wants
if (strchr (command, 'e')>0){break;}
if (strchr (command, 'o')>0){pImg1->O1Flag=1;}
if (strchr (command, 'a')>0){pImg1->AutoFlag=1;}
if (strchr (command, 'f')>0){pImg1->FluoFlag=1;}
if (strchr (command, 'r')>0){pImg1->RecordFlag=1;}
if (strchr (command, 't')>0){pImg1->DlpFlag=1;}
if (strchr (command, 'z')>0){pImg1->scriptFlag=1;}
if (strchr (command, 'v')>0){pImg1->writeImgFlag=1;}
if (strchr (command, 'i')>0){displayDlpSample(pDlp1->alpid,1);
    system("cls");  continue;}
if (strchr (command, 'd')>0){calibrateDLP(pImg1,pDlp1,dlpcalibfile);
    system("cls");  continue;}
if (strchr (command, 'u')>0){verifyDlpCalib(pImg1,pDlp1);
    system("cls");  continue;}
if (strchr (command, 'l')>0){StageCalibrate(pImg1,&mySerial);
    system("cls");  continue;}
if (strchr (command, 'b')>0){CamBG(pImg1);
    system("cls");  continue;}
if (strchr (command, 'c')>0){Cam(pImg1,&mySerial,pDlp1,pScript);
    system("cls");  continue;}
if (strchr (command, 'p')>0){StageCommand(&mySerial);
    system("cls");  }

// Reset loop
printf("BeforeCLS\n");
system("cls");
Blank_DLP(pImg1,pDlp1);
/*-------------------End main menu loop-------------------*/

/*-------------------Begin shutdown sequence-------------------*/
system("cls");
printf("Shutting down...
");
printf("----------------------------------------
");
    // Disconnect the ALP/DMD
    T2DLP_off(pDlp1->alpid);
    // Close down PVcam
    delete[] circBufferInMemory;
    CloseCameraAndUninit(pImg1->hCam);

/*-------------------End shutdown sequence-------------------*/
return 0;
}

AnthonysCalculations.h
#ifndef ANTHONYSCALCULATIONS_H_
#define ANTHONYSCALCULATIONS_H_

void isolate_largest_object( IplImage* img_in, IplImage* img_out, CvMemStorage* storage);
void subtract_fitc_mcherry_profile(struct Images* pImg1);
void hollow_filter( struct Images* pImg1, CvMat* ker);
void round(double *a);
int sign(double a);
void midpoint(CvPoint2D32f* A, CvPoint2D32f* B, CvPoint2D32f* MP);
int linspace(int x0, int x1, int N, int n);
void divByScalar(IplImage *src, IplImage *dest, double divVal);
void mulByScalar(IplImage *src, IplImage *dest, IplImage *temp, double mulVal);
void subFromScalar(IplImage *src, IplImage *dest, double subVal);

173
#endif

AnthonysCalculations.cpp
/*
*      AnthonysCalculations.cpp
*      Anthony Fouad
*      Fang-Yen Group, 6/2014
*
*      Various necessary calculations, ranging from simple
*      mathematical operations to whole segmentation of the worm
*/

#include "stdafx.h"
#include "stdlib.h"
#include "stdio.h"
#include <iostream>
#include <cmath>

// OpenCV 2.4 functions
#include "opencv\cv.h"
#include "opencv\highgui.h"

// Anthony's functions
#include "AnthonysTimer.h"
#include "AnthonysColors.h"
#include "Images.h"
#include "AnthonysCamFuncs.h"

using namespace std;
using namespace cv;

/*
 * Remove small blobs from the binary image, leaving only the worm
 * Adapted from:
 * http://4imageprocessing.blogspot.com/2009/05/opencv4-remove-small-blobs-from-binary.html
 */

void isolate_largest_object(IplImage* img_in, IplImage* img_out, CvMemStorage* storage) {
    cvCopy(img_in, img_out); // return image
    CvSeq* contours = NULL;
    double area;
    double maxArea = 1;
    int ct = 0;

    // STEP 1: Find all contours in binary image
    cvFindContours(img_in, storage, &contours, sizeof(CvContour), CV_RETR_LIST, CV_CHAIN_APPROX_SIMPLE);

    // STEP 2: Get the largest size object in the image.  While loop over all the contours
    while (contours) // loop over all the contours
    {
        ct = ct + 1;
        area = cvContourArea(contours, CV_WHOLE_SEQ);
        if (fabs(area) > maxArea)
        {
            maxArea = fabs(area);
        }
        contours = contours->h_next; // jump to the next contour
    }

    // STEP 3: preserve only the biggest object
    ct = 0;
    cvFindContours(img_in, storage, &contours, sizeof(CvContour), CV_RETR_LIST, CV_CHAIN_APPROX_SIMPLE);
    while (contours) // loop over all the contours
    {
        ct = ct + 1;
        area = cvContourArea(contours, CV_WHOLE_SEQ);
        if (fabs(area) > maxArea)
        {
            maxArea = fabs(area);
        }
        contours = contours->h_next;
    }
}
if( fabs(area) < maxArea )  // if the area of the contour is less than threshold remove it
{
    // Remove the contours by drawing in black
    cvDrawContours( img_out, contours, kk, kk, -1, CV_FILLED, 8 ); // removes white areas
}  
else  
{
    // If this is the big object, Fill in holes
    cvDrawContours( img_out, contours, ww, ww, -1, CV_FILLED, 8 ); // fills in holes
}  
contours = contours->h_next;  // jump to the next contour

return;

*/

/*
   When Worm-In-FITC systems are used for validation, the FITC's red components bleed into the mCherry channel and mess up the targeting. This function subtracts the illumination box from the image.
*/

void subtract_fitc_mcherry_profile(struct Images* pImg1){

    // Prepare the subtraction image
    cvZero(pImg1->FluoImgGray8bitTemp);
    int subval = 10;
    cvRectangle4pt(pImg1->FluoImgGray8bitTemp,pImg1->max_wing1,pImg1->max_wing2,pImg1->max2_wing1,pImg1->max2_wing2,cvScalar(subval,subval,subval),-1);

    // Subtract the subtraction image from the segmentation image
    cvSub(pImg1->FluoImgGray8bit,pImg1->FluoImgGray8bitTemp,pImg1->FluoImgGray8bit);

}

/*
   * Apply Anthony's hollow filter to an pImg1->MainImgGray, accentuating large bright objects. The
*/
* kernel ker should be saved to disk by matlab, or otherwise supplied to the function
* /

void hollow_filter( struct Images* pImg1, CvMat* ker){
    cvFilter2D(pImg1->MainImggray, pImg1->MainImggray, ker);
}

/*/ 
* round a number a. 
*/

void round(double *a){
    *a = floor(*a+0.5);
}

/*/ 
* return the sign (+1 or -1) of number a 
*/

int sign(double a){
    if(a>0){return 1;}
    else {return 0;}
}

/*/ 
* Calculates the midpoint between 2 points 
*/

void midpoint(CvPoint2D32f* A, CvPoint2D32f* B, CvPoint2D32f* MP){
    *MP = cvPoint2D32f( (A->x+B->x)/2 , (A->y+B->y)/2);
}

/*/ 
* linspace (similar to the same command in Matlab) 
* 
* The output xn is the nth point along the vector from x0 to x1 with N subunits 
* see: linspace_Cstyle. 
* 
* setting n = 1 means that the output is simply x0 
* setting n = N means that the output is simply x1 
*/

int linspace(int x0, int x1, int N, int n){
    double dx = ((double) x1 - (double) x0) / ((double) N - 1);
}
double xnd = x0 + dx*(n-1);
round(&xnd);
int xn = (int) xnd;
return xn;
}

/*
* Divide an image by a scalar value. From example online:
*/

void divByScalar(IplImage *src, IplImage *dest, double divVal) {
/*we use this for the division*/
IplImage* div= cvCreateImage(
    cvSize(src->width, src->height),
    src->depth,
    src->nChannels );

/*this sets every scalar to divVal*/
cvSet( div, cvScalar(divVal,divVal,divVal), NULL);
cvDiv( src, div, dest, 1 );
}

/*
* Multiply an image by a scalar value. From example online:
*/

void mulByScalar(IplImage *src, IplImage *dest, IplImage *temp, double mulVal) {
/*this sets every scalar in the temporary image to divVal*/
cvSet( temp, cvScalar(mulVal,mulVal,mulVal), NULL);

/* Now multiply the temp image by the source image */
cvMul( src, temp, dest, 1 );
}

/*
* Add an image to a scalar value. Adapted From example online:
*/
void subFromScalar(IplImage *src, IplImage *dest, double subVal) {
    /*we use this for the operation*/
    IplImage* subimg= cvCreateImage(
        cvSize(src->width, src->height),
        src->depth,
        src->nChannels
    );

    /*this sets every scalar to divVal*/
    cvSet( subimg, cvScalar(subVal,subVal,subVal), NULL);
    cvSub( subimg, src , dest );

    /* release the image */
    cvReleaseImage(&subimg);
}

/*
 * Normalize a uint8 grayscale image.
 */
void normalize2uint8(IplImage *src, IplImage*dest, IplImage *temp,
                      IplImage *mask){
    double max, min;
    CvPoint max_loc, min_loc;
    // STEP 1: Find the minimum value of src, but exclude
    cvMinMaxLoc(src,&min,&max,&min_loc,&max_loc);
// STEP 2: Subtract the minimum value away
cvSet(temp,cvScalar(min,min,min));
cvSub(src,temp,src);

// STEP 3: Find the new maximum
cvMinMaxLoc(src,&min,&max,&min_loc,&max_loc,mask);

// STEP 4: Divide by the new maximum (normalizes to uint16)
mulByScalar(src,dest,temp,255.0/max);

void normalize2uint16(IplImage *src, IplImage*dest, IplImage *temp, IplImage *mask){

    // Enforce 16 bit inputs, except for mask which must be uint8
    if(src->depth != 16 || dest->depth!=16 || temp->depth != 16 ||
      mask->depth != 8){
        printf("Incorrect depth image supplied to normalize2uint16. Skipping...\n");
        return;
    }

    // Enforce grayscale image input requirement
    if(src->nChannels != 1 || dest->nChannels != 1 || temp->
      nChannels != 1 || mask->nChannels != 1 ){
        printf("Incorrect (probably colored) image supplied to normalize2uint16. Skipping...\n");
        return;
    }

    // Declare variables
    double max, min;
    CvPoint max_loc, min_loc;

    // STEP 1: Find the minimum value of src, but exclude border
    // regions
    cvMinMaxLoc(src,&min,&max,&min_loc,&max_loc);

    // STEP 2: Subtract the minimum value away
    cvSet(temp,cvScalar(min,min,min));
cvSub(src,temp,src);
// STEP 3: Find the new maximum
cvMinMaxLoc(src,&min,&max,&min_loc,&max_loc,mask);

// STEP 4: Divide by the new maximum (normalizes to uint16)
mulByScalar(src,dest,temp,65535.0/max);
}

// Distance between two floating point CvPoint2D32f's
double dist32f(CvPoint2D32f A,CvPoint2D32f B){
    double x1 = (double) A.x;
    double x2 = (double) B.x;
    double y1 = (double) A.y;
    double y2 = (double) B.y;

    double d = sqrt(pow(x1-x2,2) + pow(y1-y2,2));
    return d;
}

// Distance between two integer point CvPoints
double dist(CvPoint A,CvPoint B){
    double x1 = (double) A.x;
    double x2 = (double) B.x;
    double y1 = (double) A.y;
    double y2 = (double) B.y;

    double d = sqrt(pow(x1-x2,2) + pow(y1-y2,2));
    return d;
}

// Theta between two vectors, VecA and VecB, computed by way of the
// dot product
float fPointDotTheta(CvPoint2D32f* VecA, CvPoint2D32f* VecB){
    // Return the angle between two normalized (starting at 0) vectors
    CvPoint2D32f RefPt= cvPoint2D32f(0,0);
    float dot = ((VecA->x)*(VecB->x) + (VecA->y)*(VecB->y));
    float norm = dist32f(RefPt,*VecA) * dist32f(RefPt,*VecB);
    return acos(dot/norm);
}

/*
 * ANDY'S FUNCTION:
 * Calculate distance between two points on a loop (e.g. contour bound)
 */
int DistBetPtsOnCircBound(int PtsOnBound, int IndexA, int IndexB) {
    int distance;
    if (labs(IndexA-IndexB)<labs(PtsOnBound- labs(IndexA-IndexB))){ // go with the smallest value
        distance=labs(IndexA-IndexB);
    } else{
        distance=labs(PtsOnBound- labs(IndexA-IndexB));
    }
    return distance;
}

/*
* ANDY'S FUNCTIONS:
* Smoothen contours with a gaussian kernel
*/

void ConvolveInt1D (const int *src, int *dst, int length, int *kernel, int klength, int normfactor) {
    int j, k, ind, anchor, sum;
    anchor = klength/2;
    for (j = 0; j < length; j++) {
        sum = 0;
        for (k = 0; k < klength; k++) {
            ind = j + k - anchor;
            ind = ind > 0 ? ind : 0;
            ind = ind < length ? ind : (length - 1);
            sum = sum + src[ind]*kernel[k];
        }
        dst[j] = (int) (1.0*sum/normfactor + 0.5);
    }
}

void ConvolveCvPtSeq (const CvSeq *src, CvSeq *dst, int *kernel, int klength, int normfactor) {
    int j, *x, *y, *xc, *yc;
    CvPoint pt;
    x = (int *) malloc (src->total * sizeof(int));
    y = (int *) malloc (src->total * sizeof(int));
    xc = (int *) malloc (src->total * sizeof(int));
    yc = (int *) malloc (src->total * sizeof(int));
    for (j = 0; j < src->total; j++) {
        x[j] = ((CvPoint *) cvGetSeqElem(src, j))->x;
        y[j] = ((CvPoint *) cvGetSeqElem(src, j))->y;
    }
    ConvolveInt1D(x, xc, src->total, kernel, klength, normfactor);
ConvolveInt1D(y, yc, src->total, kernel, klength, normfactor);

for (j = 0; j < src->total; j++) {
    pt.x = xc[j];
    pt.y = yc[j];
    cvSeqPush(dst, &pt);
}

free(x);
free(y);
free(xc);
free(yc);

void CreateGaussianKernel (double sigma, int **kernel, int *klength, int *normfactor) {
    int ll, ul, x;
    double n;
    ll = (int) (-3 * sigma) - 1;
    ul = (int) (3 * sigma) + 1;
    *klength = ul - ll + 1;
    *kernel = (int*) malloc (*klength * sizeof(int));
    *normfactor = 0;
    n = exp(-1.0*ll*ll/(2*sigma*sigma));
    for (x = 0; x < *klength; x++) {
        (*kernel)[x] = (int) (exp(-1.0*(x+ll)*(x+ll)/(2*sigma*sigma))/n + 0.5);
        *normfactor += (*kernel)[x];
    }
}

void smoothPtSequence (const CvSeq *src, CvSeq *dst, double sigma) {
    cvClearSeq(dst);
    int *kernel, klength, normfactor;
    CreateGaussianKernel(sigma, &kernel, &klength, &normfactor);
    ConvolveCvPtSeq(src, dst, kernel, klength, normfactor);
    free(kernel);
}

/*
 * Force consistency in head tail segmentation by making the head close to the last frame's head
 */
void checkHeadTail(struct Images* pImg1){
// Get the distances between the head ref and the current head and the current tail.
    double d_head_headlast = dist32f(cvPoint2D32f(pImg1->head->x,pImg1->head->y),cvPoint2D32f(pImg1->headLast->x,pImg1->headLast->y));
    double d_head_taillast = dist32f(cvPoint2D32f(pImg1->tail->x,pImg1->tail->y),cvPoint2D32f(pImg1->headLast->x,pImg1->headLast->y));

    // If head ref is closest to the current tail, better flip the current head/tail
    if(d_head_taillast<d_head_headlast){
        CvPoint TEMPPT = cvPoint(pImg1->head->x,pImg1->head->y);
        double TEMPIDX = pImg1->tailindex;
        *pImg1->head = *pImg1->tail;
        *pImg1->tail = TEMPPT;

        pImg1->tailindex = pImg1->headindex;
        pImg1->headindex = TEMPIDX;
    }

    // After checking / flipping, set the new headLast and tailLast values;
    *pImg1->headLast = *pImg1->head;
    *pImg1->tailLast = *pImg1->tail;
}

/*
 * Manually flip the head and tail, if specified. Also flip the pharynx bulbs
*/

void manualHeadTailFlip(struct Images* pImg1){

    // Flip head and tail (if using darkfield)
    CvPoint TEMPPT = cvPoint(pImg1->head->x,pImg1->head->y);
    double TEMPIDX = pImg1->tailindex;
    *pImg1->head = *pImg1->tail;
    *pImg1->tail = TEMPPT;

    pImg1->tailindex = pImg1->headindex;
    pImg1->headindex = TEMPIDX;

    // After checking / flipping, set the new headLast and tailLast values;
    *pImg1->headLast = *pImg1->head;

*pImg1->tallass = *pImg1->tail;

// Flip terminal bulb and corpus (if using pharynx)
TEMPPT = cvPoint(pImg1->ph_met->x,pImg1->ph_met->y);
*pImg1->ph_ter = *pImg1->ph_met;
*pImg1->ph_met = TEMPPT;

// After checking / flipping, set the new headLast and
tailLast values;
*pImg1->ph_met_old = *pImg1->ph_met;
*pImg1->ph_ter_old = *pImg1->ph_ter;

}

/*
 * Find the fluorescence image centroid, multiple uses.
 */

void fluo_moments(struct Images* pImg1){

    // Get image centroid
    float sum_moment_r = 0;
    float sum_moment_c = 0;
    float sum_mass_r   = 0;
    float sum_mass_c   = 0;
    int x, y;
    uchar temp;

    // Get the pixels from the input IplImage add up
    their masses (intensities)
    for (y=0; y<pImg1->FluoImgThresh->height; y++){
        for (x=0; x<pImg1->FluoImgThresh->width;
            x++){
            temp = CV_IMAGE_ELEM(pImg1-
                >FluoImgThresh,uchar,y,x);
            sum_moment_r = sum_moment_r + temp
            * (float) y;
            sum_moment_c = sum_moment_c + temp
            * (float) x;
            sum_mass_r   = sum_mass_r   + temp
            ;
            sum_mass_c   = sum_mass_c   + temp
            ;
        }
    }

    pImg1->Centroid->y = sum_moment_r / sum_mass_r;
    pImg1->Centroid->x = sum_moment_c / sum_mass_c;
    185
// Draw the worm centroid
CvPoint ImCtr = cvPoint(pImg1->Centroid->x, pImg1->Centroid->y);
PImg1->Centroid = ImCtr;
// cvDrawCross(pImg1->FluoImg, &ImCtr, ww, 1, 15);
}

/*
 * Find the GFP maximum (for pharynx tracking, this is the maxima of everything not in the pharynx).
 * This can be used for validation -- GFP intensity as a function of position illuminated
 */

void get_gfp_peak(struct Images* pImg1) {
    // Validation image has been copied in Cam()
    // Get an inverse of the pharynx mask
    cvCopy(pImg1->FluoImgThresh, pImg1->FluoImgThreshInv);
    imcomplement(pImg1->FluoImgThreshInv, pImg1->FluoImgGrayTemp);
    //
}

/*
 * Crop the centerline at a fixed distance, in pixels, instead of a relative position.
 * Useful when worm body may not be fully onscreen
 */

int focus_pt_idx_by_distance(struct Images* pImg1, int dmax) {
    // Declare the return variable's value if no match is found
    int irtr = 0;
    double this_dist = 0;
    double total_dist = 0;

    // Cycle through all points in the Cline, keeping track of the total distance travelled
    for (int i = 1; i < pImg1->Centerline->total; i++) {
CvPoint* last = (CvPoint*) cvGetSeqElem(pImg1->Centerline,i-1);
CvPoint* curr = (CvPoint*) cvGetSeqElem(pImg1->Centerline,i);

this_dist = dist(*last,*curr);
total_dist=total_dist+this_dist;

if(total_dist>dmax || i == pImg1->Centerline->total){
    irtr = i;
    break;
}
}

// Return the distance-selected index
return irtr;

void track_puncta(struct Images* pImg1, CvFont font1){

    // Calculate puncta position in the ROI
    int radius = 15;
    double max_val ,min_val;
    if(pImg1->MaxFlag==1){
       // If we are tracking a good puncta, set
        the search ROI to nearby areas.
        if(pImg1->LostTrackFlag==0){
            cvSetImageROI(FLUO_IMAGE_GRY,cvRect(pImg1->max_loc->x-radius/2,pImg1->max_loc->y-radius/2,radius,radius));
        }
        else{
            cvSetImageROI(FLUO_IMAGE_GRY,cvRect(0,0,pImg1->FluoImg->width,pImg1->FluoImg->height/2));
        }

    }
// Find the nearby maxima (mode 1) or minima (mode 11)
if(pImg1->tracktypeflag==1){
    cvMinMaxLoc(FLUO_IMAGE_GRY,&min_val,&max_val,NULL,pImg1->max_loc);
}
else{
    cvMinMaxLoc(FLUO_IMAGE_GRY,&min_val,&max_val,pImg1->max_loc,NULL);
}

// Require that the tracked puncta is very bright (4x min+1). If not, it is probably a fake puncta or noise, meaning we have lost track of the puncta. In this case, stop moving the stage and then search the entire.

// If bright puncta
if(true){ //max_val < 2*(min_val+1)
    pImg1->max_loc->x += FLUO_IMAGE_GRY->roi->xOffset;
    pImg1->max_loc->y += FLUO_IMAGE_GRY->roi->yOffset;
    pImg1->LostTrackFlag = 0;
}
else{ // If not bright puncta
    // Stop tracking the stage
    pImg1->max_loc->x = pImg1->fluo_img_ctr->x;
    pImg1->max_loc->y = pImg1->fluo_img_ctr->y;
    // Throw a flag to indicate that we've lost track and need a large search area
    pImg1->LostTrackFlag = 1;
    cvPutText(FLUO_IMAGE_CLR,"Failed to track puncta",cvPoint(10,10),&font1,yy);
}

/*
 * Advanced puncta tracking: Pick one tip (head or terminus or the other of the pharynx as the target.
*/
void track_pharynx(struct Images* pImg1, CvFont font1){

    // Determine head and terminus, figure out which one is the terminus.
    // Clear the sampled sequence
    cvClearSeq(pImg1->ContoursSamp);
    // Downsampling pharyngeal contours to save time
    int Npts = std::min(28, pImg1->Contours->total);
    int step = pImg1->Contours->total / Npts;
    step = std::max(step, 1); // minimum step size is 1.

    for(int i=0; i<pImg1->Contours->total; i+=step){
        CvPoint * bndpt = (CvPoint *) cvGetSeqElem(pImg1->Contours,i);
        cvSeqPush(pImg1->ContoursSamp, bndpt);
    }

    // Find which two points are separated by the longest pathlength
    int Dmax = 0;
    int Dtemp = 0;
    int imax = 0;
    int jmax = 0;
    int ct = 0;
    for(int i = 0; i<Npts; i++){
        for(int j=i+1; j<Npts; j++){
            ct += 1;
            CvPoint * bndpti = (CvPoint *) cvGetSeqElem(pImg1->ContoursSamp,i);
            CvPoint * bndptj = (CvPoint *) cvGetSeqElem(pImg1->ContoursSamp,j);

            Dtemp = dist32f(cvPoint2D32f(bndpti->x, bndpti->y), cvPoint2D32f(bndptj->x, bndptj->y));

            if(Dtemp > Dmax){
                Dmax = Dtemp;
                imax = i;
                jmax = j;
            }
        }
    }
}
Retrieve the two points from the sequence
CvPoint* ep1 = (CvPoint *) cvGetSeqElem(pImg1->ContoursSamp,imax);
CvPoint* ep2 = (CvPoint *) cvGetSeqElem(pImg1->ContoursSamp,jmax);

Figure out which of the two endpoints is the tail/terminus of the pharynx by comparing to where
The terminus was in the last frame.
double d_ep1_phterold = dist32f(cvPoint2D32f(ep1->x, ep1->y), cvPoint2D32f(pImg1->ph_ter_old->x, pImg1->ph_ter_old->y));
double d_ep2_phterold = dist32f(cvPoint2D32f(ep2->x, ep2->y), cvPoint2D32f(pImg1->ph_ter_old->x, pImg1->ph_ter_old->y));

if (d_ep1_phterold < d_ep2_phterold){
    *pImg1->ph_ter = *ep1;
    *pImg1->ph_met = *ep2;
} else{
    *pImg1->ph_ter = *ep2;
    *pImg1->ph_met = *ep1;
}

Store the location of the terminus for use in finding the terminus in the next frame
*pImg1->ph_ter_old = *pImg1->ph_ter;

Take whichever bulb the user has requested
if(pImg1->BulbSelectFlag==1){
    *pImg1->max_loc = *pImg1->ph_ter;
} else{
    *pImg1->max_loc = *pImg1->ph_met;
}

BULB TRACKING ONLY (#21):
Find the maxima nearest selected bulb and assign that to be the tracked puncta.
For edge tracking (#22), just use the max_loc found on the pharynx edge.
if(pImg1->tracktypeflag==21){
    int radius = 8;
if(pImg1->max_loc->x > radius){
    cvSetImageROI(FLUO_IMAGE_GRY,cvRect(pImg1->max_loc-
    >x-radius/2,pImg1->max_loc->y-radius/2,ADIUS,ADIUS));
    cvMinMaxLoc(FLUO_IMAGE_GRY,&pImg1->max_val,NULL,pImg1-
    ->max_loc);
    pImg1->max_loc->x += pImg1->FluoImgGray8bit-
    >roi->xOffset; // Put x,y back in global coordinates, not ROI-localized
    pImg1->max_loc->y += pImg1->FluoImgGray8bit-
    >roi->yOffset;
    cvResetImageROI(FLUO_IMAGE_GRY);
}

// WING TRACKING ONLY(#23 && #26)
// illuminate a vector perpendicular to the head direction
// Only do this if a validly large pharynx was found
if((pImg1->tracktypeflag==23 || pImg1->tracktypeflag==26) &&
(dist(*ep1,*ep2) > 4)){

    // get the pharynx orientation vector
    CvPoint V = cvPoint(pImg1->ph_met->x-pImg1->ph_ter-
    ->x,pImg1->ph_met->y-pImg1->ph_ter->y);

    // get the perpendicular vector, making it 2*20% as long as
    // the pharynx.
    double f = ((double) pImg1->wingPct )/100.0;
    CvPoint Vp = cvPoint(-(V.y)*f,V.x*f);

    // Make the perpendicular vector centered at the tracking
    // point
    *pImg1->max_wing1 = cvPoint(pImg1->max_loc->x + Vp.x ,
    pImg1->max_loc->y + Vp.y);
    *pImg1->max_wing2 = cvPoint(pImg1->max_loc->x - Vp.x ,
    pImg1->max_loc->y - Vp.y);
}
else if(pImg1->tracktypeflag==23 ){
    *pImg1->max_wing1 = cvPoint(pImg1->max_loc->x + 1 , pImg1-
    ->max_loc->y + 1);
    *pImg1->max_wing2 = cvPoint(pImg1->max_loc->x - 1 , pImg1-
    ->max_loc->y - 1);
    cvPutText(FLUO_IMAGE_CLR,"TOO SMALL",cvPoint(10,12),&font1,yy);
    return;
}

// PHARYNX RELATIVE LOCATION FROM TERMINAL BULB TRACKING OR
// ARBITRARY ENDPOINT ONLY (#24 && #27 && #28 && #51)
/illuminate a segment of the pharynx with a wing.
if ( is_tracktype(pImg1->tracktypeflag,5,24,27,28,51,52) &&
dist(*ep1,*ep2) > 4){
    int Nsegs = pImg1->NumSteps;
    int thispt= pImg1->FocusPtIdx;
    int xn = 0;
    int yn = 0;

    // Calculate the region of the pharynx to illuminate
    if(pImg1->BulbSelectFlag==1){
        xn=linspace(pImg1->ph_ter->x,pImg1->ph_met->x,Nsegs,thispt);
        yn=linspace(pImg1->ph_ter->y,pImg1->ph_met->y,Nsegs,thispt);
    }
    else{
        xn=linspace(pImg1->ph_met->x,pImg1->ph_ter->x,Nsegs,thispt);
        yn=linspace(pImg1->ph_met->y,pImg1->ph_ter->y,Nsegs,thispt);
    }
    *pImg1->max_loc = cvPoint(xn,yn);

    // If using two arbitrary endpoints, force the start point
    index to be inside [1 FocusPtIdx]
    if(pImg1->StartPtIdx<1)   {pImg1->StartPtIdx =1; }
    if(pImg1->StartPtIdx>pImg1->FocusPtIdx) {pImg1->StartPtIdx = pImg1->FocusPtIdx; }

    // If using two arbitrary endpoints, calculate the second
    one
    thispt = pImg1->StartPtIdx;
    if(pImg1->BulbSelectFlag==1){
        xn=linspace(pImg1->ph_ter->x,pImg1->ph_met->x,Nsegs,thispt);
        yn=linspace(pImg1->ph_ter->y,pImg1->ph_met->y,Nsegs,thispt);
    }
    else{
        xn=linspace(pImg1->ph_met->x,pImg1->ph_ter->x,Nsegs,thispt);
        yn=linspace(pImg1->ph_met->y,pImg1->ph_ter->y,Nsegs,thispt);
    }
    *pImg1->StartPt = cvPoint(xn,yn);
// Generate perpendicular wings for this subpoint

// get the pharynx orientation vector
CvPoint V = cvPoint(pImg1->ph_met->x-pImg1->ph_ter->x,pImg1->ph_met->y-pImg1->ph_ter->y);

// get the perpendicular vector, making it f*100% as long as the pharynx.
double f = ((double) pImg1->wingPct)/100.0;
CvPoint Vp = cvPoint(-V.y*f,V.x*f);

// Make the perpendicular vector centered at the tracking point
*pImg1->max_wing1 = cvPoint(pImg1->max_loc->x + Vp.x , pImg1->max_loc->y + Vp.y);
*pImg1->max_wing2 = cvPoint(pImg1->max_loc->x - Vp.x , pImg1->max_loc->y - Vp.y);

// Make the second perpendicular vector centered at the terminal bulb, unless we are doing two arbitrary endpoints; then make it centered at the other endpoint
if(is_tracktype(pImg1->tracktypeflag,2,28,52)){
    *pImg1->max2_wing1 = cvPoint(pImg1->StartPt->x + Vp.x , pImg1->StartPt->y + Vp.y);
    *pImg1->max2_wing2 = cvPoint(pImg1->StartPt->x - Vp.x , pImg1->StartPt->y - Vp.y);
}
else{
    *pImg1->max2_wing1 = cvPoint(pImg1->ph_ter->x + Vp.x , pImg1->ph_ter->y + Vp.y);
    *pImg1->max2_wing2 = cvPoint(pImg1->ph_ter->x - Vp.x , pImg1->ph_ter->y - Vp.y);
}

else if(pImg1->tracktypeflag==24 || pImg1->tracktypeflag==27){
    *pImg1->max_wing1 = cvPoint(pImg1->max_loc->x + 1, pImg1->max_loc->y + 1);
    *pImg1->max_wing2 = cvPoint(pImg1->max_loc->x - 1, pImg1->max_loc->y - 1);
    *pImg1->max2_wing1= cvPoint(pImg1->ph_ter->x + 1 , pImg1->ph_ter->y + 1 );
    *pImg1->max2_wing2= cvPoint(pImg1->ph_ter->x - 1 , pImg1->ph_ter->y - 1 );
    cvPutText(FLUO_IMAGE_CLR,"TOO SMALL",cvPoint(10,12),&font1,yy);
return;
}

// PHARYNX Dual-wing tracking (hits ADE and CEP in pDat-1::ChR2

if(pImg1->tracktypeflag==25 && (dist(*ep1,*ep2) > 4)){

    /* Get the first wing */

    // get the pharynx orientation vector
    CvPoint V = cvPoint(pImg1->ph_met->x-pImg1->ph_ter->x,pImg1->ph_met->y-pImg1->ph_ter->y);

    // get the perpendicular vector, making it 2*20% as long as the pharynx.
    double f = (((double) pImg1->wingPct)/100.0;
    CvPoint Vp = cvPoint(-V.y)*f,V.x*f);

    // Make the perpendicular vector centered at the tracking point
    *pImg1->max_wing1 = cvPoint(pImg1->ph_ter->x + Vp.x
    , pImg1->ph_ter->y + Vp.y);
    *pImg1->max_wing2 = cvPoint(pImg1->ph_ter->x - Vp.x
    , pImg1->ph_ter->y - Vp.y);

    /* Get the second wing */

    *pImg1->max2_wing1= cvPoint(pImg1->ph_met->x + Vp.x
    , pImg1->ph_met->y + Vp.y);
    *pImg1->max2_wing2= cvPoint(pImg1->ph_met->x - Vp.x
    , pImg1->ph_met->y - Vp.y);

}
else if(pImg1->tracktypeflag==25 ){

    *pImg1->max_wing1 = cvPoint(pImg1->max_loc->x + 1 , pImg1->max_loc->y + 1);
    *pImg1->max_wing2 = cvPoint(pImg1->max_loc->x - 1 , pImg1->max_loc->y - 1);
    cvPutText(FLUO_IMAGE_CLR,"TOO SMALL",cvPoint(10,12),&font1,yy);
    return;
}

    cvPutText(FLUO_IMAGE_CLR,"Switch Phar Bulb = lft clk",cvPoint(10,12),&font1,yy);

    // Add pharynx contours to Centerline for printing to disk
pImg1->Centerline=cvCloneSeq(pImg1->Contours);
}

/*
 * Dummy puncta tracking: set the tracker to be a
 * predetermined point series
 */

void track_dummy(struct Images* pImg1, int ct){
    int ID = floor( (double) ct/50.0);        // Show each spot for 50 frames
    *pImg1->max_loc = cvPoint(55+4*ID,35+4*ID);
}

/*
 * Complete worm segmentation, up to and including head/tail
 * identification.
 * also draws the contour of the largest object on pImg1->Mainimggray and color.
 * subtronly defines whether to return after subtraction and skip
 * segmentation
 */
void track_whole_worm(struct Images* pImg1, int subtronly){
    /*
    * SECTION 1: Background subtraction
    */
               // Implemented in Cam()

    /*
    * SECTION 2: Smoothing, thresholding and contours
    */
               // Clean up sequences
    cvClearSeq(pImg1->s);
    cvClearSeq(pImg1->phi);
    cvClearSeq(pImg1->dphi_ds);
    cvClearSeq(pImg1->Centerline);
    cvClearSeq(pImg1->CenterlineTemp);
    cvClearSeq(pImg1->LeftBound);
    cvClearSeq(pImg1->RightBound);
// Default the Centerline to zeros
CvPoint* this_point = (CvPoint*)
malloc(sizeof(CvPoint));
    *this_point = cvPoint(0,0);
for (int i=0; i<pImg1->NumSteps;i++){
    cvSeqPush(pImg1->Centerline,this_point);
}

// Find the contours
if(true){
    //cvSmooth(pImg1->FluoImgGray8bit,pImg1->SEimg,CV_GAUSSIAN,pImg1->SE);
    // Smooth the image
    cvThreshold(pImg1->FluoImgGray8bit,pImg1->FluoImgThresh,pImg1->fluoThresh,255,THRESH_MODE);
    // Threshold the image
    cvRectangle(pImg1->FluoImgThresh,cvPoint(1,1),cvPoint(3,pImg1->FluoImgThresh->height),kk,-1);
    isolate_largest_object(pImg1->FluoImgThresh,pImg1->FluoImgGray8bitTemp,pImg1->tempMem);
    // Delete small binary objects
    cvFindContours(pImg1->FluoImgGray8bitTemp,
pImg1->contourMemTemp,&pImg1->ContoursTemp,sizeof(CvContour),
    CV_RETR_EXTERNAL,CV_CHAIN_APPROX_NONE,
cvPoint(0,0));

    // If the contours fail to identify they have a memory address of 0x0
    if(pImg1->ContoursTemp > 0){
        if(pImg1->ContoursTemp->total > 25){
            smoothPtSequence(pImg1->ContoursTemp,pImg1->Contours,pImg1->ker);
            cvDrawPlot(FLUO_IMAGE_GRY,pImg1->Contours,ww,1); // Draws contours on the actual image
        }
    }
}

/*
  * SECTION 3: image moment
*/

// DELETED. See void fluo_moments for similar code.
/* SECTION 4: Head and tail */

if (pImg1->Contours->total > 25) {

    // Set default head and tail indices to prevent crashes if none is found
    pImg1->headindex = 1;
    pImg1->tailindex = 1;
    pImg1->mostcurvy = 0;
    pImg1->secondmostcurvy = 0;
    pImg1->mostcurvyindex = 1;
    pImg1->secondmostcurvyindex = 0;

    /** step 1: go through and get the curvature of each point by taking the dot products of the incoming and exiting vectors to that point see my matlab demo "c2_newheadtailpractice.mm" **/

    // set the fixed reference vector
    pImg1->fvecn = cvPoint2D32f(1,0);

    for (int i = 0; i < pImg1->Contours->total; i++) {

        // calculate the angle of the tangent vector with respect to a fixed one by taking the dot product between the fixed vector (x,y) = (1,0) and the current 'sliding' vector
        pImg1->aheadpt = (CvPoint*) cvGetSeqElem(pImg1->Contours, i+pImg1->curvestep);
        pImg1->behindpt = (CvPoint*) cvGetSeqElem(pImg1->Contours, i-pImg1->curvestep);

        pImg1->tvecn = cvPoint2D32f( (pImg1->aheadpt->x) - (pImg1->behindpt->x),
                                   (pImg1->aheadpt->y) - (pImg1->behindpt->y));

        pImg1->thisphi = fPointDotTheta(&pImg1->fvecn, &pImg1->tvecn);

        cvSeqPush(pImg1->phi, pImg1->thisphi);
    }
}
// calculate the arc length along the segment we just used.

pImg1->d = dist32f(cvPoint2D32f( (pImg1->aheadpt->x) , (pImg1->aheadpt->y)),
                   cvPoint2D32f( (pImg1->behindpt->x) , (pImg1->behindpt->y)));

cvSeqPush(pImg1->s,&pImg1->d);
}

/** step 2: find the derivative of the curvature dphi/ds **/

for (int i = 0; i < pImg1->Contours->total; i++) {
    pImg1->phi1 = (float*) cvGetSeqElem(pImg1->phi,i);
    pImg1->phi2 = (float*) cvGetSeqElem(pImg1->phi,i+1);
    pImg1->d1   = (float*) cvGetSeqElem(pImg1->s,i);
    pImg1->d2   = (float*) cvGetSeqElem(pImg1->s,i+1);
    //pImg1->thisdpds = abs( ((*pImg1->phi2) - (*pImg1->phi1)) / ((*pImg1->d1) - (*pImg1->d2)));
    // This is wrong but for some reason it works better
    pImg1->thisdpds = abs((*pImg1->phi2) / (*pImg1->d1) - (*pImg1->phi1) / (*pImg1->d2));
    cvSeqPush(pImg1->dphi_ds,&pImg1->thisdpds);
}

/// find the tail (most curvy point)
for (int i = 0; i < pImg1->Contours->total; i++) {
    cvGetSeqElem(pImg1->dphi_ds,i);
    if (*pImg1->dotprodptr >pImg1->mostcurvy) {
        //if this location is curvier than the previous mostcurvy location
        pImg1->mostcurvy = *pImg1->dotprodptr;
        //replace the mostcurvy point
        pImg1->mostcurvyindex = i;
    }
}

*pImg1->tail = *(CvPoint*)
   cvGetSeqElem(pImg1->Contours, pImg1->mostcurvyindex);
   pImg1->tailindex = pImg1->mostcurvyindex;
///exclude the tail and points near the tail before estimating
/// the head (next curviest point)

for (int i = 0; i < pImg1->Contours->total; i++) {
    pImg1->dotprodptr = (float*)cvGetSeqElem(pImg1->dphi_ds, i);
    pImg1->neartail = (float)DistBetPtsOnCircBound(pImg1->Contours->total, i, pImg1->tailindex);
    if (*pImg1->dotprodptr > pImg1->secondmostcurvy && pImg1->neartail > pImg1->Contours->total/4) {
        pImg1->secondmostcurvy = *pImg1->dotprodptr;
        pImg1->secondmostcurvyindex = i;
    }
}

*pImg1->head = *(CvPoint*)cvGetSeqElem(pImg1->Contours, pImg1->secondmostcurvyindex);
pImg1->headindex = pImg1->secondmostcurvyindex;
checkHeadTail(pImg1);

}

else{return;}

/*
 * SECTION 5: Worm centerline
 */

// Split up the sides. Note that "Left" and "Right" are arbitrary names

if (pImg1->headindex==pImg1->tailindex) printf("Error! pImg1->HeadIndex==pImg1->TailIndex in SegmentWorm()!\n");
pImg1->LeftBound=cvSeqSlice(pImg1->Contours, cvSlice(pImg1->headindex,pImg1->tailindex),pImg1->leftBoundMem ,1);
pImg1->RightBound=cvSeqSlice(pImg1->Contours, cvSlice(pImg1->tailindex,pImg1->headindex),pImg1->rightBoundMem, 1);
cvSeqInvert(pImg1->RightBound);

if(pImg1->LeftBound->total < 3){printf("BAD SEGMENTATION - ABORTING\n"); return;};
// Seed Cline with the head value
*pImg1->midP = cvPoint2D32f(pImg1->head->x,pImg1->head->y);
*pImg1->midPi = cvPoint(pImg1->midP->x,pImg1->midP->y);
cvSeqPush(pImg1->CenterlineTemp,pImg1->midPi);

// Going along the worm, add each coordinate of the centerline by finding the midpoint between the sides in that region.

for(int i=2; i<=pImg1->NumSteps-1; i++){
    double pImg1->idxA = (double)i/(double)pImg1->NumSteps * (double)pImg1->LeftBound->total;
    pImg1->idxB = (double)i/(double)pImg1->NumSteps * (double)pImg1->RightBound->total;
    round(&pImg1->idxA);
    round(&pImg1->idxB);
    pImg1->Ai = (CvPoint*) cvGetSeqElem(pImg1->LeftBound,pImg1->idxA);
    *pImg1->thisA = cvPoint2D32f((double)pImg1->Ai->x,(double)pImg1->Ai->y);

    pImg1->Bi = (CvPoint*) cvGetSeqElem(pImg1->RightBound,pImg1->idxB);
    *pImg1->thisB = cvPoint2D32f((double)pImg1->Bi->x,(double)pImg1->Bi->y);
    midpoint(pImg1->thisA,pImg1->thisB,pImg1->midP);
    *pImg1->midPi = cvPoint(pImg1->midP->x,pImg1->midP->y);
    cvSeqPush(pImg1->CenterlineTemp,pImg1->midPi);
}

*pImg1->midP = cvPoint2D32f(pImg1->tail->x,pImg1->tail->y);
*pImg1->midPi = cvPoint(pImg1->midP->x,pImg1->midP->y);
cvSeqPush(pImg1->CenterlineTemp,pImg1->midPi);

// Smooth the centerline with the same kernel as the contour
*pImg1->CenterlineTemp = smoothPtSequence(pImg1->Centerline,pInp1->ker,cvCreateMemStorage(0));

// Resample the centerline to force equivalent arclengths between all points.
resampleSeqConstPtsPerArcLength(Cline, pImg1->Centerline, NumSteps);
*pImg1->Centerline = cvCloneSeq(pImg1->CenterlineTemp,pImg1->centerlineMem);
// Get the focus point (along the centerline)
if (pImg1->tracktypeflag!=32){
    pImg1->FocusPt = (CvPoint*) cvGetSeqElem(pImg1->Centerline,pImg1->FocusPtIdx);
    *pImg1->max_loc = *pImg1->FocusPt;
}
else{
    pImg1->FocusPtIdx = focus_pt_idx_by_distance(pImg1,pImg1->FocusPtDist);
    pImg1->FocusPt = (CvPoint*) cvGetSeqElem(pImg1->Centerline,pImg1->FocusPtIdx);
    *pImg1->max_loc = *pImg1->FocusPt;
    pImg1->StartPtIdx = focus_pt_idx_by_distance(pImg1,pImg1->StartPtDist);
    pImg1->StartPt = (CvPoint*) cvGetSeqElem(pImg1->Centerline,pImg1->StartPtIdx);
}

pImg1->StartPt = (CvPoint*) cvGetSeqElem(pImg1->Centerline,pImg1->StartPtIdx);
    // Assign the Head point and the focus point to be the pharynx bulb edges
    // Verify that the area of illumination is well within the bounds of the image.
    if (pImg1->FocusPtIdx > 0.99 * pImg1->Centerline->total && is_tracktype(pImg1->tracktypeflag,1,32)){
        pImg1->FocusPtIdx = 0.99* pImg1->Centerline->total;
        pImg1->FocusPt = (CvPoint*) cvGetSeqElem(pImg1->Centerline,pImg1->FocusPtIdx);
        pImg1->StartPt = (CvPoint*) cvGetSeqElem(pImg1->Centerline,0);
        printf("BAD SEGMENTATION - OUT OF BOUNDS ILLUM REGION\n");
        return;
    }

    if(pImg1->FocusPtIdx==0){pImg1->FocusPtIdx=1; pImg1->FocusPt = (CvPoint*) cvGetSeqElem(pImg1->Centerline,pImg1->FocusPtIdx );}
}
Advanced puncta tracking: Forecast the puncta based on worm's previous motion.

Applies to any medium of puncta tracking.

```c
void forecast_puncta(struct Images* pImg1){

    // Setup variables
    double *mx      = (double*) malloc(sizeof(double));
    double *my      = (double*) malloc(sizeof(double));
    CvPoint* tmp   = (CvPoint*) malloc(sizeof(CvPoint));
    *mx            = 0;
    *my            = 0;

    // estimate the velocity, in pixels/millisec, of the puncta. Make sure fps, mx and my are nonzero.
    if(pImg1->fps<15){pImg1->fps = 30;} else{
        *mx = (pImg1->max_loc->x - pImg1->max_loc_old->x) /
             (1000/pImg1->fps);  // pixels per millisecond
        *my = (pImg1->max_loc->y - pImg1->max_loc_old->y) /
             (1000/pImg1->fps);
    }

    // linearly extrapolate the future position of the puncta
    *tmp = cvPoint(pImg1->max_loc->x + (int)(*mx) * (pImg1->dt_pred)), pImg1->max_loc->y + (int)(*my) * (pImg1->dt));

    // Keep track of the max_loc (original value) for use in the next frame's velocity calculation
    *pImg1->max_loc_old = *pImg1->max_loc;

    // Draw the original cross on the fluo image to view the effect of forecasting.
    cvDrawCross(pImg1->FluoImg,pImg1->max_loc,ww,1,8);

    // Finally, reassign max_loc to be the forecast point
    *pImg1->max_loc = *tmp;

```
// force the max_loc to be in-bounds to prevent crashes
pImg1->max_loc->x = std::max(pImg1->max_loc->x,1);  
pImg1->max_loc->x = std::min(pImg1->max_loc->x,pImg1->FluoImg->width-1);  
pImg1->max_loc->y = std::max(pImg1->max_loc->y,1);  
pImg1->max_loc->y = std::min(pImg1->max_loc->y,pImg1->FluoImg->height-1);
}

AnthonysCamFuncs.h
#ifndef ANTHONYSCAMFUNCS_H_
#define ANTHONYSCAMFUNCS_H_

// Accessory functions
#include "master.h"
#include "pvcam.h"

class SimpleSerial; /* forward declaration of SimpleSerial */
void getFrameRate(double *ct, double *diff, double *tcum, double *rate);
void cvDrawCross(IplImage* img, CvPoint* ctr, CvScalar color = bb, int thickness = 1, int radius = 5);
void cvRectangle4pt(IplImage* img, CvPoint* pt1, CvPoint* pt2, CvPoint* pt3, CvPoint* pt4, CvScalar clr, int thickness=-1);
void cvDrawCross2D32f(IplImage* img, CvPoint2D32f* ctr, CvScalar color = bb, int thickness = 1, int radius = 5);
void cvDrawPlot(IplImage* img, CvSeq* seq, CvScalar color, int thickness = 1);
void cvDrawPlotFlat(IplImage* img, CvSeq* seq, int r, int capflag, int sideflag);
void cvDrawPlot2D32f(IplImage* img, CvSeq* seq, CvScalar color, int thickness = 1);
void cvRotateImage(IplImage* src, IplImage* dst, double angle);
void saveFrame(IplImage* img_dark, IplImage* img_fluo, int ct2, int FluoFlag = 0);
void imcomplement(IplImage* img, IplImage* tempimg);
void set_status_image(struct Images* pImg1, const char* cRate);
void CallBackFunc(int event, int x, int y, int flags, void* userdata);

// Check if a file exists. Used before opening a new file for the
written data
int exists(const char *fname);

// Find available USB cameras
int find_system_cameras(int hmin);
// Main camera/tracking function
void Cam(struct Images *pImg1, SimpleSerial *mySerial, struct DlpCalib *pDlp1, struct Script *pScript);

// Multithreading function
int GetPvcamFrame(int hCam, uns16* frame, rgn_type region, uns32 size , int PVCAM_ACQUIRE_MODE, int *readyflag);

// Secondary camera functions for setting the segmentation
void CamBG(struct Images *pImg1);

#endif

AnthonysCamFuncs.cpp
/*
 * AnthonysCamFuncs.cpp
 * Anthony Fouad
 * Fang-Yen Group, 6/2014
 * Functions used read from the cameras, set initial
 * segmentations, and related accessories
 */

//Standard functions
#include "stdafx.h"
#include "stdlib.h"
#include "stdio.h"
#include <iostream>
#include <cmath>

// OpenCV 2.4 functions
#include "opencv\cv.h"
#include "opencv\highgui.h"

// Boost functions
#include "boost\asio.hpp"
#include "SimpleSerial.h"
#include "boost\thread.hpp"

//PVcam functions
#include "master.h"
#include "pvcam.h"
#include "PVcamAccessoryFuncs.h"
// Anthony's functions
#include "Images.h"
#include "AnthonysColors.h"
#include "AnthonysTimer.h"
#include "AnthonysCalculations.h"
#include "StageAccessoryFuncs.h"
#include "DlpAccessoryFuncs.h"

using namespace std;

/*
 *  get the framerate (requires Anthony's tic-toc definitions).
 *  Calculate the rate every 1 s
 */
void getFrameRate(double *ct, double *diff, double *tcum, double *rate){
  // ct:      is the current frame number (up to infinity, more or less)
  // tend:    the time in seconds that elapsed between #tic and #toc.
  // tcum:    the total time elapsed in this group of 10 frames
  *tcum = *tcum + *diff;
  if (*tcum > 0.5){
    *rate = *ct / *tcum;
    *tcum = 0;
    *ct = 0;
  }
  *diff = 0;
}

/*
 * Draws a cross at the coordinate ctr.
 */
void cvDrawCross(IplImage* img, CvPoint* ctr, CvScalar color, int thickness, int radius){
    // Draw the vertical line in the cross
    CvPoint pt1 = cvPoint(ctr->x, ctr->y + radius);
    CvPoint pt2 = cvPoint(ctr->x, ctr->y - radius);
    cvLine(img, pt1, pt2, color, thickness, 8, 0);

    // Draw the horizontal line in the cross
    pt1 = cvPoint(ctr->x + radius, ctr->y);
    pt2 = cvPoint(ctr->x - radius, ctr->y);
    cvLine(img, pt1, pt2, color, thickness, 8, 0);
}

/*
 * Draws a cross at the coordinate ctr with the point specified as a
 * CvPoint2D32f.
 */
void cvDrawCross(IplImage* img, CvPoint2D32f* ctr, CvScalar color = bb, int thickness = 1, int radius = 5){
    CvPoint ictr = cvPoint(ctr->x, ctr->y);
    // ...
cvDrawCross(img,&ictr,color,thickness,radius);

/*
* Draw a four point (i.e. rotated) filled rectangle (line==-1)
*/

void cvRectangle4pt(IplImage* img, CvPoint* pt1, CvPoint* pt2, CvPoint* pt3, CvPoint* pt4, CvScalar clr, int thickness){

    // Filled mode: fill in the area
    if(thickness<0){
        // Set up the list of boundary points. Double fill to avoid the hourglass
        CvPoint pts[1][8];
        pts[0][0] = *pt1;
        pts[0][1] = *pt2;
        pts[0][2] = *pt4;
        pts[0][3] = *pt3;

        CvPoint* ppt[1] = {pts[0]};
        int npt[] = {4};

        // Fill a polygon
        cvFillPoly(img, ppt , npt,1,clr);
    }

    // Line mode: just draw lines connecting the points
    else{
        cvLine(img,*pt1,*pt2,clr,thickness);
        cvLine(img,*pt2,*pt4,clr,thickness);
        cvLine(img,*pt4,*pt3,clr,thickness);
        cvLine(img,*pt3,*pt1,clr,thickness);
    }
}

/*
* Draws a matlab style plot of a sequence of CvPoint* elements
*/

void cvDrawPlot(IplImage* img, CvSeq* seq, CvScalar color, int thickness){
// Draw the vertical line in the cross
CvPoint *A, *B;
for(int i=0; i<seq->total-1; i++){
    A = (CvPoint*) cvGetSeqElem(seq,i);
    B = (CvPoint*) cvGetSeqElem(seq,i+1);
    cvLine(img,*A,*B,color,thickness,8,0);
}

/*
 * Same as cvDrawPlot, EXCEPT that line endings are flat instead of round.
 * r is the half-thickness of the line.
 * draws filled rectangles between each point in the sequence
 * sideflag has values 0, 1 or 2.
 * 0 means both sides.
 * 1 means side 1.
 * 2 means side 2.
 */

void cvDrawPlotFlat(IplImage* img, CvSeq* seq, int r, int capflag, int sideflag){

    // Draw a circle at the first point to round that edge ONLY (nose tip)
    CvPoint* Pi = (CvPoint*) cvGetSeqElem(seq,0);
    if(capflag){
        //cvCircle(img,*Pi,r,ww,-1);
    }

    // If a long sequence, draw the usual chain of rectangles
    if(seq->total>8){
        // Draw forwards filled rectangles
        for(int i = 8; i<seq->total;i+=1){

            // Get the current point and last point
            CvPoint* Pi = (CvPoint*) cvGetSeqElem(seq,i);
            CvPoint* Pm = (CvPoint*) cvGetSeqElem(seq,i-8);

            // Get the normal vector to the vector between the points
            CvPoint2D32f V = cvPoint2D32f(Pi->x-Pm->x, Pi->y-Pm->y);

            // Normalize the vector to length 1
            double normV = sqrt(pow(V.x,2)+pow(V.y,2));
            V = cvPoint2D32f(V.x/normV,V.y/normV);

        }
    }
// Get the perpendicular vector, making it length r
CvPoint2D32f Vp = cvPoint2D32f(-(V.y)*r,(V.x)*r);

// Get the four perpendicular wings to the current point and its predecessor

// Preallocate the points
CvPoint wing1a = cvPoint(0,0);
CvPoint wing1b = cvPoint(0,0);
CvPoint wing2a = cvPoint(0,0);
CvPoint wing2b = cvPoint(0,0);

// If side1 or side2 flags are triggered, draw over only half of the worm image.
if(sideflag==1){
    wing1a = cvPoint(Pi->x+Vp.x,Pi->y+Vp.y);
    wing1b = cvPoint(Pi->x,Pi->y);
    wing2a = cvPoint(Pm->x+Vp.x,Pm->y+Vp.y);
    wing2b = cvPoint(Pm->x,Pm->y);
}
else if(sideflag==2){
    wing1a = cvPoint(Pi->x,Pi->y);
    wing1b = cvPoint(Pi->x-Vp.x,Pi->y-Vp.y);
    wing2a = cvPoint(Pm->x,Pm->y);
    wing2b = cvPoint(Pm->x-Vp.x,Pm->y-Vp.y);
}
else{
    wing1a = cvPoint(Pi->x+Vp.x,Pi->y+Vp.y);
    wing1b = cvPoint(Pi->x-Vp.x,Pi->y-Vp.y);
    wing2a = cvPoint(Pm->x+Vp.x,Pm->y+Vp.y);
    wing2b = cvPoint(Pm->x-Vp.x,Pm->y-Vp.y);
}

// Draw the rectangle between these points
vRectangle4pt(img,&wing1a,&wing1b,&wing2a,&wing2b,ww,-1);
}
}

// If a short sequence, draw one line using the thickness of the requested sequence, at the midpoint
else if (seq->total>2){
    // Get the current point and last point
    CvPoint* Pi = (CvPoint*) cvGetSeqElem(seq,0);
    CvPoint* Pm = (CvPoint*) cvGetSeqElem(seq,2);

// Get the normal vector to the vector between the points
CvPoint2D32f V = cvPoint2D32f(Pi->x-Pm->x,Pi->y-Pm->y);

// Normalize the vector to length 1
double normV = sqrt(pow(V.x, 2) + pow(V.y, 2));
if(normV==0){normV=0.25;} V = cvPoint2D32f(V.x/normV,V.y/normV);

// Get the perpendicularly vector, making it length r
CvPoint2D32f Vp = cvPoint2D32f(-(V.y)*r,(V.x)*r);

// Get the four perpendicular wings to the current point and its predecessor
CvPoint wing1a = cvPoint(Pi->x+Vp.x,Pi->y+Vp.y);
CvPoint wing1b = cvPoint(Pi->x-Vp.x,Pi->y-Vp.y);

// Draw the rectangle between these points
cvLine(img,wing1a,wing1b,ww,seq->total);

*
* Draws a matlab style plot of a sequence of CvPoint2D32f* elements
* /
void cvDrawPlot2D32f(IplImage* img, CvSeq* seq, CvScalar color, int thickness){

    // Draw the vertical line in the cross
    CvPoint2D32f *A, *B;
    for(int i=0; i<seq->total-1; i++){
        A = (CvPoint2D32f*) cvGetSeqElem(seq,i);
        B = (CvPoint2D32f*) cvGetSeqElem(seq,i+1);
        cvLine(img,cvPoint(A->x,A->y),cvPoint(B->x,B->y),color,thickness,8,0);
    }
}

/*
* Rotate an image. Adapted from:
http://stackoverflow.com/questions/2289690/opencv-how-to-rotate-iplimage
*/
void cvRotateImage(IplImage *src, IplImage *dst, double angle)
{
    CvPoint2D32f src_center = cvPoint2D32f(src->width/2.0F, src->height/2.0F);
    CvMat* rot_mat = cvCreateMat(2,3,CV_32FC1);
    CvMat* warp_mat = cvCreateMat(2,3,CV_32FC1);
    rot_mat = cv2DRotationMatrix(src_center, angle,1,rot_mat);
    cvWarpAffine(src, dst, rot_mat);
}

/*
 * Save a frame to disk
 */

void saveFrame(IplImage* img_dark, IplImage* img_fluo, int ct2, int FluoFlag)
{
    char fluobase[] = {"fluo_cap_"};
    char darkbase[] = {"dark_cap_"};

    char toprint[100];
    sprintf(toprint, "%s%d.jpg", darkbase, ct2);
    cvSaveImage(toprint, img_dark);

    if(FluoFlag){
        sprintf(toprint, "%s%d.jpg", fluobase, ct2);
        cvSaveImage(toprint, img_fluo);
    }
}

/*
 * invert an image
 */

void imcomplement(IplImage* img, IplImage *tempimg){

    // Set the maximum value for 8 or 16 bit images
    double maxval = 255;
    if(img->depth==16){maxval = 65535;}

    // Enforce equivalent qualities between img and tempimg
    if(img->nChannels != tempimg->nChannels){printf("Unmatched channels supplied to imcomplement. ABORTING...\n");}
    if(img->depth != tempimg->depth)
        {printf("Unmatched depths supplied to imcomplement. ABORTING...\n");}
// Perform the flip
    cvSet( tempimg, cvScalar(maxval,maxval,maxval), NULL);
    cvSub( tempimg,img,img);
}

/*
 * Callback function for mouse clicks on the image (used for fluo tracking)
 */

void CallBackFunc(int event, int x, int y, int flags, void* userdata){
    // Rearrange the input struct to be Images* format
    struct Images* pImg1 = (Images*) userdata;

    // Get the button down and put the coordinates in pImg1
    if ( event == CV_EVENT_LBUTTONDOWN )
    {
        // Store the event in pImg1 so it can be detected elsewhere
        pImg1->clickEvent = event;

        pImg1->max_loc->x = x;
        pImg1->max_loc->y = y;
        pImg1->startAutoFlag = 1; // Start automation once the user left or right clicks
        //std::cout << "Selected position : (" << pImg1->max_loc->x << ", " << pImg1->max_loc->y << ")" << " | Bulb = " << pImg1->BulbSelectFlag << std::endl;

        // Allow the user to flip which pharyngeal puncta is in use, if applicable
        if( pImg1->BulbSelectFlag == 1) {pImg1->BulbSelectFlag=2;}
        else if(pImg1->BulbSelectFlag == 2) {pImg1->BulbSelectFlag=1;}
    }

    // If the user clicks the right mouse button, it's time to start the experiment
    if ( event == CV_EVENT_RBUTTONDOWN )
    {
        pImg1->startAutoFlag = 1; // Start automation once the user left or right clicks
// If no experiment is running, then START the experiment on right click.
if(pImg1->startExptFlagSignal==0){
pImg1->startExptFlagSignal=1; }

// else, if an experiment is running, then STOP the experiment on right click
else if (pImg1->startExptFlagSignal==1){
pImg1->startExptFlagSignal=0;
}

printf("Experiment status: %d\n",pImg1->startExptFlagSignal);
}

// If the user clicks the middle button, reset the image center to that point
if (event == CV_EVENT_MBUTTONDOWN){
pImg1->fluo_img_ctr->x = x;
pImg1->fluo_img_ctr->y = y;
//printf("mid button (x,y) = (%d,%d)\n",x,y);
}

/*
 * Update the system status image
 */

void set_status_image(struct Images* pImg1, const char* cRate){

    // Start with a blank image
    cvCopyImage(pImg1->StatusImgBlank,pImg1->StatusImg);

    // Plot some text on it
    CvFont font1;
cvInitFont(&font1,CV_FONT_HERSHEY_SIMPLEX,.5,.5,0,1,8);

    // Plot the FPS
    cvPutText(pImg1->StatusImg,cRate,cvPoint(290,120),&font1,yy);

    // Plot dark cam status
    cvPutText(pImg1->StatusImg,"ON",cvPoint(260,170),&font1,yy);

    // Plot fluo cam status
    if(pImg1->FluoFlag == 1){ cvPutText(pImg1->StatusImg,"ON",cvPoint(260,195),&font1,yy);}
else{ cvPutText(pImg1->StatusImg,"OFF",cvPoint(260,195),&font1,yy);}  

// Plot record status
char *astring = (char*) malloc(100*sizeof(char));
sprintf(astring,"%d",pImg1->RecordInterval);
if(pImg1->RecordFlag==1){ cvPutText(pImg1->StatusImg,(const char*) astring,cvPoint(260,210),&font1,yy);}
else{ cvPutText(pImg1->StatusImg,"OFF",cvPoint(260,220),&font1,yy);}

// Plot automation type
if (pImg1->AutoFlag == 0) { sprintf(astring,"NONE",pImg1->RecordInterval);}
elself (pImg1->joystickflag==0) { sprintf(astring,"AUTO",pImg1->RecordInterval);}
elselse if (pImg1->joystickflag==1) { sprintf(astring,"MANUAL",pImg1->RecordInterval);}
else { sprintf(astring,"ERROR",pImg1->RecordInterval);}cvPutText(pImg1->StatusImg,(const char*) astring,cvPoint(260,245),&font1,yy);

// Plot the overlay status
if (pImg1->OlFlag == 0) { sprintf(astring,"OFF");}
ellself (pImg1->OlFlag == 1 ) { sprintf(astring,"ON");}
cvPutText(pImg1->StatusImg,(const char*) astring,cvPoint(260,270),&font1,yy);

// Plot the focus point
sprintf(astring,"(%d-%d)/(%d-%d)",pImg1->StartPtIdx,pImg1->FocusPtIdx,pImg1->StartPtDist,pImg1->FocusPtDist);cvPutText(pImg1->StatusImg,(const char*) astring,cvPoint(260,295),&font1,yy);

// Plot focus point move speed
sprintf(astring,"%d",pImg1->FocusPtStep);cvPutText(pImg1->StatusImg,(const char*) astring,cvPoint(260,320),&font1,yy);

// Plot the targeting on/off status
if(pImg1->MaxFlag) { sprintf(astring,"ON");}
elselse { sprintf(astring,"OFF");}
cvPutText(pImg1->StatusImg,(const char*) astring,cvPoint(260,345),&font1,yy);

// Plot the illumination side

213
if (pImg1->sideflag==1){sprintf(astring,"Side 1");}
   else if (pImg1->sideflag==2){sprintf(astring,"Side 2");}
   else {sprintf(astring,"All");}  
cvPutText(pImg1->StatusImg,(const char*) astring,cvPoint(260,395),&font1,yy);

   // Plot the invert status yes/no
   if (pImg1->invertregionflag==1){sprintf(astring,"INVERTED");  cvPutText(pImg1->StatusImg,(const char*) astring,cvPoint(260,420),&font1,rr);}  
   else  
   {sprintf(astring,"Off");  cvPutText(pImg1->StatusImg,(const char*) astring,cvPoint(260,420),&font1,yy);}  

   // Show the image
   cvShowImage("System status",pImg1->StatusImg);
}

/*
* Check if a file exists. Return 1 if it does, 0 if it does not.
*/

int exists(const char *fname)
{
   FILE *file;
   if (file = fopen(fname, "r"))
   {
       fclose(file);
       return 1;
   }
   return 0;
}

/*
* Figure out the camera handles of all attached USB cameras
*/

int find_system_cameras(int hmin)
{
   int camid = -2;
for(int i=hmin; i<3; i++){
    CvCapture* cv_cap = cvCaptureFromCAM(i);
    IplImage *img = cvQueryFrame(cv_cap);
    if(img > 0){
        if(camid == -2){
            camid = i;
            break;
        }
    }
    cvReleaseCapture(&cv_cap);
}

if (camid > -2 && hmin<1){
    printf("Successfully connected to darkfield camera (%d).\n",camid);
} else if(camid > -2 && hmin>=1){
    printf("Successfully connected to fluoresc. camera (%d).\n\n",camid);
} else{
    printf("------>FAILED TO FIND ANY VALID USB CAMERAS.\n\n");
}
return camid;

/*
 * Draw FLUO segmentation on the image
 */

void preview_dlp_live(struct Images* pImg1, struct DlpCalib* pDlp1, struct Script* pScript){

    // Setup font
    CvFont font2;
    cvInitFont(&font2,CV_FONT_HERSHEY_SIMPLEX,.5,.5,0,1,8);

    // Get the current command and arguments
    char this_command[100]; strcpy(this_command,pScript[pDlp1->dlpscriptstep].command);
    int this_arg[6]; for(int i=0; i<6;i++){this_arg[i] = pScript[pDlp1->dlpscriptstep].arg[i];}

    // If we are in a DLP_ON, show the DLP state onscreen for the user to preview live

if(pImg1->startExptFlag==1 & !strcmp(this_command,"DLP_ON")){
    cvDrawCross(FLUO_IMAGE_CLR,pDlp1->target,ww,1,8);
    // Overwrite the red puncta track with a
    blue one

    if(pImg1->tracktypeflag!=23 & pImg1-
    >tracktypeflag!=25){
        cvEllipse(FLUO_IMAGE_CLR,*pDlp1-
        >target,
        // Also draw the inner ellipse
        cvSize(pDlp1->r2 * pDlp1->Xratio , pDlp1-
        >r2 * pDlp1->Xratio),
        0,0,360,ww,1,8);
        cvEllipse(FLUO_IMAGE_CLR,*pDlp1-
        >target,
        // Also draw an ellipse showing the
        illuminated area.
        cvSize(pDlp1->r1 * pDlp1->Xratio , pDlp1-
        >r1 * pDlp1->Xratio),
        0,0,360,ww,1,8);
    }

    if(pImg1->tracktypeflag==23 || pImg1-
    >tracktypeflag==24){
        cvDrawCross(FLUO_IMAGE_CLR,pImg1-
    >max_wing1,ww,1,4);
        // Also draw the wings
        cvDrawCross(FLUO_IMAGE_CLR,pImg1-
    >max_wing2,ww,1,4);
        cvLine(FLUO_IMAGE_CLR,*pImg1-
    >max_wing1,*pImg1-
    >max_wing2,ww,pDlp1->r1*2,8);
    }

    if(is_tracktype(pImg1->tracktypeflag,3,25,27,28)){
        cvDrawCross(FLUO_IMAGE_CLR,pImg1-
    >max_wing1,ww,1,4);
        // Also draw the wings
        cvDrawCross(FLUO_IMAGE_CLR,pImg1-
    >max_wing2,ww,1,4);
        cvDrawCross(FLUO_IMAGE_CLR,pImg1-
    >max2_wing1,ww,1,4);
        // Also draw the wings
        cvDrawCross(FLUO_IMAGE_CLR,pImg1-
    >max2_wing2,ww,1,4);
        cvRectangle4pt(FLUO_IMAGE_CLR,pImg1-
    >max_wing1,pImg1->max_wing2,pImg1->max2_wing1,pImg1->max2_wing2,ww,1);
    }

    if(pImg1->tracktypeflag==26){
        cvDrawCross(FLUO_IMAGE_CLR,pImg1-
    >max_wing1,ww,1,4);
        // Also draw the wings
        cvDrawCross(FLUO_IMAGE_CLR,pImg1-
    >max_wing2,ww,1,4);
cvEllipse(FLUO_IMAGE_CLR,*pDlp1->target,
    // Also draw an ellipse showing the illuminated area. The astigmatism is in the PVCAM image not in the DMD illumination because checkered patterns viewed on the darkfield camera look fine.
    cvSize(pDlp1->r3 * pDlp1->Xratio, pDlp1->r3* pDlp1->Xratio),
    0,0,360,ww,1,8);

    }
    
    if((pImg1->tracktypeflag >= 31 && pImg1->tracktypeflag <= 34 )&& pImg1->Contours->total>2 && pImg1->Centerline->total>35 && pImg1->OlFlag){
        cvClearSeq(pImg1->SliceShow);
        if(pDlp1->r1 < 50){
            for (int i=pImg1->StartPtIdx; i<pImg1->FocusPtIdx; i++){
                // Extract each point as a CvPoint*
                CvPoint* pt = (CvPoint*)cvGetSeqElem(pImg1->Centerline,i);
                // Feed translated point to the to-illum sequence
                cvSeqPush(pImg1->SliceShow,pt);
            }
        }
        else{
            CvPoint* pt = (CvPoint*)cvGetSeqElem(pImg1->Centerline,1);
            cvCircle(FLUO_IMAGE_CLR,*pt,pDlp1->r1,gg,2);
        }
    }
    
    if( pImg1->tracktypeflag>=51 && pImg1->tracktypeflag<=59){
        if (pDlp1->r1 < 50){
            cvDrawCross(FLUO_IMAGE_CLR,pImg1->max_wing1,ww,1,4);
            // Also draw the wings
cvDrawCross(FLUO_IMAGE_CLR,pImg1->max_wing2,ww,1,4);
            cvDrawCross(FLUO_IMAGE_CLR,pImg1->max2_wing1,ww,1,4);
            // Also draw the wings
cvDrawCross(FLUO_IMAGE_CLR,pImg1->max2_wing2,ww,1,4);
    }
    else{
        cvCircle(FLUO_IMAGE_CLR,*pImg1->max_loc,pDlp1->r1,gg,2);
    }
}

// Regardless of the DLP state, draw the metacorpus (really the tip of the pharynx) to show the program's orientation.
if(pImg1->tracktypeflag>=21 && pImg1->tracktypeflag<=29){
    cvCircle(FLUO_IMAGE_CLR,*pImg1->ph_ter,4,pp,1);
}

// If we are preparing for box illumination, draw the candidate box
if((is_tracktype(pImg1->track,5,25,27,28,51,52)) && !(pImg1->startExptFlag==1 && !strcmp(this_command,"DLP_ON"))){
    cvRectangle4pt(FLUO_IMAGE_CLR,pImg1->max_wing1,pImg1->max_wing2,pImg1->max2_wing1,pImg1->max2_wing2,rr,1);
}

// Regardless of the DLP state, draw the whole worm segmentation if using darkfield
if(pImg1->tracktypeflag>=31 && pImg1->tracktypeflag<=39 && pImg1->OlFlag){
    cvDrawPlot(FLUO_IMAGE_CLR,pImg1->Centerline,gg,1);
    cvCircle(FLUO_IMAGE_CLR,*pImg1->head,10,gg,2);
    cvDrawCross(FLUO_IMAGE_CLR,pImg1->tail,rr,2,10);
    cvCircle(FLUO_IMAGE_CLR,*pImg1->FocusPt,5,rr,-1);
    cvCircle(FLUO_IMAGE_CLR,*pImg1->StartPt,5,gg,-1);
    cvDrawPlot(FLUO_IMAGE_CLR,pImg1->Contours,bb,1);
}

// Regardless of the DLP state, draw the manual bound of illumination if using track type 12
if(pImg1->tracktypeflag ==12){
    cvDrawPlot(FLUO_IMAGE_CLR,pImg1->manualBoundClosed,rr,1);
    if(pImg1->drawstreakflag==1){
        cvRectangle(FLUO_IMAGE_CLR,cvPoint(0,0),cvPoint(FLUO_IMAGE_CLR->width,18),ww,-1,8);
        cvPutText(FLUO_IMAGE_CLR,"MANUAL DRAW MODE ACTIVATED!!",cvPoint(10,14),&font2,kk);
    }
}
/**
 * Correct the target puncta location for stage movements
 */

void compensate_target(struct Images* pImg1, struct DlpCalib* pDlp1) {

    // Determine whether the stage is commanded to move this iteration
    double diffx, diffy;

    int xbar = pImg1->fluo_img_ctr->x;
    int ybar = pImg1->fluo_img_ctr->y;
    double scalefactor = 0;

    // Use approximate scalefactor appropriate to the fluo camera in use
    // (6.67 um/pixel with 4x binning and image in small corner of sensor, 20x, 11/19/14)
    if (PVCAM_ACQUIRE_MODE_DEF<0){scalefactor = 0.75;}
    else{scalefactor=STAGE_SCALE_20X;}

    diffx = ((double) pDlp1->target->x - (double) xbar) * (scalefactor);
    diffy = -((double) pDlp1->target->y - (double) ybar) * (scalefactor);

    if (abs(diffx) > STEP_MIN || abs(diffy) > STEP_MIN) {

        // If so, adjust the target towards the image center.

        // Find the difference in PIXELS, at least to see whether we are above/below/left/right from the
        diffx = -((double) pDlp1->target->x - (double) xbar);
        diffy = -((double) pDlp1->target->y - (double) ybar);

        pDlp1->target->x += sign(diffx)*STAGE_SPEED_X_20X;
        pDlp1->target->y += sign(diffy)*STAGE_SPEED_Y_20X;
    }
}

void record_HQ_fluo(FILE *file, IplImage *img, int ct2) {
// Save the fluo frame to a text file, uncompressed (takes a long
time).
unsigned long x, y;

// Get the pixels from the input IplImage and put them in the
uchar image
for (y=0; y<NSIZEY; y++){
    for (x=0; x<NSIZEX; x++){
        fprintf(file, "%u\t", CV_IMAGE_ELEM(img, uchar, y, x));
    }
    fprintf(file, "\n");
}

/*
 * Import a single number parameter from text file.
*/
void import_param(int *value, char *fname){
    FILE *file = fopen(fname, "r");
    fscanf(file, "%i", value);
    fclose(file);
}

/*
 * Write (update) a single number parameter to text file.
*/
void update_param(int value, char *fname){
    FILE *file = fopen(fname, "w");
    fprintf(file, "%d\n", value);
    fclose(file);
}

/*--------------------------------------------------------------
----------------
Multithreading functions: Read from cameras or record
to disk in parallell.
These functions are called by threads created in the
Cam() function.
They can be called in parallell by each thread to
record images in parallell
---------------------------------------------------------------*/
int GetDarkfieldFrame(CvCapture* cap, IplImage* img, int *readyflag){
    /* Get frames forever until we interrupt the thread. */
for(;;){
    *readyflag = 0;
    if(cap !=0){
        IplImage* img2 = cvQueryFrame(cap);
        cvFlip(img2,img2,1);
        cvCopy(img2,img);
        *readyflag = 1;

        /* Wait for the frame to be retrieved OR wait for 30 ms, whichever comes first.
         * This allows frames to be read as fast as possible when the user is actually reading them.
         * At other times it updates at about 30Hz */
        int waittime = 0;
        while(*readyflag==1){boost::this_thread::sleep(boost::posix_time::milliseconds(3)); waittime +=3; if(waittime>1){*readyflag = 0;}}
    }
    /* Check whether the thread using this function is marked for termination */
    boost::this_thread::interruption_point();
    return 0;
}

int GetPvcamFrame(int hCam, uns16* frame, rgn_type region, uns32 size ,
int PVCAM_ACQUIRE_MODE, int *readyflag){
    /* Get frame when signalled. */
    for(;;){
        if(*readyflag ==1){
            // Retrieve fluorescence image
            PVCAM_acquire(hCam, frame, region, size,
            PVCAM_ACQUIRE_MODE);
            *readyflag = 0;
        }

        /* Check whether the thread using this function is marked for termination */
        boost::this_thread::interruption_point();
    }
    return 0;
}

// If signalled, write the new frame to disk. If not, sleep.
int WriteMovieToDisk(CvVideoWriter *writer, IplImage* smallimg,
IplImage* largeimg, int *readyflag){
    for(;;){
        // If signalled, write the new frame to disk. If not, sleep.
    }
}
if(*readyflag == 1) {
    if(SAVE_LARGE){
        cvResize(smallimg,largeimg);
        cvWriteFrame(writer,largeimg);
    }
    else{
        cvWriteFrame(writer,smallimg);
    }
    *readyflag = 0;
}

// Pause for a few ms
boost::this_thread::sleep(boost::posix_time::milliseconds(3));

/* Check if the thread has been marked for interruption */
boost::this_thread::interruption_point();
}

return 0;
}

// If signalled, aim the DLP at the worm. If not, sleep.
int FireDlpThread(struct Images* pImg1, struct DlpCalib *pDlp1, struct Script *pScript, int *dlpdoneflag, int *wholeillumflag, int *readyflag, int *dlptrigflag){
    int ct=0;
    for(;;){
        // If fire command occured, fire the DLP and set the readyflag back to 0.
        if(*readyflag == 1){
            Fire_DLP_by_script(pImg1, pDlp1, pScript, dlpdoneflag,wholeillumflag);
            *readyflag = 0;
        }
        else if(pImg1->scriptFlag==1){
            Blank_DLP(pImg1,pDlp1);
            *readyflag = 0;
        }*

        /* Check if the thread has been marked for interruption */
        boost::this_thread::interruption_point();
    }
    return 0;
}

// If signalled, move the stage to keep the worm's target point within the field of view
int MoveStageThread(CvPoint* fluo_img_ctr, CvPoint* max_loc,
SimpleSerial *mySerial, bool *joystickflag, int *readyflag, int *cauto,
int *stagespeed){
    for(;;){
        // Move stage automatically
        if(*readyflag == 1 && *cauto < 500){
            StageAutomate(fluo_img_ctr,*cauto,max_loc,mySerial,*joystickflag,
stagespeed);
            *readyflag = 0;
        }

        // Move stage manually if user click the arrow keys
        else if(*readyflag == 1 && *cauto >= 500){
            StageJoyStickManual(mySerial,*cauto, stagespeed);
            *readyflag = 0;
        }

        // Pause for a few ms
        boost::this_thread::sleep(boost::posix_time::milliseconds(3));

        /* Check if the thread has been marked for interruption */
        boost::this_thread::interruption_point();
    
    return 0;
}

// If signalled, display video mages to the user. Also allows user to
draw an analysis region

int DisplayImgThread(IplImage *DF_IMG, IplImage *FL_IMG, IplImage *
DLP_IMG, int fluoflag, int *c, struct Images* pImg1, char *cRate, int *
readyflag){

    // HighGUI windows must be initialized from within the thread
    where they are updated
    cvNamedWindow("Live feed",0);
    if(fluoflag== 1){cvNamedWindow("Fluo feed",0);}
    cvNamedWindow("System status",1);
    cvSetMouseCallback("Fluo feed", CallBackFunc, (void*) pImg1);
    *c = -5;

    for(;;){
        if(*readyflag == 1){

// Show the two images
if(DF_IMG != 0){cvShowImage("Live feed",DF_IMG);}
if(fluoflag== 1){cvShowImage("Fluo feed",FL_IMG);}
//cvShowImage("Dlp command",DLP_IMG);

// Apply text labels to status image frame
set_status_image(pImg1,cRate);

// Signal which key was pressed
*c = cvWaitKey(1);
*readyflag = 0;
}

if(pImg1->endLoopFlag==1){*c = 27;}

// Close all the OpenCV windows
if(*c==27){cvDestroyAllWindows();}

//-- Check if the thread has been marked for interruption */
boost::this_thread::interruption_point();

return 0;

/*---------------------------------------------------------------
----------------
Primary camera functions
- Cam() is the main action function of this code. It
executes the closed loop control and imaging
algorithm.
- CamBG() is used to calculate a background subtraction image, to correct
for uneven lighting during darkfield tracking (31-39). It now operates
using the optiMOS camera instead of the
ICcamera.
- CamThresh() [DEPRECATED] was previously used to set the threshold. Thresholding
is not done in real time during experiments, see STAGEBOOST_SHORTCUT_KEYS.txt for details.
---------------------------------------------------------------*/
void Cam(struct Images *pImg1, SimpleSerial *mySerial, struct DlpCalib *pDlp1, struct Script *pScript)
{
    // Setup some variables
    int c, cauto, stagespeed, ct2, numContours, savect, dlpdoneflag,
        displayimgdoneflag, rflag1, rflag2, rflag3, rflag4, rflag5, rflag6,
        rflag7, rflag8, rflag1ext, rflag2ext, rflag3ext, rflag4ext, prefint,
        dlptrigflag;
    double ct, tcum, rate, rateinit , max_val, min_val,
    totalrecordingtime;
    char cRate[100] = "{0 FPS}";
    char fout_df[300];
    char fout_fl[300];
    char fout_tx[300];
    char fout_txx[300];
    char fout_txy[300];
    char fout_sc[300];
    char fout_va[300];
    char flabel[300];
    char cFramePrint[300];
    char pref[300];
    char date[300];
    char expt_time[300] = "{0 s}";
    CvFont font1;
    cvInitFont(&font1,CV_FONT_HERSHEY_SIMPLEX,.5,.5,0,1,8);
    CvFont font2;
    cvInitFont(&font2,CV_FONT_HERSHEY_SIMPLEX,.345,.345,0,1,8);
    cv::Mat m = cvCreateMat(pImg1->FluoImg->height,pImg1->FluoImg->width,CV_16U); // temporary Mat used for PVCAM pixel manipulations
    CvVideoWriter *writerFl;
    CvVideoWriter *writerDf;
    CvPoint max_loc = cvPoint(0,0);
    CvPoint fluo_img_ctr = cvPoint(0,0);
    FILE *txtout;
    FILE *txtoutx;
    FILE *txtouty;
    FILE *scout;
totalrecordingtime = 0;
pImg1->endLoopFlag=0;
c = 0;
cauto = 0;
stagespeed = 50;
pImg1->joystickflag = 0;

    // If recording is enabled, set the output filename
    if(pImg1->RecordFlag==1){

225
// Get the date automatically
    time_t t = time(NULL);
    struct tm tm = *localtime(&t);

    if  
        (tm.tm_mon + 1<10  && tm.tm_mday <
10){sprintf(date,"%d-0%0%d-%0%d", tm.tm_year + 1900, tm.tm_mon + 1, 
        tm.tm_mday);}
    else if  
        (tm.tm_mon + 1>=10 && tm.tm_mday <
10){sprintf(date,"%d-%d-0%0%d", tm.tm_year + 1900, tm.tm_mon + 1, 
        tm.tm_mday);}
    else if  
        (tm.tm_mon + 1<10  && tm.tm_mday>=
10){sprintf(date,"%d-0%0%d-%d", tm.tm_year + 1900, tm.tm_mon + 1, 
        tm.tm_mday);}
    else if  
        (tm.tm_mon + 1>=10 && tm.tm_mday>=
10){sprintf(date,"%d-%d-%d", tm.tm_year + 1900, tm.tm_mon + 1, 
        tm.tm_mday);}

// Get worm number today
    system("CLS");
    cout << "Enter worm number only.\nEXAMPLE: 1\n";
    cin >> prefint;

// Get descriptive label
    cout << "\nEnter special label only.\nEXAMPLE: w\n";
    cin >> flabel;

    if(prefint < 10){sprintf(pref,"w0%d",prefint);}
    else {sprintf(pref,"w%d",prefint);}

// Assemble the output filenames
    sprintf(fout_df,"F:/Local Data/%s_%s_%s_DF.avi",date,pref,flabel);
    sprintf(fout_fl,"F:/Local Data/%s_%s_%s_FL.avi",date,pref,flabel);
    sprintf(fout_va,"F:/Local Data/%s_%s_%s_VA.txt",date,pref,flabel);
    sprintf(fout_tx,"F:/Local Data/%s_%s_%s_TX.txt",date,pref,flabel);
    sprintf(fout_txx,"F:/Local Data/%s_%s_%s_TXx.txt",date,pref,flabel);
    sprintf(fout_txy,"F:/Local Data/%s_%s_%s_TXy.txt",date,pref,flabel);
    sprintf(fout_sc,"F:/Local Data/%s_%s_%s_SC.txt",date,pref,flabel);

    // Make sure file is not about to be overwritten.
    c=0;
if(exists(fout_df)==1){
    printf("\n%s\nERROR: The requested file above (and
probably others) already exists. Overwriting is not allowed, and the
experiment will now exit to main menu.\n",fout_df);
    system("PAUSE");
    c = 27;
    return;
}
if(SAVE_LARGE){
    writerFl = cvCreateVideoWriter(fout_fl,
CV_FOURCC('M','J','P','G'), pImg1->FrameRateFixed, cvSize(pImg1->FluoImg8bitLarge->width,pImg1->FluoImg8bitLarge->height), false);
    writerDf = cvCreateVideoWriter(fout_df,
CV_FOURCC('M','J','P','G'), pImg1->FrameRateFixed, cvSize(pImg1->MainImg8bitLarge->width,pImg1->MainImg8bitLarge->height), false);
} else {
    writerFl = cvCreateVideoWriter(fout_fl,
CV_FOURCC('M','J','P','G'), pImg1->FrameRateFixed, cvSize(FLUO_IMAGE_GRY->width,FLUO_IMAGE_GRY->height), false);
    writerDf = cvCreateVideoWriter(fout_df,
CV_FOURCC('M','J','P','G'), pImg1->FrameRateFixed, cvSize(pImg1->MainImggray->width,pImg1->MainImggray->height), false);
}
txtout = fopen(fout_tx,"w");
txtoutx= fopen(fout_txx,"w");
txtouty= fopen(fout_txy,"w");

// Setup darkfield capture
CvCapture* cv_cap = (CvCapture*) malloc(sizeof(CvCapture*));
cv_cap = cvCreateCameraCapture(pImg1->CamID);
IplImage* img_color = cvQueryFrame(cv_cap);
IplImage* img_color2;

// Setup fluorescence capture, if using ICC instead of PVCAM
CvCapture* cv_cap_fl = (CvCapture*) malloc(sizeof(CvCapture*));
if(pImg1->PVCAM_ACQUIRE_MODE<0){cv_cap_fl = cvCaptureFromCAM(pImg1->Cam2ID);

cvSetCaptureProperty(cv_cap_fl,CV_CAP_PROP_AUTO_EXPOSURE,0);
    img_color2 = cvQueryFrame(cv_cap_fl);
}

// Define an imaging mask, the area of valid pixels
// STEP 3a: Set a mask to exclude non-assigned pixels near the
// bottom of the image
// Set the fluorescence image default value
cvSet(FLUO_IMAGE_CLR,cvScalar(1,1,1));
cvSet(FLUO_IMAGE_GRY,cvScalar(1,1,1));

// Disable autogain for both cameras
if(1)
{cvSetCaptureProperty(cv_cap,CV_CAP_PROP_AUTO_EXPOSURE , 0);} 
if(pImg1-
PVCAM_ACQUIRE_MODE<0){cvSetCaptureProperty(cv_cap_fl,CV_CAP_PROP_AUTO_EXPOSURE , 0);} 

// Load the user params from disk, so they "stick" between
sessions
import_param(&pImg1-fluoThresh ,PARAM_FILE_PVCAM_THRESH);
import_param(&stagespeed ,PARAM_FILE_STAGE_SPEED);

// Threading "ready" flags. These flags are shared between their
respective thread and the Cam() function to signal when a thread should
be activated vs. when it must be dormant
rflag1 = 0; // Flag keeps track, within the thread,
of whether a frame has been recorded in full. Changes 1/0 all the time
as frames read.
rflag2 = 0; // 
rflag3 = 0; // Flag keeps track, within the thread,
of whether we are ready to write the darkfield frame to disk.
rflag4 = 0; // 
rflag8 = 0; // 
rflag5 = 0; // Flag keeps track, within its thread,
of whether to start firing the DLP.
rflag6 = 0; // Flag keeps track, within its thread,
of whether to start moving the stage.
rflag7 = 2; // Flag keeps track, within its thread,
of whether to start showing frames to the user's screen. Start at state
2, which will neither start recording nor pause the program
rflag1ext = 0; // Flag OUTSIDE the thread, keeps track of
whether a frame has been recorded at all. Stays at 1 forever when true
at any time.
rflag2ext = 0; // 
rflag3ext = 0; // Flag OUTSIDE the thread, keeps track of
whether a frame has been written at all. Stays at 1 forever when true
at any time within the current loop iteration.
rflag4ext = 0; //
displayimgdoneflag = 1;
dlptrigflag = 1; // Flag keeps track of whether the DLP has been triggered on this iteration or not. We must wait for it to be triggered, AND wait for it to be finished.

// Setup some temporary / dummy images
IplImage *img_color_th = cvCloneImage(pImg1->MainImg); // Camera image (df)
IplImage *img_color_f1 = cvCloneImage(FLUO_IMAGE_CLR); // Camera image (fluo)
IplImage *DF_IMG = cvCloneImage(pImg1->MainImg); // Display image (df)
IplImage *FL_IMG = cvCloneImage(FLUO_IMAGE_CLR); // Display image (fluo)
IplImage *DLP_IMG = cvCloneImage(pImg1->ProtoDlpImage); // DLP image (deprecated)
IplImage *TempGrayImage = cvCloneImage(pImg1->FluoImgThresh); // ROI-selecting image multiplied by the thresh image

// Start up the camera threads. Frames will be grabbed from them as needed. Can use either PVCAM (mode=[0:2]) or ICC (mode=-1)
boost::thread t1(&GetDarkfieldFrame, cv_cap, img_color_th, &rflag1);
boost::thread t2(FLUO_THREAD_PARAMETERS);

// Start up the recording threads. These will record a frame to disk only when signalled.
boost::thread t3(&WriteMovieToDisk, writerDf, pImg1->MainImggray, pImg1->MainImg8bitLarge, &rflag3);
boost::thread t4(&WriteMovieToDisk, writerFl, pImg1->FluoImgGray8bitWrite, pImg1->FluoImgGray8bitLarge, &rflag4);

// Start up the DLP firing thread. This will start aiming the DLP only when signalled
boost::thread t5(&FireDlpThread, pImg1, pDlp1, pScript, &dlpdoneflag, &pImg1->tracktypeflag, &rflag5, &dlptrigflag);

// Start up the stage automation thread
boost::thread t6(&MoveStageThread, &fluo_img_ctr, &max_loc, mySerial, &pImg1->joystickflag, &rflag6, &cauto, &stagespeed);

// Start up the image display thread
boost::thread t7(&DisplayImgThread, DF_IMG, FL_IMG, DLP_IMG, pImg1->FluoFlag, &c, pImg1, cRate, &rflag7);

// Main loop where all frame grabbing and processing happens
ct = 0;
ct2 = 0;
tcum = 0;
rate=0;
savect = 0;
dlpdoneflag = 0;
double waittime = 0;

for(;;){
    tic;
tic3;
    ct=ct+1;    // Number of iters since last FPS reading
    (reset frequently)
    ct2 = ct2+1;    // Total number of iters in this execution
    of Cam()
    pImg1->startExptFlag=pImg1->startExptFlagSignal;    // start
    experiment on this iteration if the mouse callback says so.

    rflag1ext = 0;
rflag2ext = 0;

    // If indicated, retrieve darkfield video frame only
    if(pImg1->FluoFlag<1){
        while(!(rflag1ext)){
            if(rflag1==1){rflag1ext =1;}
        }
        // Recover the darkfield frame from its thread.
cvCopy(img_color_th,pImg1->MainImg);
        //cvFlip(pImg1->MainImg,pImg1->MainImg,0);
        //cvFlip(pImg1->MainImg,pImg1->MainImg,1);
    }
    // If fluorescence is requested, retrieve both frames in
    paalell
    else{
        // Wait for both frames to be finished writing to
disk from the previous iteration
        while(rflag3==1){}
        while(rflag4==1){}

        // Wait for the dlp to be done aiming on the previous
        iteration. (can't allow framerate to be faster than DLP update rate)
        while(rflag5==1){} 

        // Get PVCAM frame
        boost::this_thread::sleep(boost::posix_time::milliseconds(1.5*pImg1->exp_time));    // Wait a moment (don't want old frames)
rflag2 = 1;
while(rflag2==1){};

// Recover the darkfield frame from its thread.
Rotate it. Some frames may be duplicated because IC is not as fast as optiMOS.
cvCopy(img_color_th ,pImg1->MainImg);
cvFlip(pImg1->MainImg,pImg1->MainImg,0);
cvFlip(pImg1->MainImg,pImg1->MainImg,1);

// Recover the fluorescence frame from its thread
if(pImg1->PVCAM_ACQUIRE_MODE>=0){
    frame2Ipl(pImg1->frame,pImg1->FluoImgGray,&m,&pImg1->region);
cvFlip(pImg1->FluoImgGray,pImg1->FluoImgGray,0);
cvFlip(pImg1->FluoImgGray,pImg1->FluoImgGray,1);
} else{
    cvCopy(img_color_f1,FLUO_IMAGE_CLR);
cvFlip(FLUO_IMAGE_CLR,FLUO_IMAGE_CLR);

cvCvtColor(FLUO_IMAGE_CLR,FLUO_IMAGE_GRY,CV_RGB2GRAY);
}

// Pre-process the retrieved fluo image
if(PVCAM_ACQUIRE_MODE_DEF>=0){
    // Subtract background to enhance contrast
    if(1){
        cvSub(FLUO_IMAGE_GRY,pImg1->BGavg,FLUO_IMAGE_GRY);
cvSub(FLUO_IMAGE_CLR,pImg1->BGavgColor,FLUO_IMAGE_CLR);
    }
    // Optionally square the image to enhance contrast
    if(pImg1->squareFlag){
        cvMul(FLUO_IMAGE_GRY,FLUO_IMAGE_GRY,FLUO_IMAGE_GRY);
    }
    mulByScalar(FLUO_IMAGE_GRY,FLUO_IMAGE_GRY,pImg1->FluoImgGrayTemp,10);
}
// Allow the user to toggle powering the image on/off

if (c==112 && pImg1->squareFlag==1) {
pImg1->squareFlag = 0; c=-5; printf("Power 2 image turned OFF\n");}
else if (c==112 && pImg1->squareFlag==0) {
pImg1->squareFlag = 1; c=-5; printf("Power 2 image turned ON\n");}

// Invert (if using brightfield)

if(INVERT_MODE){imcomplement(FLUO_IMAGE_GRY,pImg1->FluoImgGrayTemp);}

// Black out the top and bottom few pixels

if(THRESH_MODE == CV_THRESH_BINARY){
    cvRectangle(FLUO_IMAGE_GRY,cvPoint(0,FLUO_IMAGE_GRY->height-5),cvPoint(FLUO_IMAGE_GRY->width,FLUO_IMAGE_GRY->height),kk,-1);
    cvRectangle(FLUO_IMAGE_GRY,cvPoint(0,0),cvPoint(FLUO_IMAGE_GRY->width,2),kk,-1);
}
else{
    cvRectangle(FLUO_IMAGE_GRY,cvPoint(0,FLUO_IMAGE_GRY->height-5),cvPoint(FLUO_IMAGE_GRY->width,FLUO_IMAGE_GRY->height),ww,-1);
    cvRectangle(FLUO_IMAGE_GRY,cvPoint(0,0),cvPoint(FLUO_IMAGE_GRY->width,2),ww,-1);
}

// If using PVCAM, switch the image to an 8bit. Thresholding in openCV needs 8bit for some reason.
    cvConvertScale(FLUO_IMAGE_GRY,pImg1->FluoImgGray8bit,1./256);
    cvCvtColor(FLUO_IMAGE_GRY,FLUO_IMAGE_CLR,CV_GRAY2RGB);

    // Copy the gray image to the validation image (before normalizing)
    cvCopy(FLUO_IMAGE_GRY,pImg1->FluoImgGrayValid);

    // Convert a color image version to a normalized image if needed for viewing high contrast. The raw pixel data sent to segmentation (8 bit) is not affected.
    normalize2uint16(FLUO_IMAGE_GRY, FLUO_IMAGE_GRY, pImg1->FluoImgGrayTemp,pImg1->mask);
cvCvtColor(FLUO_IMAGE_GRY,FLUO_IMAGE_CLR,CV_GRAY2RGB);

// Threshold the fluorescent image (i.e. for pharynx)
// Thresholding in openCV must be 8 bit for some reason.
// Darkfield tracking does not require thresholding here, so modes 31-39 are excluded.
if( (pImg1->tracktypeflag>=20 && pImg1->tracktypeflag<=29) || (pImg1->tracktypeflag>=51 && pImg1->tracktypeflag<=59) ){

    // Threshold
    cvThreshold(pImg1->FluoImgGray8bit,pImg1->FluoImgThresh,pImg1->fluoThresh,255,CV_THRESH_BINARY);

    // Check that at least 1 pixel has a value >0.
    if not, set a dummy region to 255 to prevent crashes
    cvRectangle(pImg1->FluoImgThresh,cvPoint(pImg1->fluo_img_ctr->x-1,pImg1->fluo_img_ctr->y-1),cvPoint(pImg1->fluo_img_ctr->x+1,pImg1->fluo_img_ctr->y+1),ww,-1);

    // Delete the GFP channel from segmentation
    if(FITC_MODE){
        cvRectangle(pImg1->FluoImgThresh,cvPoint(0,FLUO_IMAGE_GRY->height/2),cvPoint(FLUO_IMAGE_GRY->width,FLUO_IMAGE_GRY->height),kk,1);
    }

    // Delete small objects
    isolate_largest_object(pImg1->FluoImgThresh,pImg1->FluoImgThresh,pImg1->tempMem);

    // find the outer contours
    cvFindContours(pImg1->FluoImgThresh,pImg1->contourMem,&pImg1->Contours);

    //Draw the contours on the user's image
    cvDrawContours(FLUO_IMAGE_CLR,pImg1->Contours,yy,yy,1);
}

    // Convert grayscale image
cvCvtColor(pImg1->MainImg,pImg1->MainImggray,CV_RGB2GRAY);

    // If writing to disk is requested, start the frame writing right away. It requires no further processing.

    if(pImg1->RecordFlag == 1 && ct2%SAVE_INTERVAL == 0){
    // Normalize it to uint8 first. Do not alter the image used for calculation.
    //normalize2uint8(pImg1->FluoImgGray8bit,pImg1->FluoImgGray8bitWrite,pImg1->FluoImgGray8bitTemp,pImg1->mask);
    cvCopy(pImg1->FluoImgGray8bitWrite)
    >FluoImgGray8bitWrite);
    if(SAVE_LARGE == 1){
    mulByScalar(pImg1->FluoImgGray8bitWrite,pImg1->FluoImgGray8bitWrite,pImg1->FluoImgGray8bitTemp,2);
    }
    } // Check whether the appropriate save mode is used
    if(pImg1->tracktypeflag==12 && SAVE_LARGE==0){printf("WARNING: SAVE_LARGE is off while track type is 12\n");}
    else if (pImg1->tracktypeflag!=12 && SAVE_LARGE == 1){printf("WARNING: SAVE_LARGE is on while track type is not 12\n");}
    rflag3 = 1;
    rflag4 = 1;
}

    // If requested, popup the camera settings windows (press "c")
    if(c==99)
    {cvSetCaptureProperty(cv_cap,CV_CAP_PROP_SETTINGS , 1); c=-5;}
    if(c==99 && pImg1->PVCAM_ACQUIRE_MODE<0){cvSetCaptureProperty(cv_cap_fl,CV_CAP_PROP_SETTINGS , 1);c=-5;}

    // If requested, process the fluorescence frame (from PVCAM), and track the puncta
    char fluotmessage[100];
    if(pImg1->FluoFlag == 1){
    // Select tracking program based on the user's input at startup

    234
if((pImg1->tracktypeflag==1 || pImg1->tracktypeflag==11) && pImg1->MaxFlag){
    track_puncta(pImg1,font1);
} else if (pImg1->tracktypeflag==12){
    // Note: 12 does not use puncta tracking (or any tracking). It relies on manual user input. Don't do anything here.
} else if(pImg1->tracktypeflag>=20 && pImg1->tracktypeflag<=29 && pImg1->MaxFlag){
    track_pharynx(pImg1,font1);
} else if(pImg1->tracktypeflag==4 && pImg1->MaxFlag){
    track_dummy(pImg1,ct2);
} else if(pImg1->tracktypeflag>=31 && pImg1->tracktypeflag<=39 && pImg1->MaxFlag){
    track_whole_worm(pImg1,0);
} else if(pImg1->tracktypeflag>=51 && pImg1->tracktypeflag<=59 && pImg1->MaxFlag){
    track_pharynx(pImg1,font1);
} else if(pImg1->MaxFlag){
    printf("ERROR: FAILED TO ROUTE TRACKING METHOD REQUEST in CAM()\n");
}

// Allow user to update the tracked point and overlay status for darkfield methods. However, if the tracked point has been // hard coded into the script (DLP_ON argument 4), user commands will be ignored.

if(c==102)
{manualHeadTailFlip(pImg1); c=-5;}
if(c==103)
{pImg1->sideflag+=1; if(pImg1->sideflag==3){pImg1->sideflag=0;} printf("Sideflag=%d\n",pImg1->sideflag);}
if(c==111 && pImg1->OlFlag == 0)
{pImg1->OlFlag = 1; c=-5;}
if(c==111 && pImg1->OlFlag == 1)
{pImg1->OlFlag = 0; c=-5;}

235
if(pImg1->startExptFlag==1 &&
!strcmp(pScript[pDlp1->dlpscriptstep].command,"DLP_ON") &&

pScript[0].arg[2]){
        printf("Script coded focus pt is
out of bounds -- ABORTING!\n");
    } else{
        pImg1->FocusPtIdx = pScript[pDlp1->dlpscriptstep].arg[4];
    }
}

else if(pImg1->tracktypeflag!=32){

    // All modes except 32 use manual
    specification of tracking point index
    if(c==117 && pImg1->FocusPtIdx < pImg1->NumSteps)
        {pImg1->FocusPtIdx = pImg1->FocusPtIdx + pImg1->FocusPtStep; c=-5;}
    if(c==100 && pImg1->FocusPtIdx > 1 &&
pImg1->FocusPtIdx >= pImg1->StartPtIdx )
        {pImg1->FocusPtIdx = pImg1->FocusPtIdx - pImg1->FocusPtStep; c=-5;}
    if(c==121 && pImg1->StartPtIdx < pImg1->FocusPtIdx)
        {pImg1->StartPtIdx = pImg1->StartPtIdx + pImg1->FocusPtStep;
pImg1->FocusPtDist  = pImg1->FocusPtDist + pImg1->FocusPtStep; c=-5;}
    } else {
        // Mode 32 uses distance specification of
        tracking point index, in case part of the worm is offscreen
        if(c==117) {pImg1->FocusPtDist = pImg1->FocusPtDist + pImg1->FocusPtStep; c=-5;}
        if(c==100) {pImg1->FocusPtDist = pImg1->FocusPtDist - pImg1->FocusPtStep; c=-5;}
        if(c==121) {pImg1->StartPtDist = pImg1->StartPtDist + pImg1->FocusPtStep;
pImg1->FocusPtDist  = pImg1->FocusPtDist  + pImg1->FocusPtStep; c=-5;}
    }

236
if(c==115) {pImg1->StartPtDist = pImg1->FocusPtStep; pImg1->FocusPtDist = pImg1->FocusPtStep; c=-5;}
}

// Let the user change the step size of the focus point
if(c==54) {pImg1->FocusPtStep = 10; c=-5;}
if(c==51) {pImg1->FocusPtStep = 1; c=-5;}

// Forecast the tracked puncta based on the puncta's velocity, but only if starting an experiment. WARNING: puncta forecast interferes with handling of user drawn regions (type 12)
if(pImg1->startAutoFlag==1 && pImg1->tracktypeflag!=12){forecast_puncta(pImg1);}

// Turn tracking on or off using the "+" key. On by default.
if(c==43 && pImg1->MaxFlag==0){pImg1->MaxFlag=1; c=-5;}
if(c==43 && pImg1->MaxFlag==1){pImg1->MaxFlag=0; c=-5;}

// Allow user to draw a manual "streak" of illumination (tracking mode 12)
if(c==109 && pImg1->tracktypeflag==12){pImg1->drawstreakflag=1;printf("draw streak flag = %d\n",pImg1->drawstreakflag); *pImg1->max_loc_old = *pImg1->max_loc; c=-5;}
if(c==109 && pImg1->drawstreakflag==1 && pImg1->tracktypeflag==12){pImg1->drawstreakflag=0;printf("draw streak flag = %d\n",pImg1->drawstreakflag); c=-5;}
if(c==110 && pImg1->tracktypeflag==12){
if(pImg1->manualBound->total>0){
    CvPoint* firstManualPoint=(CvPoint*) cvGetSeqElem(pImg1->manualBound,0);
    cvSeqPop(pImg1->manualBoundClosed);
    cvSeqPop(pImg1->manualBoundClosed); cvSeqPush(pImg1->manualBoundClosed,firstManualPoint);
    cvSeqPop(pImg1->manualBound); }
    c=-5; }
}
if(pImg1->tracktypeflag==12 && pImg1->drawstreakflag && pImg1->clickEvent == CV_EVENT_LBUTTONDOWN){
    // Add the currently clicked point to the manual bound instead of moving the tracked point
    cvSeqPush(pImg1->manualBound,pImg1->max_loc);

    // Also add the currently clicked point to the closed manual bound and close it.
    pImg1->manualBoundClosed = cvCloneSeq(pImg1->manualBound,pImg1->manualBoundClosedMem);
    CvPoint* firstManualPoint= (CvPoint*)cvGetSeqElem(pImg1->manualBound,0);
    cvSeqPush(pImg1->manualBoundClosed,firstManualPoint);
    *pImg1->max_loc = *pImg1->max_loc_old;
}

if(pImg1->tracktypeflag==12){
    // Copy the manual region to the centerline variable so it is written to disk.
    pImg1->Centerline = cvCloneSeq(pImg1->manualBoundClosed,pImg1->centerlineMem);
    pImg1->clickEvent = 0;
}

// Invert the targeting region image
if(c==105 && pImg1->invertregionflag==0){pImg1->invertregionflag=1; c=-5; printf("invert region flag = %d\n",pImg1->invertregionflag);}
if(c==105 && pImg1->invertregionflag==1){pImg1->invertregionflag=0; c=-5; printf("invert region flag = %d\n",pImg1->invertregionflag);}

// Allow the user to adjust the DLP calibration in real time. 1/2 move p00x up and down. 4/5 move p00y up and down. Changes must be MANUALLY recorded
if(c==49){pDlp1->p00x += 2;
    printf("p00x=%f\n",pDlp1->p00x); c=-5;}
if(c==50){pDlp1->p00x -= 2;
    printf("p00x=%f\n",pDlp1->p00x); c=-5;}
if(c==52){pDlp1->p00y += 2;
    printf("p00y=%f\n",pDlp1->p00y); c=-5;}
if(c==53){pDlp1->p00y -= 2;
    printf("p00y=%f\n",pDlp1->p00y); c=-5;}

238
// Allow the user to adjust the threshold in real time
if (c==56) {pImg1->fluoThresh += 2;
printf("FluoImg Threshold = %d\n", pImg1->fluoThresh); c=-2;}
if (c==55) {pImg1->fluoThresh -= 2;
printf("FluoImg Threshold = %d\n", pImg1->fluoThresh); c=-2;}

// Draw the puncta tracking
cvDrawCross(FLUO_IMAGE_CLR, pImg1->max_loc, rr, 1, pDlp1->r1/2);

// Draw the image center
if (pImg1->tracktypeflag>=31 && pImg1->tracktypeflag<=39) {max_loc = *pImg1->head; // For stage automation
else {max_loc = *pImg1->max_loc;} // For stage automation

// For DLP aiming
*pDlp1->target = *pImg1->max_loc;
*pDlp1->target_wing1 = *pImg1->max_wing1;
*pDlp1->target_wing2 = *pImg1->max_wing2;
*pDlp1->target_wing3 = *pImg1->max2_wing1;
*pDlp1->target_wing4 = *pImg1->max2_wing2;

// Compensate the tracking puncta for stage motion
//if(pImg1->AutoFlag == 1 && pImg1->startAutoFlag == 1 ) {compensate_target(pImg1, pDlp1);}
// Print user instructions as long as the experiment hasn't started yet
if(pImg1->startExptFlag<1){cvPutText(FLUO_IMAGE_CLR,"(•) starts expt. Lft click tracks",cvPoint(10,pImg1->FluoImg->height-6),&font2,rr); pDlp1->dlpscriptstep=2; pDlp1->dlptime = 0;} else if(pImg1->startExptFlag==1){sprintf(expt_time,"T=%f s",pDlp1->totaldlptime);
cvPutText(FLUO_IMAGE_CLR,expt_time,cvPoint(10,pImg1->FluoImg->height-6),&font2,rr);}     

// Print system data to disk
if(pImg1->RecordFlag==1 && ct2%SAVE_INTERVAL == 0){fprintf(txtout, "%f\t%f\t%f\t%f\t%f\t%f\t%f\n", totalrecordingtime,pImg1->startExptFlag,pDlp1->dlpscriptstep,
pDlp1->target->x, pDlp1->target->y, pImg1->fluo_img_ctr->x, pImg1->fluo_img_ctr->y,
pImg1->max_wing1->x, pImg1->max_wing1->y, pImg1->max_wing2->x, pImg1->max_wing2->y,
pImg1->max2_wing1->x, pImg1->max2_wing1->y, pImg1->max2_wing2->x, pImg1->max2_wing2->y,
pImg1->FocusPtIdx , pImg1->NumSteps, pImg1->FocusPtDist, pImg1->StartPtIdx,
pImg1->gfp_val);}
// Print worm CLINE data to disk
if(pImg1->RecordFlag==1 && ct2%SAVE_INTERVAL == 0){

    //Print frame time
    fprintf(txtoutx,"%f\t",totalrecordingtime);
    fprintf(txtouty,"%f\t",totalrecordingtime);
    for(int i = 0; i < pImg1->Centerline->total; i++){
        // Get current point
        CvPoint* this_point = (CvPoint*)cvGetSeqElem(pImg1->Centerline,i);
        // Print x point
        fprintf(txtoutx,"%d\t",this_point->x);
        //Print y point
        fprintf(txtouty,"%d\t",this_point->y);
    }
    //Print new line
    fprintf(txtoutx,"\n");
    fprintf(txtouty,"\n");
}

    // If requested and possible, target the worm with the DLP.
    if (pImg1->FluoFlag == 1 && pImg1->scriptFlag==1 && pImg1->startExptFlag==1){rflag5 = 1;}
    if (dlpdoneflag == 1){pImg1->startExptFlag=0; pImg1->startExptFlagSignal =0;} // When the script is done, end the experiment (but don't exit the tracking)

    // Allow the user to turn on/off the virtual joystick (j)
    if(c==106){StageHalt(mySerial); pImg1->joystickflag = !pImg1->joystickflag; c=-5; //printf("Joystick flag: %d\n",pImg1->joystickflag);
    }

    // Force-disable stage movement in tracking mode 12
    if(pImg1->tracktypeflag==12){pImg1->AutoFlag=0;
    pImg1->startAutoFlag = 0;}

241
// Move the worm towards the center
cauto = c;
    if (pImg1->AutoFlag == 1 && pImg1->startAutoFlag == 1){rflag6 = 1;}

// If indicated, use the fluorescence centroid as the focus point
    if (c==103){pImg1->targetFlag = 0; c=-5;}
    if (c==104){pImg1->targetFlag = 1; c=-5;}

// If we are in a stimulation, show the blue cross instead of red one
    preview_dlp_live(pImg1,pDlp1,pScript);

// Show EVERY OTHER frame, because this takes some time.
if (ct2%SHOW_INTERVAL==1){
    // Wait for the images to be finished displaying to the user from the previous iteration
    while (rflag7==1){}

    // Show video frames
    cvCopy(pImg1->MainImg,DF_IMG);
    cvCopy(FLUO_IMAGE_CLR,FL_IMG);
    rflag7 = 1;
}

// break loop if esc is pressed
if (c==27){break; c=-5;}
else if (pImg1->endLoopFlag){break;}

// Hold the framerate constant
toc3;
    double frametimemax = 1/pImg1->FrameRateFixed;
    waittime = max(frametimemax-diff3,0.0);

    boost::this_thread::sleep(boost::posix_time::milliseconds(waittime*1000));

// Update the DLP timer in case we are running scripts.
toc;
    totalrecordingtime += diff;
    pDlp1->totalrecordingtime = totalrecordingtime;
    pImg1->dt = diff;
    if (pImg1->startExptFlag==1){
        pDlp1->dlptime += diff;
        pDlp1->totaldlptime += diff;
    }
else{pDlp1->totaldlptime=0;}

242
getFrameRate(&ct,&diff,&tcum, &rate);
pImg1->fps = rate;
// set aside the un-rounded FPS value for use in forecasting
round(&rate);
sprintf(cRate,"%d/%d FPS",(int) rate, (int) pImg1->FrameRateFixed);

// Reset the ready flags for the threads, indicating that
we are ready to grab a new frame.
rflag1ext = 0;
rflag2ext = 0;

} /* Loop Shutdown */

// Cancel all camera and writing threads. Then wait for them to
actually be cancelled.
t1.interrupt();
t2.interrupt();
t3.interrupt();
t4.interrupt();
t5.interrupt();
t6.interrupt();
t7.interrupt();
t1.join();
t2.join();
t3.join();
t4.join();
t5.join();
t6.join();
t7.join();

// Halt stage motion
pImg1->joystickFlag = 0;
pImg1->startExptFlag = 0;
pImg1->startExptFlagSignal = 0;
StageHalt(mySerial);

// Release the darkfield camera capture
cvReleaseCapture(&cv_cap);
if(pImg1->PVCAM_ACQUIRE_MODE<0){cvReleaseCapture(&cv_cap_fl);} 

// Close video recorders if used
if(pImg1->RecordFlag==1){
    cvReleaseVideoWriter(&writerDf);
    cvReleaseVideoWriter(&writerFl);
}
// Close the ICcam property windows
cvSetCaptureProperty(cv_cap,CV_CAP_PROP_SETTINGS,0);
if(pImg1->PVCAM_ACQUIRE_MODE<0){cvSetCaptureProperty(cv_cap_fl,CV_CAP_PROP_SETTINGS,0);}

// Release the temporary matrix
m.release();

// Stop the file writing, print the script to file.
if(pImg1->RecordFlag==1){
   fclose(txtout);
   fclose(txtoutx);
   fclose(txtouty);
   scrout = fopen(fout_sc,"w");
   fprint_script(pScript,scrout);
   fclose(scrout);
}

// Reset the experiment starter flag
pImg1->startExptFlag=0;

// Save the fluo threshold to disk so it can be retrieved next time
update_param(pImg1->fluoThresh ,PARAM_FILE_PVCAM_THRESH);
update_param(stagespeed ,PARAM_FILE_STAGE_SPEED);

// Write the updated calibration values to disk, in case the user fine-tuned them
update_dlp_calib_file(pDlp1);
printf("AFTER UPDATE");

// Go back to the main menu
return;

void CamBG(struct Images *pImg1)
{
   // Setup some variables
   int c = -99;
   double ct, tcum, rate;
   char cRate[100] = {"0 FPS"};
   char cImg1[100] = {"Background #1: Not assigned"};
   char cImg2[100] = {"Background #2: Not assigned"};
   char cImg3[100] = {"Background #3: Not assigned"};
   CvCapture* cv_cap = cvCaptureFromCAM(pImg1->CamID);
CvFont font1;
cvInitFont(&font1,CV_FONT_HERSHEY_SIMPLEX,.5,.5,0,1,8);
cvNamedWindow("Fluo feed",0);
cv::Mat m = cvCreateMat(pImg1->FluoImg->height,pImg1->FluoImg->width,CV_16U); // temporary Mat used for PVCAM pixel manipulations

// Main loop where all frame grabbing and processing happens
cnt = 0;
tcum=0;
rate = 0;

// Run for a while to let user get background frames frames
for(;;){
    cnt=ct+1;
tic;

    // Retrieve video frame
    if(pImg1->PVCAM_ACQUIRE_MODE>=0){
        boost::this_thread::sleep(boost::posix_time::milliseconds(10));
PVCAM_acquire( pImg1->hCam, pImg1->frame, pImg1->region, pImg1->size, pImg1->PVCAM_ACQUIRE_MODE);
frame2Ipl(pImg1->frame,FLUO_IMAGE_GRY,&m,&pImg1->region);

        // Flip the image
        cvFlip(FLUO_IMAGE_GRY,FLUO_IMAGE_GRY,0);
cvFlip(FLUO_IMAGE_GRY,FLUO_IMAGE_GRY,1);

        // Invert (if using brightfield)
        if(INVERT_MODE){imcomplement(FLUO_IMAGE_GRY,pImg1->FluoImgGrayTemp);}

        // convert. Normalize for viewing only, not for subtraction
        normalize2uint16(FLUO_IMAGE_GRY,pImg1->FluoImgGrayTemp,pImg1->FluoImgGrayTemp2,pImg1->mask);
cvCvtColor(pImg1->FluoImgGrayTemp,FLUO_IMAGE_CLR,CV_GRAY2RGB);
    }
else{printf("PVCAM mode indicates a non-PVCAM camera\n");}

    // Apply text labels to video frame
    cvPutText(FLUO_IMAGE_CLR,"MOVE CAMERA TO BACKGROUND LOCATION",cvPoint(2,15),&font1,rr);
cvPutText(FLUO_IMAGE_CLR,"[1,2,3] assigns current frame to background 1, 2 or 3",cvPoint(2,35),&font1,rr);
cvPutText(FLUO_IMAGE_CLR,cRate,cvPoint(2,55),&font1,rr);
cvPutText(FLUO_IMAGE_CLR,cImg1,cvPoint(2,75),&font1,rr);
cvPutText(FLUO_IMAGE_CLR,cImg2,cvPoint(2,95),&font1,rr);
cvPutText(FLUO_IMAGE_CLR,cImg3,cvPoint(2,115),&font1,rr);
cvPutText(FLUO_IMAGE_CLR,"ESC averages [1-3] and then returns to the menu",cvPoint(2,135),&font1,rr);

// break loop if esc is pressed
if(c==27){
    // If all 3 backgrounds were specified, save the averaged image
    if(!strcmp(cImg1,"Background #1: Saved") && !strcmp(cImg2,"Background #2: Saved") && !strcmp(cImg3,"Background #3: Saved")){
        // Average 3 images:
        // STEP 1: (A+B)/2
        IplImage* temp = cvCreateImage(cvGetSize(FLUO_IMAGE_GRY),IPL_DEPTH_16U,1);
        IplImage* tempB = cvCreateImage(cvGetSize(FLUO_IMAGE_GRY),IPL_DEPTH_16U,1);
        cvAddWeighted(pImg1-BG1gray,0.5,pImg1-BG2gray,0.5,0,temp);

        // STEP 2: (A+B)/2/(3/2) = (A+B) / 3
        divByScalar(temp, tempB, 3/2);

        // STEP 3: (A+B) / 3 + C/3
        cvAddWeighted(temp,0.5,tempB,0.5,0,pImg1-BGavg);

        // Also save a color version
        cvCvtColor(pImg1-BGavg,pImg1-BGavgColor,CV_GRAY2BGR);
        pImg1->SubtrFlag = 1;
        cv::Mat mwrite(pImg1->BGavg);
        cv::imwrite("temp_BGavg.png",mwrite);
    }
    break;
}

// Save gray versions of the background image. Convert to MAT before saving 16 bit images.
if(c== 49){
    cvCopy(FLUO_IMAGE_GRY,pImg1->BG1gray);
cvSmooth(pImg1->BG1gray, pImg1->BG1gray, CV_GAUSSIAN, 25, 25);
cv::Mat mwrite(pImg1->BG1gray);
cv::imwrite("temp_BG1.png", mwrite);
sprintf(cImg1, "Background #1: Saved");
}
if(c==50){
    cvCopy(FLUO_IMAGE_GRY, pImg1->BG2gray);
    cvSmooth(pImg1->BG2gray, pImg1->BG2gray, CV_GAUSSIAN, 25, 25);
    cv::Mat mwrite(pImg1->BG2gray);
    cv::imwrite("temp_BG2.png", mwrite);
    sprintf(cImg2, "Background #2: Saved");
}
if(c==51){
    cvCopy(FLUO_IMAGE_GRY, pImg1->BG3gray);
    cvSmooth(pImg1->BG3gray, pImg1->BG3gray, CV_GAUSSIAN, 25, 25);
    cv::Mat mwrite(pImg1->BG3gray);
    cv::imwrite("temp_BG3.png", mwrite);
    sprintf(cImg3, "Background #3: Saved");
}

// Show video frame if valid frame address
if(FLUO_IMAGE_CLR != 0){cvShowImage("Fluo feed", FLUO_IMAGE_CLR);}
c = cvWaitKey(1);

toc;
getFrameRate(&ct, &diff, &tcum, &rate);
round(&rate);
sprintf(cRate, "%d FPS", rate);

// Close Program
cvReleaseCapture(&cv_cap);
cvDestroyWindow("Fluo feed");
return;
AnthonysColors.h
/*
 * AnthonysColors.h
 * Anthony Fouad
 * Fang-Yen Group, 6/2014
 *
 * Contains a series of MATALB-like color definitions for quick use with OPENCV.
 */
#endif ANTHONYSCOLORS_H_
#define ANTHONYSCOLORS_H_
#include "opencv\cv.h"
#include "stdafx.h"
#define bb CV_RGB( 0 , 0 , 65500 )
#define gg CV_RGB( 0 , 65500 , 0 )
#define yy CV_RGB( 65500 , 65500 , 0 )
#define kk CV_RGB( 0 , 0 , 0 )
#define pp CV_RGB( 65500 , 0 , 65500 )
#define ww CV_RGB( 65500 , 65500 , 65500 )
#define rr CV_RGB( 65500 , 0 , 0 )
#define mm CV_RGB( 65500 , 65500*21/255, 65500*179/255 )
#endif

AnthonysTimer.h
/*
 * AnthonysTimer.h
 * Anthony Fouad
 * Fang-Yen Group, 6/2014
 *
 * Contains a MATLAB-like tic toc timer. It is applied just like in MATLAB: put "tic" to start and "toc" to end the timer. Optionally display results.
 * The timer is used for frame rate calculation in AnthonysCamFuncs -> Cam()
#ifndef ANTHONYSTIMER_H_
#define ANTHONYSTIMER_H_
#include "time.h"

#define tic time_t tstart, tend; tstart = clock();
#define toc tend = clock(); double diff = (((double)tend -
(double)tstart) / 1000000.0F ) * 1000; //std::cout << "It took "<< diff
<<" second(s)."<< std::endl;

// separate tictoc definition for printing out little segments' performance.
#define tic2 time_t tstart2, tend2; tstart2 = clock();
#define toc2 tend2 = clock(); double diff2 = (((double)tend2 -
(double)tstart2) / 1000000.0F ) * 1000; std::cout << "It took "<< diff2
<<" second(s)."<< std::endl;

// separate tictoc definition for determining if the Cam() main loop is exceeding the commanded framerate
#define tic3 time_t tstart3, tend3; tstart3 = clock();
#define toc3 tend3 = clock(); double diff3 = (((double)tend3 -
(double)tstart3) / 1000000.0F ) * 1000; // std::cout << "It took "<<
diff3 <<" second(s)."<< std::endl;
#endif

DlpAccessoryFuncs.h
/*
 * DlpAccessoryFuncs.h
 * Anthony Fouad
 * Fang-Yen Group, 6/2014
 * *
 * Declares the functions for controlling the DLP. Derived heavily from Leifer & Fang-Yen, 2010
 */

#ifndef DLPACCESSORYFUNCS_H_
#define DLPACCESSORYFUNCS_H_

long T2DLP_on();

//returns the ID of the DMD

249
```c
int T2DLP_off(long alpid);

// takes an ID of the DMD
int T2DLP_SendFrame(unsigned char *image, long alpid);

int T2DLP_clear(long myDLP);

// clear the DLP mirrors
void Ipl2uchar(IplImage* imgin, unsigned char *imgout);
unsigned char *SampleImages(int ID);
void DlpPunctaImage(struct Images* pImg1, CvPoint* ctr, int r1, int r2);
    // Prepare a puncta image for the DLP, save it in pImg1->DlpImage
unsigned char * DlpPunctaImageSimple(CvPoint* ctr, int radius);
void calibrateDLP(struct Images* pImg1, struct DlpCalib *pDlp1, char *fname);
void verifyDlpCalib(struct Images* pImg1, struct DlpCalib *pDlp1);
void update_dlp_calib_file(struct DlpCalib *pDlp1);
void Fire_DLP_at_worm(struct Images* pImg1, struct DlpCalib *pDlp1);
int Fire_DLP_by_script(struct Images* pImg1, struct DlpCalib *pDlp1, struct Script *pScript, int *dlpdoneflag, int *wholeillumflag);
void Blank_DLP(struct DlpCalib *pDlp1);
void displayDlpSample(long alpid, int imgid = 1, int pauseflag = 1);
#define T2DLP_HAPPY 0
#define T2DLP_SAD -1

/* Hard Code in the dimensions. */
* We are Using ALPB_DMDTYPE_XGA_055X
* Which is XGA .55 Type X
* 1024 by 768 mirrors
*/
#define NSIZEX 1024
#define NSIZEY 768

#endif /* TALK2DLP_H */
```

DlpAccessoryFuncs.cpp

/*
 * DlpAccessoryFuncs.cpp
 * Anthony Fouad
 * Fang-Yen Group, 6/2014
 *
 * Defines the functions for controlling the DLP. Derived heavily from Leifer & Fang-Yen, 2010.
 * It uses the ALPbasic API
 */
/* 
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include "../StageBoost/3rdPartyLibs/alpbasic.h"
#include "Images.h"
#include "boost\thread.hpp"
#include "AnthonysColors.h"
#include "AnthonysTimer.h"
#include "AnthonysCalculations.h"
#include "AnthonysCamFuncs.h"
#include "PVcamAccessoryFuncs.h"
#include "DlpAccessoryFuncs.h"
/

* Turn the DLP on
*/

long T2DLP_on(){
    long ret;                  // this is for what the
ALPbDevAlloc will return
    unsigned long serial;    // This will be the serial number
    for
        long alpid;          // Create an ID for the DMD.
            //Allocate memory for the DMD. Value of ret=0 means ALPB_SUCCESS
            ret = AlpbDevAlloc( 0, &alpid );

            if (0>ret) { // If that didn't work
                AlpbDevFree((ALPB_HDEVICE) alpid );     // close device
                return (long) T2DLP_SAD;              //throw an
                driver
            }
            // Query serial number
            ret = AlpbDevInquire( alpid, ALPB_DEV_SERIAL, &serial );
            if (0>ret){
                printf("Error: AlpbDevInquire (Serial number)\n");
                AlpbDevFree((ALPB_HDEVICE) alpid ); // close device driver
                return (long) T2DLP_SAD;
            }
            //printf("\tThe allocated ALP has the serial number %d\n", serial
            );
            //printf("\tAlpid is %f\n",alpid);
            T2DLP_clear(alpid);
return alpid;
}

/*
 * Clear the DLP mirrors
 */
int T2DLP_clear(long myDLP){
    AlpbDevReset((ALPB_HDEVICE) myDLP, ALPB_RESET_GLOBAL,0);
    return 0;
}

/*
 * Disconnect from the DLP
 */
int T2DLP_off(long alpid){
    printf("Closing the DLP device driver...\n");
    AlpbDevReset((ALPB_HDEVICE) alpid, ALPB_RESET_GLOBAL,0);
    AlpbDevFree((ALPB_HDEVICE) alpid); // close device driver
    return T2DLP_HAPPY;
}

/*
 * Send an image to the DLP. Equivalent to the script commands:
 * 1. Load
 * 2. Reset
 */
int T2DLP_SendFrame(unsigned char * image, long alpid){
    long ret;
    ret = AlpbDevLoadRows( alpid, image, 0, NSIZEY-1 );
    if (0>ret){
        printf("DLP: Error sending image to DLP.\n");
    }

    // Reset DMD mirrors
    ret = AlpbDevReset( alpid, ALPB_RESET_GLOBAL, 0 );

    return (int) ret;
}

/*
 * Convert an IplImage to an unsigned char image for use with the DLP
 */
void Ipl2uchar(IplImage* imgin, unsigned char * imgout){
    unsigned long x, y;

    // Get the pixels from the input IplImage and put them in the uchar image
    for (y=0; y<NSIZEY; y++){
        for (x=0; x<NSIZEX; x++){
            imgout[y*NSIZEX + x] = CV_IMAGE_ELEM(imgin,uchar,y,x);
        }
    }
}

/*
 * Prepare a sample image to show to the DLP
 */

unsigned char *SampleImages(int ID)
{
    unsigned long x, y;
    unsigned char *image;
    image = (unsigned char*) malloc(NSIZEX*NSIZEY);
    int idx;

    if (ID==0) {
        // Picture 0: horizontal bars (appears not to work
        11/14/2014)
        for (y=0; y<NSIZEY; y++)
            FillMemory(
                image + 0*NSIZEY*NSIZEX + y*NSIZEX, // row
                NSIZEX,       // row size in bytes
                (y&256)? 0 : 128 ); // image data: either 0 or 128
    } else if(ID==1){
        // Picture 1: checkered pattern
        for (y=0; y<NSIZEY; y++){
            for (x=0; x<NSIZEX; x++){
                idx = (y)*NSIZEX + x;
                image[idx] = (unsigned char) ((x^y)& 32)? 0 :
                128;
            }
        }
    } else if(ID==2){
        // Picture 2: LARGE checkered pattern
        for (y=0; y<NSIZEY; y++)
            253
for (x=0; x<NSIZEX; x++){
    idx = (y)*NSIZEX + x;
    image[idx] = (unsigned char) (y%32<=16)? 0 : 128;
}
}
}

else if (ID==3){
    // Picture 3: whole plane illumination
    for (y=0; y<NSIZEY; y++){
        for (x=0; x<NSIZEX; x++){
            idx = (y)*NSIZEX + x;
            image[idx] = (unsigned char) 128;
        }
    }
}

else if (ID==4){
    // Picture 4: no illumination
    for (y=0; y<NSIZEY; y++){
        for (x=0; x<NSIZEX; x++){
            idx = (y)*NSIZEX + x;
            image[idx] = (unsigned char) 0;
        }
    }
}

return image;

/*
 * Prepare an illumination image of a circle around some puncta to send to the DLP. Provide the center and radius of the circle.
 * The resulting image is fully compatible with T2DLP_sendframe
 */

void DlpPunctaImage(struct Images* pImg1, CvPoint* ctr, int r1, int r2){

    // Generate an IplImage rendition of the DMD pattern where values are 0 (off) or 128 (on)
    cvZero(pImg1->ProtoDlpImage);
    cvDrawCircle(pImg1->ProtoDlpImage,*ctr,r1,ww,-1,8,0); // thickness = -1 : the circle will be filled
    mulByScalar(pImg1->ProtoDlpImage,pImg1->ProtoDlpImage,pImg1->ProtoDlpScratchImage,128); // Make all ones into 128's

254
// Draw the inner disk -- things NOT to stimulate, if any
cvDrawCircle(pImg1->ProtoDlpImage,*ctr,r2,kk,-1,8,0);

// Flip the image if requested
if(pImg1->invertregionflag==1){
    subFromScalar(pImg1->ProtoDlpImage,pImg1->ProtoDlpImage,128);
}

// Convert the IplImage to an unsigned char image by filling in
image
Ipl2uchar(pImg1->ProtoDlpImage,pImg1->DlpImg);

// pImg1->DlpImage is the unsigned character image, which is
fully compatible with T2DLP_SendFrame

// Make sure the user didn't specify the radii backwards
if(r1<r2){printf("WARNING: r1<r2 -- you probably are not getting
any illumination!\n");}

void DlpPunctaAndStreakImage(struct Images* pImg1, CvPoint* ctr, int r1){

    // Generate an IplImage rendition of the DMD pattern where values
are 0 (off) or 128 (on)
cvZero(pImg1->ProtoDlpImage);

    // Draw the circle
cvDrawCircle(pImg1->ProtoDlpImage,*ctr,r1,ww,-1,8,0); // thickness = -1 : the circle will be filled

    // Draw the manual ROI streak boundaries
cvDrawPlot(pImg1->ProtoDlpImage,pImg1->SliceFire,ww,1);

    // Fill in the streak
cvFindContours(pImg1->ProtoDlpImage,pImg1->contourMem,&pImg1->Contours,sizeof(CvContour)*50,CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE);
cvDrawContours( pImg1->ProtoDlpImage, pImg1->Contours, ww, ww, 1, CV_FILLED);

    // Convert 1's into 128's
mulByScalar(pImg1->ProtoDlpImage,pImg1->ProtoDlpImage,pImg1->ProtoDlpScratchImage,128); // Make all ones into 128's
void DlpWingImage(struct Images* pImg1, CvPoint* w1, CvPoint*w2 , int r1){

    //Make sure the streak (wing) image is not too big
    if(r1>100){printf("r1 (streak) is too thick. Aborting uchar...\n"); return;}

    // Generate an IplImage rendition of the DMD pattern where values
    // are 0 (off) or 128 (on)
    cvZero(pImg1->ProtoDlpImage);
    CvPoint pt = cvPoint(0,0);

    // Draw streak of illumination whose thickness is the radial
distance times 2 (would be the diameter if this were a puncta image)
cvLine(pImg1->ProtoDlpImage,*w1,*w2,ww,2*r1,8);

    // Make the image 0 (off) or 128 (on)
mulByScalar(pImg1->ProtoDlpImage,pImg1->ProtoDlpImage,pImg1->
    >ProtoDlpScratchImage,128); // Make all ones into 128's

    // Convert the IplImage to an unsigned char image by filling in
    image
    Ipl2uchar(pImg1->ProtoDlpImage,pImg1->DlpImg);

    // pImg1->DlpImage is the unsigned character image, which is
    fully compatible with T2DLP_SendFrame
}
/* * Prepare an illumination image of TWO streak perpendicular to the * worm pharynx. * The resulting image is fully compatible with T2DLP_sendframe */

void Dlp2WingImage(struct Images* pImg1, CvPoint* w1, CvPoint*w2, CvPoint *w3, CvPoint* w4, int r1){
    // If this is a SRARP mode (e.g. 51) check if SRARP has been requested
    if(pImg1->tracktypeflag>=51 && pImg1->tracktypeflag<=59 && r1 >=50){
        cvSet(pImg1->ProtoDlpImage,ww);
    } else {
        // Make sure the streak (wing) image is not too big
        if(r1>100){printf("r1 (streak) is too thick. Aborting uchar...\n"); return;}
        // Design a contour containing all four wing points, to illuminate the box
        cvClearSeq(pImg1->Contours);
        cvSeqPush(pImg1->Contours,w1);
        cvSeqPush(pImg1->Contours,w2);
        cvSeqPush(pImg1->Contours,w4);
        cvSeqPush(pImg1->Contours,w3);

        // Generate an IplImage rendition of the DMD pattern where values are 0 (off) or 128 (on)
        cvZero(pImg1->ProtoDlpImage);
        CvPoint pt = cvPoint(0,0);

        // Draw the 4-point box on the DLP image
        cvDrawContours(pImg1->ProtoDlpImage,pImg1->Contours,ww,ww,1,-1,8);

        ///// Draw first streak of illumination whose thickness is the radial distance times 2 (would be the diameter if this were a puncta image)
        //cvLine(pImg1->ProtoDlpImage,*w1,*w2,ww,2*r1,8);
        ///// Draw second streak

    } 257
/cvLine(pImg1->ProtoDlpImage,*w3,*w4,ww,2*r1,8);
   // Make the image 0 (off) or 128 (on)
   mulByScalar(pImg1->ProtoDlpImage,pImg1->ProtoDlpImage,pImg1->ProtoDlpScratchImage,128);  // Make all ones into 128's

   // Convert the IplImage to an unsigned char image by filling in image
   Ipl2uchar(pImg1->ProtoDlpImage,pImg1->DlpImg);
   // pImg1->DlpImage is the unsigned character image, which is fully compatible with T2DLP_SendFrame

}*/

void DlpWingAndPDEImage(struct Images* pImg1, CvPoint* ctr , CvPoint* w1, CvPoint*w2 , int r1, int r2, int r3){

   //Make sure the streak (wing) image is not too big
   if(r1>40){printf("r1 (streak) is too thick. Aborting uchar...\n"); return;}

   // Generate an IplImage rendition of the DMD pattern where values are 0 (off) or 128 (on)
   cvZero(pImg1->ProtoDlpImage);
   CvPoint pt = cvPoint(0,0);

   // Draw the hollow circle
   cvDrawCircle(pImg1->ProtoDlpImage,*ctr,r2,ww,-1,8,0);  // thickness = -1 : the circle will be filled
   cvDrawCircle(pImg1->ProtoDlpImage,*ctr,r3,kk,-1,8,0);

   // Draw streak of illumination whose thickness is the radial distance times 2 (would be the diameter if this were a puncta image)
   cvLine(pImg1->ProtoDlpImage,*w1,*w2,ww,2*r1,8);

   // Make the image 0 (off) or 128 (on)
mulByScalar(pImg1->ProtoDlpImage,pImg1->ProtoDlpImage,pImg1->ProtoDlpScratchImage,128); // Make all ones into 128's

// Convert the IplImage to an unsigned char image by filling in image
Ipl2uchar(pImg1->ProtoDlpImage,pImg1->DlpImg);

// pImg1->DlpImage is the unsigned character image, which is fully compatible with T2DLP_SendFrame

// Make sure the user didn't specify the radii backwards
if(r2<r3){printf("WARNING: r2<r3 -- you probably are not getting any illumination!
");}

}

/* Prepare an illumination image to illuminate a fraction of the worm body. Illuminate a thick line between the two points *
* Also capable of SRARP illumination (#32): radii larger than 50 are assumed to mean "illuminate the whole worm"
*/

void DlpBodyIllumImage(struct Images* pImg1, CvSeq* seq, int r1){

    // Generate an IplImage rendition of the DMD pattern where values are 0 (off) or 128 (on)
    cvZero(pImg1->ProtoDlpImage);

    if(r1<50){
        if(seq->total>=3){cvDrawPlotFlat(pImg1->ProtoDlpImage,seq,r1,pImg1->tracktypeflag!=34,pImg1->sideflag);}
        else
            {printf("FAILED TO UPDATE DLP - region is too small\n");}
    }
    else{
        cvSet(pImg1->ProtoDlpImage,ww);
    }

    // Flip the image if requested
    if(pImg1->invertregionflag==1){
        subFromScalar(pImg1->ProtoDlpImage,pImg1->ProtoDlpImage,128);
    }

    // pImg1->DlpImage is the unsigned character image, which is fully compatible with T2DLP_SendFrame
Ipl2uchar(pImg1->ProtoDlpImage,pImg1->DlpImg);
}

/*
 * Simple version of DlpPunctaImage for testing purposes
 */
unsigned char * DlpPunctaImageSimple(CvPoint* ctr, int radius){
    // Setup some variables
    unsigned char *imgout;
    imgout = (unsigned char*) malloc( NSIZEX*NSIZEY);
    IplImage *imgin = cvCreateImage(cvSize(NSIZEX,NSIZEY),IPL_DEPTH_8U,1);
    IplImage *scratch=cvCreateImage(cvSize(NSIZEX,NSIZEY),IPL_DEPTH_8U,1);
    // Generate an IplImage rendition of the DMD pattern where values are 0 (off) or 128 (on)
    cvZero(imgin);
    cvDrawCircle(imgin,*ctr,radius,ww,-1,8,0);  // thickness = -1 : the circle will be filled
    mulByScalar(imgin,imgin,scratch,128);  // Make all ones into 128's

    // Convert the IplImage to an unsigned char image by filling in image
    //Ipl2uchar(imgin,imgout);
    printf("TURNED OFF SEE FUNCTION\n");

    // imgout is the unsigned character image, which is fully compatible with T2DLP_SendFrame
    return imgout;
}

/*
 * Calibrate the DLP:
 * 1. Turn on each DLP pixel, one-by-one, including a small radius around each one
 * 2. Figure out which camera pixel, if any, corresponds to each DLP pixel (which cam pixel is bright)
 * 3. Generate a map indicating which DLP pixel to turn on for any given command in image pixel coordinates
 * Calibration requires that one of the following is placed on the stage:
 */
1. A uniform fluorescein slide that is very very thin (preferred), or
2. a flat, highly scattering slide (remove green filter in the color splitter to image some blue light)

```c
void calibrateDLP(struct Images* pImg1, struct DlpCalib *pDlp1, char *fname){
    /* PART 1: Get the initial correlation data between DLP coordinates and image coordinates */
    // Setup some variables
    int x,y;
    CvPoint *max_loc = (CvPoint*) malloc(sizeof(CvPoint));
        *max_loc = cvPoint(0,0);
    CvPoint ctr = cvPoint(0,0);
    CvPoint pt1,pt2;
        pt1 = cvPoint(0,0);
        pt2 = cvPoint(FLUO_IMAGE_GRY->width,FLUO_IMAGE_GRY->height/2);
    CvScalar avg_val;
    double max_val = 0;
    double min_val = 0;
    int savedanimageflag = 0;
    cv::Mat m = cvCreateMat(FLUO_IMAGE_CLR->height,FLUO_IMAGE_CLR->width,CV_16U); // temporary Mat used for PVCAM pixel manipulations
    cvNamedWindow("Fluo feed",0);
    FILE *file;
    file = fopen("YourDirectory/StageBoost/Parameters/DlpCalibrationRaw.txt","w");
        // If we are use ICCamera, open it as a cvCapture
    CvCapture* cv_cap_fl = (CvCapture*) malloc(sizeof(CvCapture*));
        if(pImg1->PVCAM_ACQUIRE_MODE<0){cv_cap_fl = cvCreateCameraCapture(pImg1->Cam2ID);}
    // Pick a threshold above which maxima must lie to not be interpreted as 0
        double thresh = 0;
        if(pImg1->PVCAM_ACQUIRE_MODE>=0) { thresh = 25;}
        else{thresh = 4.5;}
        // Main calibration loop - create a map of pixel in image to pixel on DMD
```
for(y=0;y<767;y+=50){
for(x=0;x<1023;x+=50){
// Prepare a DLP puncta image around each point
ctr = cvPoint(x,y);
DlpPunctaImage(pImg1,&ctr,8,0);
// Display it on the DLP
T2DLP_SendFrame(pImg1->DlpImg, pDlp1->alpid);
// Wait for a while to force the PVCAM to have
an up to date frame.
boost::this_thread::sleep(boost::posix_time::milliseconds(3*pImg1
->exp_time));
// Record a fluorescence (or pseudofluorescence) frame on the PVCAM or ICcam
if(pImg1->PVCAM_ACQUIRE_MODE>=0){PVCAM_acquire(
pImg1->hCam, pImg1->frame, pImg1->region, pImg1->size, pImg1>PVCAM_ACQUIRE_MODE);
frame2Ipl(pImg1>frame,FLUO_IMAGE_GRY,&m,&pImg1->region);
cvFlip(FLUO_IMAGE_GRY,FLUO_IMAGE_GRY,0);
cvFlip(FLUO_IMAGE_GRY,FLUO_IMAGE_GRY,1);
}
else{
IplImage* img_color_fl =
cvQueryFrame(cv_cap_fl);
cvFlip(img_color_fl,img_color_fl);
cvCopy(img_color_fl,FLUO_IMAGE_CLR);
cvRotateImage(FLUO_IMAGE_CLR,FLUO_IMAGE_CLR,270);
cvCvtColor(FLUO_IMAGE_CLR,FLUO_IMAGE_GRY,CV_RGB2GRAY);
}
//normalize the image
normalize2uint16(FLUO_IMAGE_GRY,FLUO_IMAGE_GRY,pImg1>FluoImgGrayTemp,pImg1->mask);
// Paint the GFP channel with a meaningless
value so that only one channel is in use (GFP) during calibration. The
Fluorescein
// we use appears to have some red components.
avg_val = cvAvg(FLUO_IMAGE_GRY);
/*cvDrawRect(FLUO_IMAGE_GRY,
262


cvPoint(0, 0), cvPoint(FLUO_IMAGE_GRY->width, FLUO_IMAGE_GRY->height * 6/10), avg_val, CV_FILLED); */

// Find out where the puncta is centered on the PVCAM image. Black out the GFP channel first, use only the mCherry for correlation

// cvDrawRect(FLUO_IMAGE_GRY, pt1, pt2, kk, CV_FILLED);

cvMinMaxLoc(FLUO_IMAGE_GRY, &min_val, &max_val, NULL, max_loc, pImg1->mask);

cvCvtColor(FLUO_IMAGE_GRY, FLUO_IMAGE_CLR, CV_GRAY2RGB);

// If the maximum is very small, there probably isn't any illumination at this point on the display.
if (max_val < (avg_val.val[0] * thresh)) {
  *max_loc = cvPoint(0, 0);
}

// Print the correlation to file as:
// Dlp X    |    Dlp Y    |    Fluo Img X    |    Fluo Img Y
printf("DLP CALIBRATION:
%dt%tt%tt%ttt%ttt%tt==%ttt%tt==%nt", x, y, max_loc->x, max_loc->y, (int) max_val, (int) avg_val.val[0]);
fprintf(file,"%tt%tt%tt%tt%tt%tt%tt%tt
", x, y, max_loc->x, max_loc->y);

// Show the image and pause for a moment
cvDrawCross(FLUO_IMAGE_CLR, max_loc, rr, 2, 20);
cvShowImage("Fluo feed", FLUO_IMAGE_CLR); int c = cvWaitKey(20);
if (c == 27) {
  cvDestroyWindow("Fluo feed");
  cvReleaseCapture(&cv_cap_fl);
  Blank_DLP(pImg1, pDlp1);
  return;
}
}

cvDestroyAllWindows();
fclose(file);
/* Get the new calibration coefficients from matlab */

char calibdir[300] = "\"YourCalibrationProgramDirectory/C2_calibration_surface.exe\"";
system(calibdir);
double tempdouble = 0;

file = fopen(fname,"r");

if (file > 0){

    // First six are the p values
    fscanf(file,"%lf\t",&tempdouble);
    pDlp1->p00x = tempdouble;
    fscanf(file,"%lf\t",&tempdouble);
    pDlp1->p10x = tempdouble;
    fscanf(file,"%lf\t",&tempdouble);
    pDlp1->p01x = tempdouble;
    fscanf(file,"%lf\t",&tempdouble);
    pDlp1->p00y = tempdouble;
    fscanf(file,"%lf\t",&tempdouble);
    pDlp1->p10y = tempdouble;
    fscanf(file,"%lf\t",&tempdouble);
    pDlp1->p01y = tempdouble;

    // Last two are the Xratio and Yratio of the PVCAM:
DLP pixel sizes
    fscanf(file,"%lf\t",&tempdouble);
    pDlp1->Xratio = tempdouble;
    fscanf(file,"%lf\t",&tempdouble);
    pDlp1->Yratio = tempdouble;
}
fclose(file);

// Close down the calibration
if(PVCAM_ACQUIRE_MODE_DEF<0){cvReleaseCapture(&cv_cap_fl);
}

Blank_DLP(pImg1,pDlp1);

/*
 * Overwrite the disk DLP calibration file, in case the user has
 * manually fine tuned it
 */
void update_dlp_calib_file(struct DlpCalib *pDlp1){

264
printf("Updating DLP calibration data...
");
FILE* file = fopen(pDlp1->CalibFile,"w");

if (file > 0){
    printf("%f\t%f\t%f\t%f\t%f\t%f\n",pDlp1->p00x,pDlp1->p10x,pDlp1->p01x,pDlp1->p00y,pDlp1->p10y,pDlp1->p01y,pDlp1->Xratio,pDlp1->Yratio);
    fprintf(file,"%f\t%f\t%f\t%f\t%f\t%f\n",pDlp1->p00x,pDlp1->p10x,pDlp1->p01x,pDlp1->p00y,pDlp1->p10y,pDlp1->p01y,pDlp1->Xratio,pDlp1->Yratio);
}
fclose(file);

/*
 * Fire the DLP at the worm! (no script, just fire continuously)
 */

void translate_slice_to_fire(CvSeq* InputSlice, CvSeq* OutputSlice, int StartPtIdx, int FocusPtIdx, double p00x, double p10x, double p01x, double p00y, double p10y, double p01y){

cvClearSeq(OutputSlice);
int Xresult, Yresult, x, y;
CvPoint data = cvPoint(0,0);

for (int i=StartPtIdx; i<FocusPtIdx; i++){
    // Extract each point as a CvPoint*
    CvPoint* pt = (CvPoint*) cvGetSeqElem(InputSlice,i);

    // Translate the point from PVCAM pixels to DLP pixels
    x = pt->x;
y = pt->y;
    Xresult = p00x + p10x * x + p01x * y;
    Yresult = p00y + p10y * x + p01y * y;
    data = cvPoint(Xresult,Yresult);

    // Feed translated point to the to-illum sequence
    cvSeqPush(OutputSlice,&data);
}
}

void Fire_DLP_at_worm(struct Images* pImg1, struct DlpCalib *pDlp1){
    /* Figure out the target position. X and Y are flipped between the DLP and the image for PVCAM.*/
}
Xresult = ptotal(1) + ptotal(2)*Xtarget + ptotal(3)*Ytarget;
Yresult = ptotal(4) + ptotal(5)*Xtarget + ptotal(6)*Ytarget;
*/

//Translate puncta to DLP coordinates

// Calculate the puncta's location on the DLP.
int x = pDlp1->target->x;
int y = pDlp1->target->y;
double p00x = pDlp1->p00x;
double p10x = pDlp1->p10x;
double p01x = pDlp1->p01x;
double p00y = pDlp1->p00y;
double p10y = pDlp1->p10y;
double p01y = pDlp1->p01y;
int Xresult = p00x + p10x * x + p01x * y;
int Yresult = p00y + p10y * x + p01y * y;

// Calculate the wing locations on the DLP. Wings are only used for some tracking modes (e.g. 20s).
x = pDlp1->target_wing1->x;
y = pDlp1->target_wing1->y;
int Xwing1 = p00x + p10x * x + p01x * y;
int Ywing1 = p00y + p10y * x + p01y * y;

x = pDlp1->target_wing2->x;
y = pDlp1->target_wing2->y;
int Xwing2 = p00x + p10x * x + p01x * y;
int Ywing2 = p00y + p10y * x + p01y * y;

x = pDlp1->target_wing3->x;
y = pDlp1->target_wing3->y;
int Xwing3 = p00x + p10x * x + p01x * y;
int Ywing3 = p00y + p10y * x + p01y * y;

x = pDlp1->target_wing4->x;
y = pDlp1->target_wing4->y;
int Xwing4 = p00x + p10x * x + p01x * y;
int Ywing4 = p00y + p10y * x + p01y * y;

// Declare the cvPoints to hold target info on DLP
CvPoint ctr = cvPoint(0,0);
CvPoint w1  = cvPoint(0,0);
CvPoint w2  = cvPoint(0,0);
CvPoint w3 = cvPoint(0,0);
CvPoint w4 = cvPoint(0,0);

// Check that the point is within range of the DLP.
if(Xresult<0 || Xresult > NSIZEX || Yresult < 0 || Yresult > NSIZEY){
    printf("Target out of range @ (%d,%d) on image @ (%d,%d) on DLP\n",pImg1->max_loc->x,pImg1->max_loc->y,ctr.x,ctr.y);
    if(Xresult<0) {Xresult=0;}
    if(Xresult>NSIZEX) {Xresult=NSIZEX;}
    if(Yresult<0) {Yresult=0;}
    if(Yresult>NSIZEY) {Yresult=NSIZEY;}
}

// Assemble the target point
ctr = cvPoint(Xresult,Yresult);
w1 = cvPoint(Xwing1,Ywing1);
w2 = cvPoint(Xwing2,Ywing2);
w3 = cvPoint(Xwing3,Ywing3);
w4 = cvPoint(Xwing4,Ywing4);

// Translate sequences to DLP coordinates

// Prepare a sequence of points to illuminate regions of the worm body at 10x (e.g. tracking modes 12, or 30s). The region is from the head to the focus point
CvPoint data = cvPoint(0,0);

if( ((pImg1->tracktypeflag>=31 && pImg1->tracktypeflag<=39)) && pImg1->Contours->total>2 && pImg1->Centerline->total>50 && pDlp1->r1<50){
    translate_slice_to_fire(pImg1->Centerline, pImg1->SliceFire, pImg1->StartPtIdx, pImg1->FocusPtIdx, p00x, p10x, p01x, p00y, p10y, p01y);
}
else if(pImg1->tracktypeflag>=31 && pImg1->tracktypeflag<=39 && pDlp1->r1<50){
    data = cvPoint(Xresult,Yresult);
cvSeqPush(pImg1->ContoursTemp,pImg1->max_loc);
cvSeqPush(pImg1->SliceFire,&data);
    printf("Failed to extract illumination region\n");
}
else if(pImg1->tracktypeflag==12 && pImg1->manualBound->total>2){
    translate_slice_to_fire(pImg1->manualBoundClosed, pImg1->SliceFire, 0, pImg1->manualBoundClosed->total, p00x, p10x, p01x, p00y, p10y, p01y);
}
else if (pImg1->tracktypeflag==12){
    cvClearSeq(pImg1->SliceFire);
}

// Prepare the uchar image to send to DLP
if (is_tracktype(pImg1->tracktypeflag,2,23,24))
    {DlpWingImage(pImg1, &w1, &w2, pDlp1->r1);} else if (is_tracktype(pImg1->tracktypeflag,5,25,27,28,51,52))
    {Dlp2WingImage(pImg1, &w1, &w2, &w3, &w4, pDlp1->r1);} else if (pImg1->tracktypeflag==26 )
    {DlpWingAndPDEImage(pImg1, &ctr, &w1, &w2, pDlp1->r1, pDlp1->r2, pDlp1->r3);} else if (pImg1->tracktypeflag>=31 && pImg1->tracktypeflag<=34)
    {DlpBodyIllumImage(pImg1,pImg1->SliceFire,pDlp1->r1);} else if (pImg1->tracktypeflag==12)
    {DlpPunctaAndStreakImage(pImg1, &ctr,pDlp1->r1);}
else
    {DlpPunctaImage(pImg1, &ctr, pDlp1->r1,pDlp1->r2);} else
    {printf("Failed to route DLP image design by track type\n");}

// Send the fire image to the DLP
T2DLP_SendFrame(pImg1->DlpImg,pDlp1->alpid);

/*
 * Fire the DLP at the worm! using the loaded script.
 */

int Fire_DLP_by_script(struct Images* pImg1, struct DlpCalib *pDlp1, struct Script *pScript, int* dlpdoneflag, int *wholeillumflag){

    // Find out which step of the script we are in, and which arguments it has.
    char this_command[100]; strcpy(this_command,pScript[pDlp1->dlscriptstep].command);
    double this_arg[6];
    for(int i=0; i<6;i++) {this_arg[i] = pScript[pDlp1->dlscriptstep].arg[i];}
    if(*dlpdoneflag!=1){

        // Find out if we have reached a repeat. If so exit this function now with code 0, which means NOT done.
        if(!strcmp(pScript[pDlp1->dlscriptstep].command, "REPEAT")){

268
pDlp1->dlprepeatcount++; printf("Hit Repeat. Repeat ct = %d\n",pDlp1->dlprepeatcount);

    // If we still need more repeat loops, reset the
    script step to the beginning
    if(pDlp1->dlprepeatcount < (double) this_arg[0]-pImg1->dt/1000){
        pDlp1->dlpscriptstep = 2;
        pDlp1->dlptime = 0;
        *dlpdoneflag = 0;
        return 0;
    }

    // If not, advance to the next step (probably an END)
    else {pDlp1->dlpscriptstep += 1;}
}

    // Find out if we need to advance to the next step (first
    argument of each DLP step is time). If so reset the DLP timer, update
    the command NOW. Not next loop.
    if(pDlp1->dlptime >= this_arg[0]-pImg1->dt/1000){

        // Update timer and scriptstep, although note that it
        won't actually be acted on until the next frame. This is so that the
        log faithfully indicates which frame the
        // DLP was turned on.
        pDlp1->dlptime = 0;
        pDlp1->dlpscriptstep += 1;
    }

    // Find out if we have reached the end. If so exit this
    function now with code 1, which means YES I AM DONE.
    if(!strcmp(pScript[pDlp1->dlpscriptstep].command , "END")){
        pDlp1->dlpscriptstep = 2;
        pDlp1->dlptime = 0;
        pDlp1->dlprepeatcount = 0;
        Blank_DLP(pImg1,pDlp1);
        *dlpdoneflag = 1;
        return 1;
    }

    // Find out if we are instructed to fire the DLP this time.
    If so, fire it.
    if(!strcmp(this_command,"DLP_ON")){

          // If we still need more repeat loops, reset the
        script step to the beginning
    if(pDlp1->dlprepeatcount < (double) this_arg[0]-pImg1->dt/1000){

        pDlp1->dlpscriptstep = 2;
        pDlp1->dlptime = 0;
        *dlpdoneflag = 0;
        return 0;
    }

    // If not, advance to the next step (probably an END)
    else {pDlp1->dlpscriptstep += 1;}
if(*wholeillumflag==0){displayDlpSample(pDlp1-alpid,3,0); }  // If targeting mode 0 is selected, just illuminate everything
else{
    pDlp1->r1 = this_arg[1];  // Update the radius of the spot size to fire.
    pDlp1->r2 = this_arg[2];  // Update the radius of the spot size to fire.
    pDlp1->r3 = this_arg[3];
    Fire_DLP_at_worm(pImg1, pDlp1);
    *dlpdoneflag = 0;  // DLP is not done until it finishes the script.
}
return 0;

// Find out if we are instructed to NOT fire the DLP at this time. If so, fire a blank image (nothing)
if(!strcmp(this_command,"DLP_OFF")){
    Blank_DLP(pImg1,pDlp1);
    *dlpdoneflag = 0;
    return 0;
}
else{
    *dlpdoneflag = 0;
    return 0;
}
return -99999;

/*
 * Instruct the DLP to fire a blank image, thus turning laser illumination off.
 */

void Blank_DLP(struct Images* pImg1, struct DlpCalib *pDlp1){
    int Yresult = 9999;  // The puncta will be WAY out of range, resulting in a black image
    int Xresult = 9999;
    CvPoint ctr = cvPoint(Xresult,Yresult);
    cvZero(pImg1->ProtoDlpImage);
    Ipl2uchar(pImg1->ProtoDlpImage,pImg1->DlpImg);
    T2DLP_SendFrame(pImg1->DlpImg,pDlp1->alpid);
cvDrawCross(FLUO_IMAGE_CLR,pImg1->max_loc,rr,1,10);
   // Draw red cross to indicate no DLP targeting in use.
   cvZero(pImg1->ProtoDlpImage);
}

/*
 * Display one of the sample images (see function above) on the DLP
 */

void displayDlpSample(long alpid, int imgid, int pauseflag){

   // get a sample image
   unsigned char *img = SampleImages(imgid);

   //Display it on the DLP until user says no more
   T2DLP_SendFrame(img,alpid);

   // Pause by defualt, unless pauseflag is specifically set to 0
   if(pauseflag == 1){
      printf("The image will display until you press any key\n");
      system("PAUSE");
   }

   free(img);
}

Images.h
/*
 * Images.h
 * Anthony Fouad
 * Fang-Yen Group, 6/2014
 *
 * Declares the main data structure for holding worm images, their segmentations, and related data.
 */
#ifndef IMAGES_H_
define IMAGES_H_

// opencv includes
#include "opencv\cv.h"
#include "opencv\highgui.h"

// PVcam includes
#include "master.h"
#include "pvcam.h"

struct Images* CreateImageStruct(uns16* circBufferInMemory, rgn_type region, rs_bool* pvstatus, int exp_time = 10, int PVCAM_ACQUIRE_MODE = 1, int width = 640, int height = 480);
struct DlpCalib* CreateDlpCalibStruct(char *fname);
struct Script CreateScriptStruct();
struct Script* import_script();
int print_script(struct Script* pScript);
int fprint_script(struct Script* pScript, FILE *fout);
int is_tracktype(int testtype,int n_args,...);

struct Images {
    // Color images
    IplImage *BG1; // Background image #1 (all 3 get averaged)
    IplImage *BG2; // Background image #2 (all 3 get averaged)
    IplImage *BG3; // Background image #3 (all 3 get averaged)
    IplImage *BGavgColor; // Background image average in color (really its the minimum of all 3)
    IplImage *MainImg; // Darkfield image from the darkfield camera
    IplImage *FluoImg; // Fluorescence image from the fluorescence camera
    IplImage *FluoImgSave; // A copy of FluoImg, blank, unlabeled and saved to disk.
    IplImage *FluoImg8bit; // Similar to above, but in 8 bit for writing to disk.
    IplImage *FluoImg8bitLarge; // Similar to above, but made larger for high quality writing.
    IplImage *FluoImgColor8bitWrite; // 8 bit fluo image for writing to disk
    IplImage *TempImgColor; // Temporary image (used as an intermediate in some calculations)
IplImage *StatusImg; // GUI image for showing the user the current status of the system
IplImage *StatusImgBlank; // Blank template of the GUI image. It gets filled in with up-to-date info 30 times per second.

// Gray images
IplImage *BGavg; // Similar to above, but in grayscale
IplImage *BG1gray; // Similar to above, but in grayscale
IplImage *BG2gray; // Similar to above, but in grayscale
IplImage *BG3gray; // Similar to above, but in grayscale
IplImage *MainImggray; // Similar to above, but in grayscale
IplImage *MainImgThresh; // Image of the thresholded worm
IplImage *MainImg8bitLarge; // Large darkfield image for high quality export to disk.
IplImage *SEimg; // Image of the smoothed worm
IplImage *TempImage; // Similar to above, but in grayscale
IplImage *FluoImgGray; // Similar to above, but in grayscale
IplImage *FluoImgGrayValid; // Scratchwork image for GFP-based validation
IplImage *FluoImgTemp; // Temporary image used for scratchwork in the pharynx tracking
IplImage *FluoImgTemp2; // Temporary image used for scratchwork in background subtraction
IplImage *FluoImgGray8bit; // 8 bit version of the gray fluo image
IplImage *FluoImgGray8bitWrite; // Version of the 8 bit image written to disk
IplImage *FluoImgGray8bitTemp; // Temporary image, but in 8bit for hist equalization.
IplImage *FluoImgColor8bitTemp; // Temporary image in color.
IplImage *FluoImgGray8bitLarge; // Gray version of the large fluo image for writing to disk rapidly
IplImage *FluoImgThresh; // Temporary image used to find the centroid of the pharynx in pharyngeal tracking.
IplImage *FluoImgThreshInv; // Inverse of the threshold image. Used in GFP-based validation.
IplImage *ProtoDlpImage; // An IPL image the size of the DLP (1024x768) which is later converted to the uchar type below
IplImage *ProtoDlpScratchImage; // Used for some scratch work on ProtoDlpImage.
unsigned char* DlpImg; // Image to be sent to the DLP
IplImage* DlpMapX; // Map correlating pixels on FluoImg to X pixels on the DLP, for use in DLP targeting.
IplImage* DlpMapY; // Map correlating pixels on FluoImg to Y pixels on the DLP, for use in DLP targeting.
IplImage* mask; // Mask showing the relevant area of the fluorescence image for intensity normalizing.

// Flag Vbls
int SubtrFlag; // Whether the background has been defined
int OlFlag; // Whether to overlay worm contours
int SegFlag; // Whether the segmentation has been defined
int AutoFlag; // Whether to automate the stage
int MaxFlag; // Whether to update the max location (tracking point) automatically. If 0, only the user specifies track.
int FluoFlag; // Whether to grab and show the fluorescence frame
int CamID; // The camera ID number for openCV
int Cam2ID; // The camera for fluorescence imaging (DEFUNCT: the PVCAM is now used for fluorescence imaging)
int RecordFlag; // Whether to record frames to disk
int RecordInterval; // Interval, in frames, at which to save a frame to disk
int ComplFlag; // Whether to take the image complement, i.e. if using transmission.
int targetFlag; // 0 to target using darkfield, 1 to target using fluorescence.
int DlpFlag; // Whether to aim the DLP at the worm or not.
int scriptFlag; // Whether to run the loaded targeting script.
int startExptFlag; // Whether to run the experiment & stage automation - only changed to 1 after the user picks a focus point
int startExptFlagSignal; // A signal from the mouse callback function to set this flag to 1 on the next iteration of Cam()
int startAutoFlag; // Whether to run the stage automation. It's best to have a robust tracking before starting the experiment.
int objective; // Records which objective is in use: 10, 20 or 40 (x).
int LostTrackFlag; // Throws 1 if we have LOST track of the puncta (i.e. it went out of the field of view, and therefore we need to search a larger area for it.)
int tracktypeflag; // Which type of tracking to use -- simple puncta is 1, pharyngeal tracking is 2
int BulbSelectFlag; // If using pharyngeal tracking, this specifies whether to track bulb 1 or 2.
int endLoopFlag; // One if the user has DOUBLE clicked on the middle mouse button, meaning exit the camera loop.
int writeImgFlag; // Save fluorescent frames to disk and PNG IMAGES for highest quality. Used for validation. Overwrites existing files
int squareFlag; // Whether to multiply the image by itself, increasing contrast.
bool joystickflag; // Whether to use "virtual joystick" mode for the stage (1), or "go relative" mode. See shortcutkeys txt file for explanation.
int sideflag; // Which side to illuminate (0=both, 1=one side, 2=other side)
int invertregionflag; // Whether to invert the targeting region (may only apply to body illumination)
int drawstreakflag; // Whether to enter the manual streak drawing mode

//PVcam Vbls
char cam_name[CAM_NAME_LEN]; // camera name
int16 hCam; // Camera handle
uns16 *frame; // Pointer to image data
rgn_type region; // Image region and binning
uns32 size; // Total size of the 1D array storing a PVcam frame
int exp_time; // The exposure time in ms of the PVcam

275
int PVCAM_ACQUIRE_MODE; // What type of PVCAM interface to use. 0 = standard frame grabbing. 1 = circular buffer overwrite. 2 = no overwrite

double FrameRateFixed; // Hold the framerate of the camera constant at a lower bound (approx 50hz) to simplify postprocessing

// Thresholding and Head-Tail segmentation Parameters
int thresh;
int SE;
int fluoThresh;

// Head-Tail Segmentation parameters (for getting curvature as moving around the worm bound, head and tail are local curvature maxima)
// And other assorder CvPoint* 's
CvPoint* aheadpt,
    *behindpt,
    *tail,
    *head,
    *taillast,
    *headlast,
    *max_loc,
    *max_loc_old,
    *max_loc_gfp,
    *img_ctr,
    *fluo_img_ctr,
    *Ai,
    *Bi,
    *midPt;
CvPoint2D32f fvecn; // fixed vector
CvPoint2D32f tvecn; // moving vector, i.e. as you slide around the worm
CvPoint2D32f *midP, *thisA, *thisB;
int aheadptr; // =0;
int behindptr; // =0;
int ptr; // =0;
int ker;
float thisphi, d, thisdpds, mostcurvy, secondmostcurvy;
float *phi1, *phi2, *d1, *d2, *dotprodptr;
int curvstep, tailindex, headindex, mostcurvyindex,
currentcurviness, secondmostcurvyindex, neartail, NumSteps, wingPct;
double idxA, idxB;
CvSeq* phi;
CvSeq* s;
CvSeq* dphi_ds;
// Pharynx segmentation
CvPoint *ph_ter, // Terminal bulb
    *ph_met, // Metacorpus
    *ph_ter_old, // Terminal bulb in the last frame
    *ph_met_old, // Metacorpus in the last frame
*max_wing1, // Point along a vector perpendicular to the worm direction, to allow compensation for puncta lag
*max_wing2, // Point along a vector perpendicular to the worm direction, to allow compensation for puncta lag
*max2_wing1, // Point along a vector perpendicular to the worm direction, to allow compensation for puncta lag (for dual neuron targeting)
*max2_wing2, // Point along a vector perpendicular to the worm direction, to allow compensation for puncta lag (for dual neuron targeting)
*gfp_loc; // Point where the GFP maxima is located

double min_val, max_val; // maxima / puncta tracking location
double dt_pred; // Forecast interval, in milliseconds, for the maxima tracking
double dt; // The time that the last frame took to execute.
double fps; // framerate, in frames per second, for use in forecast tracking.
double gfp_val; // Gets the max value of the GFP channel puncta (i.e. for validation)

// Centerline Segmentation parameters
CvPoint *FocusPt, *StartPt; // Point along the worm where the 40x objective will be moved to on each iteration.
int FocusPtIdx; // Its index (# of points from head on the worm centerline)
int StartPtIdx; // For illuminating an arbitrary region of the worm (not just from the head)
int FocusPtDist; // For absolute distance darkfield tracking -- Distance from the nose tip to the focus point
Here is the text from the image:

```c
int StartPtDist; // For absolute distance darkfield tracking -- Distance from the nose tip to the start point
int FocusPtStep;

// Centroid values
int xc;
int yc;
int M00;
int M01;
int M10;
CvMoments* Moments;
CvPoint* Centroid;

// Other Segmentation stuff
CvSeq *Contours, *ContoursTemp, *ContoursFluo, *ContoursSamp,
CvMemStorage* contourMem, *contourMemTemp, *contourMemSamp,
*centerlineMem, *rightBoundMem, *leftBoundMem, *tempMem, *SliceShowMem,
double mm_per_pix_x; // From the calibration step: how many millimeters per pixel
double mm_per_pix_y;
int clickEvent;

};

// DLP data
struct DlpCalib {

    // DLP calibration data : Contains the information about linearly translating a coordinate on the image to a coordinate on the DLP
    double p00x;
double p10x;
double p01x;
double p00y;
double p10y;
double p01y;
double Xratio;
double Yratio;
double Xshift;
double Yshift;

    // DLP id number
    long alpid;

    // DLP timer : determines when to run various script steps depending on how much time has elapsed
```
double dlptime;  // Total time for this step of the experiment
double totaldlptime;  // Total time for this experiment
double totalrecordingtime;  // total time this cam loop has been running

// Script step: which step of the script we are on. Ranges between 2 and N, where N is "END", and the first two entries [0,1] are setup steps.
int dlpscriptstep;
int dlrepeatcount;

// Spot size to fire (updated from script arguments before every firing).
int r1;  // Spot size in DLP COORDINATES
int r2;  // Inner limit of the spot size in DLP coordinates. Used for making a hollow disk of illumination. Otherwise 0 for a filled disk.
int r3;  // For track type 26, r2 and r3 form the hollow disk. r1 is the width of the streak.

// Target point (center of the spot)
CvPoint* target;
CvPoint* target_wing1;
CvPoint* target_wing2;
CvPoint* target_wing3;
CvPoint* target_wing4;

// File name of the DLP calibration surface coefficients
char CalibFile[300];

};

// Inputs are only used for some debugging versions of the main program.
struct Inputs {
    char dir[300];
    char fname[300];
    int height, width;
};

// SCRIPT directions for the experiment. Contains setup info like PVCAM exp time and region, as well as the experimental plan
struct Script {
    // Information about this particular script entry
    char command[10];
double arg[6];
};
#endif

Images.cpp
/*
 * PVcamAccessoryFuncs.cpp
 * Anthony Fouad
 * Fang-Yen Group, 6/2014
 * Functions for reading from and controlling the PVcamera
 * (i.e. Photometrics Cascade 1K by Roper Scientific).
 * These are largely based on the example codes given in the
 * PVcam 2.9.11 documentation.
 */

// Standard include files
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>

// Boost includes
#include "boost\thread.hpp"
#include "boost\asio.hpp"

// Opencv 2.4 includes
#include "opencv\cv.h"
#include "opencv\highgui.h"

//PVcam includes
#include "master.h"
#include "pvcam.h"
#include "Common.h"
#include "PVcamAccessoryFuncs.h"

//Anthony's includes
#include "AnthonysTimer.h"

/*
 * Use a file from the disk (frame.txt) as a substitute for a valid
 * PVCAM
 */

void PVCAM_file(uns16 *frame, uns32 size){

}
printf("Attempting to load the sample PVCAM file 'frame.txt'...\n");

    // Load frame.txt
    FILE *file;
    char fname[200] = "{YourDirectory/frame - 1000 image.txt}";
    file = fopen(fname,"r");
    rewind(file);
    int tempint;

    for(int i = 0; i<size; i++){
        fscanf(file,"%d",&tempint);
        frame[i] = tempint;
    }
    fclose(file);

    printf("Successfully loaded the sample PVCAM file 'frame.txt'.\n");

} /*
 * Write a PVCAM frame to disk
 */
void frame2Txt(uns16 *frame, uns32 size){

    FILE *file;
    char fname[100] = "{YourDirectory/frame.txt}";
    file = fopen(fname,"w");

    for(int i = 0; i<size; i++){
        fprintf(file,"%d\n",frame[i]);
    }
    fclose(file);
}

/*
 * Convert a PVCAM frame (1D - uns16) to openCV (2D - IplImage)
 *
 * WARNING: FAILURE TO CORRECTLY SPECIFY frame_step will result in software crash.
 *
 * If the frame supplied has m->cols * m->rows elements, then
 frame_step = 1;
 */
If the frame supplied has $2 \cdot m \cdot \text{cols} \cdot m \cdot \text{rows}$ elements, then $\text{frame\_step} = 2$.

Some older versions of PVCAM (e.g. 2.9.11) confusingly doublecount each entry in frame.

For matrix assignment, see:

http://docs.opencv.org/doc/tutorials/core/how_to_scan_images/how_to_scan_images.html#performance-difference

```c
void frame2Ipl(uns16 *frame, IplImage* imgout, cv::Mat *m, rgn_type *region, int frame_step){

    // Determine the size of the frame
    int sizei  = (region->s2-region->s1) / region->sbin;
    int sizej  = (region->p2-region->p1) / region->pbin;
    int sizef  = sizei * sizej * frame_step;       // allocated size is usually double, so 1.5 is safe

    // Setup the counting variable
    int i = 0;
    int j = 0;

    // Setup the pointer to matrix data for fast assignment
    uns16 *p=m->ptr<uns16>(j);

    // Loop through the frame vector and reshape it to an image
    for(int ct=0; ct<sizef; ct+=frame_step){

        // Update the current pixel
        i+=1;
        if(i>sizei){i=0; j+=1; p=m->ptr<uns16>(j);}    
        if(j>sizej){printf("Breaking to prevent segfault in frame2Ipl\n"); break;}
        p[i] = frame[ct];
    }
    *imgout = (cv::Mat) *m;
}
```

* Force the spatial extents of the region to be evenly divisible by the binning factors.  
  Otherwise, images can get messed up.
void verify_region(rgn_type *region){

    // Verify serial direction
    int extent = region->s2-region->s1+1;
    int remainder = extent%region->sbin;
    region->s2=remainder;

    // Verify parallel direction
    extent = region->p2-region->p1+1;
    remainder = extent%region->pbin;
    region->p2=remainder;
}

/*
 * Acquire a frame from the PVCAM using the specified acquire method
 */
void PVCAM_acquire( int16 hCam, uns16* frame, rgn_type region, uns32 size, int PVCAM_ACQUIRE_MODE)
{
    switch(PVCAM_ACQUIRE_MODE){
        case 0:
            PVCAM_acquire_standard(hCam, frame, region, size, PVCAM_ACQUIRE_MODE);
            break;
        case 1:
            PVCAM_acquire_circ(hCam, frame, region, size, PVCAM_ACQUIRE_MODE);
            break;
        case 2:
            PVCAM_acquire_circ(hCam, frame, region, size, PVCAM_ACQUIRE_MODE);
            break;
        default:
            printf("------>INVALID ACQUISITION MODE SPECIFIED\n\n");
    }
}

/*
 * Setup the PVCAM for imaging. Select the appropriate initialization scheme.
 */
int PVCAM_start(uns16* circBufferInMemory, int16 *hCam, uns32 *size, char *cam_name, rgn_type *region, int exp_time, int PVCAM_ACQUIRE_MODE ){
int pvstatus = -1;
printf("Attempting to Connect to PVCAM...\n");
switch(PVCAM_ACQUIRE_MODE){
    case 0:
        pvstatus = PVCAM_setup_standard(hCam, size, cam_name, region, exp_time);
        break;
    case 1:
        pvstatus = PVCAM_setup_circ(circBufferInMemory,hCam, size, cam_name, region, exp_time,PVCAM_ACQUIRE_MODE);
        break;
    case 2:
        pvstatus = 0;
        break;
    default:
        printf("------INVALID ACQUISITION MODE SPECIFIED\\n\\n");
}
return pvstatus;

/*---------------------------------------------------------------------
----------------------------------
*/
/*
Setup modes for the PVCAM. Never invoked except by PVCAM_start
*/
/*---------------------------------------------------------------------
----------------------------------
*/

/*
* setup for standard frame-by-frame mode (type 0). Generally not in use.
*/

int PVCAM_setup_standard(int16 *hCam, uns32 *size, char *cam_name, rgn_type *region, int exp_time){
    // Try to close down the camera first in case some user error left it open
    pl_cam_close( *hCam );
    pl_pvcam_uninit();
    pl_exp_uninit_seq();

    // Initialize the PVCam Library and Open the First Camera
    pl_pvcam_init();
    pl_cam_get_name( *hCam, cam_name );
}
int pvstatus = pl_cam_open(cam_name, hCam, OPEN_EXCLUSIVE);

// Check for errors in the camera
int16 errorcode = pl_error_code();

// Figure out how much memory to allocate for the frame data (the size of the frame)
pl_exp_init_seq();
pl_exp_setup_seq( *hCam, 1, 1, region, TIMED_MODE, exp_time, size);

// Print out the connection status
if(pvstatus){printf("Successfully Connected to PVCAM via STANDARD buffering.\n\n");}
else{
    char_ptr msg = {"NO ERROR"};
    pl_error_message(errorcode,msg);
    printf("--------FAILED TO OPEN PVCAM. %s",msg);
}

return pvstatus;

/*
 * setup circular buffering WITH overwrite (type 1)
 */

int PVCAM_setup_circ(uns16* circBufferInMemory,int16 *hCam, uns32 *frame_size, char *cam_name, rgn_type *region, int exp_time, int PVCA

// Use Photometrics' demo function to start the PVCAM.
if (InitPVCAM(g_Camera0_Name))
    {
        if (!OpenCamera(g_Camera0_Name))
            {
                printf("FAILED TO OPEN PVCAM. SYSTEM WILL NOW EXIT\n\n");
                system("PAUSE");
                CloseCameraAndUninit(g_hCam);
                return 0;
            }
    }
else
    {
        CloseCameraAndUninit(g_hCam);
        return 0;
    }
// Use a modified version of Photometrics' demo code to start the continuous frame acquisition.
// setup the acquisition, FramesToAcquire specifies number of frames to acquire in this sequence
// exposureBytes will now hold the size of the whole buffer for all the images in the sequence
uns32 exposureBytes;

// setup the acquisition
if (PV_FAIL == pl_exp_setup_cont(g_hCam, 1, region, TIMED_MODE, exp_time, &exposureBytes, (int16) PVCAM_ACQUIRE_MODE))
{
    PrintErrorMessage(pl_error_code(), "pl_exp_setup_cont() error");
    CloseCameraAndUninit(g_hCam);
    return 0;
}
else
{
    printf("Acquisition setup successful\n");
    *frame_size = exposureBytes;
}

// allocate memory for circular buffer of frames, size of one frame is returned in
// exposureBytes by pl_exp_setup_cont(), assuming this is more than 8-bit camera we allocate
// array of uns16, therefore we divide exposureBytes by 2 and multiply by CIRC_BUFF_FRAMES so the circular buffer
// can hold 20 frames

/* MOVED TO THE MAIN FUNCTION StageBoost.cpp to facilitate destruction */

// Start the acquisition
// start the continuous acquisition, again tell this function size of buffer it has for the frames
// this is called only once after camera setup with pl_exp_setup_cont()
// this is a software trigger call in TIMED trigger mode, in hardware trigger modes (Strobe, Bulb etc)
// after this call camera waits for external trigger signals to start every exposures
if (PV_FAIL == pl_exp_start_cont(g_hCam, circBufferInMemory, CIRC_BUFF_FRAMES*exposureBytes/2))

{  
    PrintErrorMessage(pl_error_code(), "pl_exp_start_seq() error");
    CloseCameraAndUninit(g_hCam);
    delete[] circBufferInMemory;
    return 0;
}

else
    printf("Acquisition start successful\n");

/* ---------------------------------------------------------------------
     ----------------------------------
     /* Acquisition modes for the PVCAM. Never invoked except by PVCAM_acquire */
     ----------------------------------
     *---------------------------------------------------------------------

/* Acquire a frame by the standard method (type 0) */
void PVCAM_acquire_standard( int16 hCam, uns16* frame, rgn_type region, uns32 size, int PVCAM_ACQUIRE_MODE ){

    int16 status;
    uns32 not_needed;
    uns16* address = (uns16*) malloc(size*2);

    // Start the acquisition
    pl_exp_start_seq(hCam, address );

    // wait for data or error (Found by ADF to be an unnecessary waste of time -- data is retrieved by the time start_seq returns)
    while( pl_exp_check_status( hCam, &status, &not_needed ) &&
            (status != READOUT_COMPLETE && status != READOUT_FAILED) );

    // Check Error Codes
    if( status == READOUT_FAILED ) {
        printf( "Data collection error: %i\n", pl_error_code() );
        return;
    }

    /* Assemble the frame */
    for(int i = 0; i<size-1; i+=1)
        frame[i] = *((uns16*)address + i/sizeof(uns16));
void PVCAM_acquire_circ( int16 hCam, uns16* frame, rgn_type region, uns32 size, int PVCAM_ACQUIRE_MODE ){

    // Setup variables
    int16 status = 1;
    uns32 not_needed;

    // wait for data or error, ONCE, to ensure most recent frame
    while( pl_exp_check_cont_status( hCam, &status, &not_needed,&not_needed ) &&
          (status != READOUT_COMPLETE && status != READOUT_FAILED) );

    // Check for error codes
    if( status == READOUT_FAILED ) {
        printf("Data collection error: %i\n", pl_error_code());
        return;
    }

    // Acquire the latest frame (if using status 1, CIRC_OVERWRITE)
    if(PVCAM_ACQUIRE_MODE==1){
        void_ptr address;
        if( pl_exp_get_latest_frame( hCam,&address) ) {
            // Store the address of the valid frame located at
            // address in my "frame" variable
            for(int i = 0; i<size-1; i++){
                frame[i] = *((uns16*)address + i);
            }
            // DOESN'T WORK: frame = (uns16*) address;
        }
    }

    // -OR- acquire the oldest frame (if using status 2,
    // CIRC_NO_OVERWRITE)
    if(PVCAM_ACQUIRE_MODE==2){
        void_ptr address;
        pl_exp_unlock_oldest_frame(hCam);

    }
}
if ( pl_exp_get_oldest_frame(hCam,&address) ) {
    // Store the valid frame located at address in my
    // "frame" variable
    for(int i = 0; i<size-1; i++){
        frame[i] = *((uns16*)address +
        i/sizeof(uns16));
    }
    // Unlock the oldest frame for overwriting
    pl_exp_unlock_oldest_frame(hCam);
} else{
    printf("pl_error_code = %d\n",pl_error_code());
}
}

PVcamAccessoryFuncs.h
/*
 *       PVcamAccessoryFuncs.h
 *     Anthony Fouad
 *     Fang-Yen Group, 6/2014
 *     
 *     Functions for reading from and controlling the PVcamera
 *     (i.e. Photometrics Cascade 1K by Roper Scientific).
 *     These are largely based on the example codes given in the
 *     PVcam 2.9.11 documentation.
 */

#ifndef PVCAMACCESSORYFUNCS_H_
#define PVCAMACCESSORYFUNCS_H_

void frame2Ipl (uns16 *frame, IplImage* imgout, cv::Mat *m,
    rgn_type *region, int frame_step=1);
void frame2Txt (uns16 *frame, uns32 size);
void verify_region (rgn_type *region);
void PVCAM_file (uns16 *frame, uns32 size);
void PVCAM_acquire (int16 hCam, uns16* frame, rgn_type region ,
    uns32 size, int PVCAM_ACQUIRE_MODE);
int PVCAM_start (uns16* circBufferInMemory, int16 *hCam, uns32
    *size, char *cam_name, rgn_type *region, int exp_time,int
    PVCAM_ACQUIRE_MODE);
// setup type subfunctions
int PVCAM_setup_standard(int16 *hCam, uns32 *size, char *cam_name, rgn_type *region, int exp_time);
int PVCAM_setup_circ (uns16* circBufferInMemory, int16 *hCam, uns32 *size, char *cam_name, rgn_type *region, int exp_time, int PVCAM_ACQUIRE_MODE);

// acquisition type subfunctions
void PVCAM_acquire_standard ( int16 hCam, uns16* frame, rgn_type region, uns32 size, int PVCAM_ACQUIRE_MODE );
void PVCAM_acquire_circ ( int16 hCam, uns16* frame, rgn_type region, uns32 size, int PVCAM_ACQUIRE_MODE );

#endif

PVcamAccessoryFuncs.cpp
/*
   *      PVcamAccessoryFuncs.cpp
   *      Anthony Fouad
   *      Fang-Yen Group, 6/2014
   *
   *      Functions for reading from and controlling the PVcamera
   *      (i.e. Photometrics Cascade 1K by Roper Scientific).
   *      These are largely based on the example codes given in the
   *      PVcam 2.9.11 documentation.
   */

// Standard include files
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>

// Boost includes
#include "boost\thread.hpp"
#include "boost\asio.hpp"

// Opencv 2.4 includes
#include "opencv\cv.h"
#include "opencv\highgui.h"

// PVcam includes
#include "master.h"
#include "pvcam.h"
#include "Common.h"
#include "PVcamAccessoryFuncs.h"
//Anthony's includes
#include "AnthonysTimer.h"

/*
 * Use a file from the disk (frame.txt) as a substitute for a valid PVCAM
 */

void PVCAM_file(uns16 *frame, uns32 size){
    printf("Attempting to load the sample PVCAM file 'frame.txt'...\n");

    // Load frame.txt
    FILE *file;
    char fname[200]  = {"YourDirectory/frame - 1000 image.txt"};
    file   = fopen(fname,"r");
    rewind(file);
    int tempint;

    for(int i = 0; i<size; i++){
        fscanf(file,"%d",&tempint);
        frame[i] = tempint;
    }
    fclose(file);

    printf("Successfully loaded the sample PVCAM file 'frame.txt'.\n");
}

/*
 * Write a PVCAM frame to disk
 */

void frame2Txt(uns16 *frame, uns32 size){

    FILE *file;
    char fname[100]  = {"YourDirectory/frame.txt"};
    file   = fopen(fname,"w");

    for(int i = 0; i<size; i++){
        fprintf(file,"%d\n",frame[i]);
    }
    fclose(file);
}
/ * Convert a PVCAM frame (1D - uns16) to openCV (2D - IplImage) * 
* WARNING: FAILURE TO CORRECTLY SPECIFY frame_step will result in software crash. * 
* * If the frame supplied has m->cols * m->rows elements, then frame_step = 1; * 
* If the frame supplied has 2* m->cols * m->rows elements, then frame_step = 2; * 
* Some older versions of PVCAM (e.g. 2.9.11) confusingly doublecount each entry in frame. * 
* For matrix assignment, see: * 
* http://docs.opencv.org/doc/tutorials/core/how_to_scan_images/how_to_scan_images.html#performance-difference * 
*/ 

void frame2Ipl(uns16 *frame, IplImage* imgout, cv::Mat *m, rgn_type *region, int frame_step){

   // Determine the size of the frame
   intsizei  = (region->s2-region->s1) / region->sbin;
   intsizej  = (region->p2-region->p1) / region->pbin;
   intsizef = sizei * sizej * frame_step;        // allocated size is usually double, so 1.5 is safe

   // Setup the counting variable
   int i = 0;
   int j = 0;

   // Setup the pointer to matrix data for fast assignment
   uns16 *p=m->ptr<uns16>(j);

   // Loop through the frame vector and reshape it to an image
   for(int ct=0; ct<sizef; ct+=frame_step){

      // Update the current pixel
      i+=1;
      if(i>sizei){i=0;j+=1; p=m->ptr<uns16>(j);} 
      if(j>sizej){printf("Breaking to prevent segfault in frame2Ipl\n"); break;}
      p[i] = frame[ct];
   }
}
*imgout = (cv::Mat) *m;

void verify_region(rgn_type *region){

    // Verify serial direction
    int extent = region->s2-region->s1+1;
    int remainder = extent%region->sbin;
    region->s2-=remainder;

    // Verify parallel direction
    extent = region->p2-region->p1+1;
    remainder = extent%region->pbin;
    region->p2-=remainder;
}

void PVCAM_acquire( int16 hCam, uns16* frame, rgn_type region, uns32 size, int PVCAM_ACQUIRE_MODE)
{
    switch(PVCAM_ACQUIRE_MODE){
    case 0:
        PVCAM_acquire_standard(hCam, frame, region, size, PVCAM_ACQUIRE_MODE);
        break;
    case 1:
        PVCAM_acquire_circ(hCam, frame, region, size, PVCAM_ACQUIRE_MODE);
        break;
    case 2:
        PVCAM_acquire_circ(hCam, frame, region, size, PVCAM_ACQUIRE_MODE);
        break;
    default:
        printf("----->INVALID ACQUISITION MODE SPECIFIED\n\n");
        break;
    }
}
/ * Setup the PVCAM for imaging. Select the appropriate initialization scheme. */

int PVCAM_start(uns16* circBufferInMemory, int16 *hCam, uns32 *size, char *cam_name, rgn_type *region, int exp_time, int PVCAM_ACQUIRE_MODE){
    int pvstatus = -1;
    printf("Attempting to Connect to PVCAM...\n");
    switch(PVCAM_ACQUIRE_MODE){
        case 0:
            pvstatus = PVCAM_setup_standard(hCam, size, cam_name, region, exp_time);
            break;
        case 1:
            pvstatus = PVCAM_setup_circ(circBufferInMemory,hCam, size, cam_name, region, exp_time,PVCAM_ACQUIRE_MODE);
            break;
        case 2:
            pvstatus = 0;
            break;
        default:
            printf("------INVALID ACQUISITION MODE SPECIFIED\n\n");
    }
    return pvstatus;
}

int PVCAM_setup_standard(int16 *hCam, uns32 *size, char *cam_name, rgn_type *region, int exp_time){

// Try to close down the camera first in case some user error left it open
pl_cam_close( *hCam );
pl_pvcam_uninit();
pl_exp_uninit_seq();

// Initialize the PVCam Library and Open the First Camera
pl_pvcam_init();
pl_cam_get_name( *hCam, cam_name );
int pvstatus = pl_cam_open(cam_name, hCam, OPEN_EXCLUSIVE );

// Check for errors in the camera
int16 errorcode = pl_error_code();

// Figure out how much memory to allocate for the frame data (the size of the frame)
pl_exp_init_seq();
pl_exp_setup_seq( *hCam, 1, 1, region, TIMED_MODE, exp_time, size );

// Print out the connection status
if(pvstatus){printf("Successfully Connected to PVCAM via STANDARD buffering.\n\n");}
else{
    char_ptr msg = {"NO ERROR"};
    pl_error_message(errorcode,msg);
    printf("------>FAILED TO OPEN PVCAM. %s",msg);
}

return pvstatus;

/*
 * setup circular buffering WITH overwrite (type 1)
 */

int PVCAM_setup_circ(uns16* circBufferInMemory,int16 *hCam, uns32 *frame_size, char *cam_name, rgn_type *region, int exp_time, int PVCAM_ACQUIRE_MODE ){

    // Use Photometrics' demo function to start the PVCAM.
    if (InitPVCAM(g_Camera0_Name))
    {
        if (!OpenCamera(g_Camera0_Name))
        {
            printf("FAILED TO OPEN PVCAM. SYSTEM WILL NOW EXIT\n\n");
        }
    }
system("PAUSE");
CloseCameraAndUninit(g_hCam);
return 0;
}

else
{
CloseCameraAndUninit(g_hCam);
return 0;
}

// Use a modified version of Photometrics' demo code to
// start the continuous frame acquisition.
// setup the acquisition, FramesToAcquire specifies number
// of frames to acquire in this sequence
// exposureBytes will now hold the size of the whole buffer
// for all the images in the sequence
uns32 exposureBytes;

// setup the acquisition
if (PV_FAIL == pl_exp_setup_cont(g_hCam, 1, region, TIMED_MODE, exp_time, &exposureBytes, (int16) PVCAM_ACQUIRE_MODE))
{
  PrintErrorMessage(pl_error_code(), "pl_exp_setup_cont() error");
  CloseCameraAndUninit(g_hCam);
  return 0;
} else
{
  printf("Acquisition setup successful\n");
  *frame_size = exposureBytes;
}

// allocate memory for circular buffer of frames, size of
// one frame is returned in
// exposureBytes by pl_exp_setup_cont(), assuming this is
// more than 8-bit camera we allocate
// array of uns16, therefore we divide exposureBytes by 2
// and multiply by CIRC_BUFF_FRAMES so the circular buffer
// can hold 20 frames

  /* MOVED TO THE MAIN FUNCTION StageBoost.cpp to facilitate
destruction */

  // Start the acquisition
//start the continuous acquisition, again tell this
function size of buffer it has for the frames
//this is called only once after camera setup with
pl_exp_setup_cont()
    //this is a software trigger call in TIMED trigger
mode, in hardware trigger modes (Strobe, Bulb etc)
    //after this call camera waits for external trigger
signals to start every exposures
    if (PV_FAIL == pl_exp_start_cont(g_hCam,
circBufferInMemory, CIRC_BUFF_FRAMES*exposureBytes/2))
        {
            PrintErrorMessage(pl_error_code(),
"pl_exp_start_seq() error");
            CloseCameraAndUninit(g_hCam);
            delete [] circBufferInMemory;
            return 0;
        }
    else
        printf("Acquisition start successful
");

}/*-----------------------------------------------*
/* Acquisition modes for the PVCAM. Never invoked
except by PVCAM_acquire */
/*-----------------------------------------------*

/* Acquire a frame by the standard method (type 0) */
void PVCAM_acquire_standard( int16 hCam, uns16* frame, rgn_type region,
uns32 size, int PVCAM_ACQUIRE_MODE ){
    int16 status;
    uns32 not_needed;
    uns16* address = (uns16*) malloc(size*2);

    // Start the acquisition
    pl_exp_start_seq(hCam, address );

    // wait for data or error (Found by ADF to be an unnecessary
    // waste of time -- data is retrieved by the time start_seq returns)
    while( pl_exp_check_status( hCam, &status, &not_needed ) &&
    (status != READOUT_COMPLETE && status !=
READOUT_FAILED) );
// Check Error Codes
if( status == READOUT_FAILED ) {
    printf( "Data collection error: %i\n", pl_error_code() );
    return;
}

/* Assemble the frame */
for( int i = 0; i < size-1; i+=1){
    frame[i] = *((uns16*)address + i/sizeof(uns16));
}

/* Acquire a frame by the circular-overwrite method (type 1) */
void PVCAM_acquire_circ( int16 hCam, uns16* frame, rgn_type region, uns32 size, int PVCAM_ACQUIRE_MODE ){

    // Setup variables
    int16 status = 1;
    uns32 not_needed;

    // wait for data or error, ONCE, to ensure most recent frame
    while( pl_exp_check_cont_status( hCam, &status, &not_needed,&not_needed ) &&
          (status != READOUT_COMPLETE && status != READOUT_FAILED) );

    // Check for error codes
    if( status == READOUT_FAILED ) {
        printf( "Data collection error: %i\n", pl_error_code());
        return;
    }

    // Acquire the latest frame (if using status 1, CIRC_OVERWRITE)
    if(PVCAM_ACQUIRE_MODE==1){
        void_ptr address;
        if( pl_exp_get_latest_frame( hCam,&address) ) {
            // Store the address of the valid frame located at
            // DOESN'T WORK: frame = (uns16*) address;
            for(int i = 0; i < size-1; i++){
                frame[i] = *((uns16*)address + i);
            }

            // DOESN'T WORK: frame = (uns16*) address;
        }
// -OR- acquire the oldest frame (if using status 2, CIRC_NO_OVERWRITE)

if(PVCAM_ACQUIRE_MODE==2){
    void_ptr address;
    pl_exp_unlock_oldest_frame(hCam);

    if ( pl_exp_get_oldest_frame(hCam,&address) ) {
        // Store the valid frame located at address in my "frame" variable
        for(int i = 0; i<size-1; i++)
            frame[i] = *((uns16*)address + i/sizeof(uns16));
    }

    // Unlock the oldest frame for overwriting
    pl_exp_unlock_oldest_frame(hCam);
}
else{
    printf("pl_error_code = %d\n",pl_error_code());
}

SimpleSerial.h
/*
 * SimpleSerial.cpp
 * adapted by Anthony Fouad
 * Fang-Yen Group, 6/2014
 * *
 * Class for the serial port connection to the stage via BOOST libraries.
 * *
 * Adapted from 2 sites:
 * *
 *   https://groups.google.com/forum/#!topic/boostusers/0RpVAVSTQXQ
 * *
 *   http://www.webalice.it/fede.tft/serial_port/serial_port.html
 */

#ifndef SIMPLESERIAL_H_
#define SIMPLESERIAL_H_

299
#include <boost/asio.hpp>
#include <string>
typedef boost::asio::serial_port_base asio_serial;

class SimpleSerial
{
public:
    SimpleSerial(std::string port, unsigned int baud_rate);
    void writeString(std::string s);
private:
    boost::asio::io_service io;
    boost::asio::serial_port serial;
};

SimpleSerial.cpp
/*
   SimpleSerial.cpp
   adapted by Anthony Fouad
   Fang-Yen Group, 6/2014
   
   Class for the serial port connection to the stage.
   
   Adapted from 2 sites:
   https://groups.google.com/forum/#!topic/boostusers/0RpVAVSTQXQ
   http://www.webalice.it/fede.tft/serial_port/serial_port.html
*/
#include "stdafx.h"
#include <boost/asio.hpp>
#include "SimpleSerial.h"
typedef boost::asio::serial_port_base asio_serial;

   /*
   * Constructor.
   * \param port device name, example "/dev/ttyUSB0" or
   * "COM4"
   * \param baud_rate communication speed, example 9600 or
   * 115200
   * \throws boost::system::system_error if cannot open
   * the serial device
   */
SimpleSerial::SimpleSerial(std::string port, unsigned int baud_rate):io(),serial(io,port) {
    serial.set_option(asio_serial::baud_rate(baud_rate));
    serial.set_option(asio_serial::flow_control(asio_serial::flow_control::none));
    serial.set_option(asio_serial::parity(asio_serial::parity::none));
    serial.set_option(asio_serial::stop_bits(asio_serial::stop_bits::one));
    serial.set_option(asio_serial::character_size(8));
}

/**
 * Write a string to the serial device.
 * \param s string to write
 * \throws boost::system::system_error on failure
 */
void SimpleSerial::writeString(std::string s) {
    boost::asio::write(serial,boost::asio::buffer(s.c_str(),s.size()));
}

StageAccessoryFuncs.h
#ifndef STAGEACCESSORYFUNCS_H_
#define STAGEACCESSORYFUNCS_H_

void StageCommand(SimpleSerial *mySerial);
void StageJoyStick(SimpleSerial *mySerial, int vx, int vy);
void StageJoyStickManual(SimpleSerial *mySerial, int cauto, int *stagespeed);
void StageCalibrate(struct Images* pImg1,SimpleSerial *mySerial);
void StageGoRel(SimpleSerial *mySerial, int x, int y);  // Command the stage to move to a relative position
void StageAutomate(CvPoint* fluo_img_ctr, int cauto, CvPoint* max_loc,
                    SimpleSerial *mySerial, bool joystickflag, int* stagespeed);
void StageHalt(SimpleSerial *mySerial);

#endif

StageAccessoryFuncs.cpp
/*
Functions used to control the stage through a serial port
(Prior Optiscan II)
*

//Standard functions
#include "stdafx.h"
#include "stdlib.h"
#include "stdio.h"
#include <iostream>
#include <cmath>

// OpenCV 2.4 functions
#include "opencv\cv.h"
#include "opencv\highgui.h"

// Boost functions
#include "SimpleSerial.h"

// Anthony's functions
#include "AnthonysTimer.h"
#include "AnthonysColors.h"
#include "AnthonysCamFuncs.h"
#include "AnthonysCalculations.h"
#include "Images.h"
#include "StageAccessoryFuncs.h"

// PVCAM functions
#include "PVcamAccessoryFuncs.h"

using namespace std;

// Send commands to the stage by serial
void StageCommand(SimpleSerial *mySerial){
    // Setup variables
    int x, y;
    char cX[50];
    char cY[50];
    char base[50] = {"GR "};
    char command[50];

    try {

for(;;){
  // Get user commands
  cout << "ENTER relative x movement, then y movement.\n-9 to exit." << endl;
  cout << "-----------------------------" << endl;
  cin >> x; if(x==-9){return;}
  cin >> y; if(x==-9){return;}

  // Send commands to the serial
  sprintf(cX,"%d",x);
  sprintf(cY,"%d",y);
  strcpy(command,base);
  strcat(command,(const char*) cX);
  strcat(command, (const char*) cY);
  strcat(command,"\r\n");
  cout << "command = " << command << endl;
  mySerial->writeString(command);
  cout << "-----------------------------" << endl;
}

} catch( boost::system::system_error& e) {
  cout<<"BOOST Error: "<<e.what()<<endl;
  system("PAUSE");
  return;
}

/*
 * Move the specified serial-controlled stage by xx and y um.
 */
void StageGoRel(SimpleSerial *mySerial, int x, int y){

  char cX[50];
  char cY[50];
  char base[50] = {"GR "};
  char command[50];

  sprintf(cX,"%d",x);
  sprintf(cY,"%d",y);
  strcpy(command,base);
  strcat(command,(const char*) cX);
```c
strcat(command," ");
strcat(command,(const char*) cY);
strcat(command,"\r");
mySerial->writeString(command);
} /*
* Set the stage to move at a constant speed(s) vx and vy.
* NOTE 1: This prevents the manual joystick from being used.
* NOTE 2: Set VS 0 0 to stop this function
*/
void StageJoyStick(SimpleSerial *mySerial, int vx, int vy){
    char cX[50];
    char cY[50];
    char base[50] = {"VS "};
    char command[50];

    // Write the string command to the serial port
    sprintf(cX,"%d",vx);
    sprintf(cY,"%d",vy);
    strcpy(command,base);
    strcat(command,(const char*) cX);
    strcat(command," ");
    strcat(command,(const char*) cY);
    strcat(command,"\r");
    mySerial->writeString(command);
}

/*
* Move the stage manually using the arrow keys and joystick mode.
* DEPRECATED 7/7/2015, not used. To manually move the stage press J to switch to Joystick mode.
*/
void StageJoyStickManual(SimpleSerial *mySerial, int cauto, int
*stagespeed){
    switch(cauto){
        ///// Up arrow
        //case 2490368:
        // StageJoyStick(mySerial,0,-*stagespeed);
        // break;
```
/** Left arrow**
//case 2424832:
// StageJoyStick(mySerial,-*stagespeed,0);
// break;

/** Right arrow**
//case 2555904:
// StageJoyStick(mySerial,+*stagespeed,0);
// break;

/** Down arrow**
//case 2621440:
// StageJoyStick(mySerial,0,*stagespeed);
// break;

// Page up (speed up)
case 2162688:
    *stagespeed += 25;
    printf("Changed stage speed to: %d\n",*stagespeed);
    break;

// Page down (slow down)
case 2228224:
    *stagespeed -= 25;
    printf("Changed stage speed to: %d\n",*stagespeed);
    break;

// Home (stop, slow to 0)
//case 2359296:
// StageHalt(mySerial);
// break;

/** End (Also stop, slow to 0)**
//case 2293760:
// StageHalt(mySerial);
// break;

//default:
// printf("Unable to move stage manually. Key press not recognized.\n");
// break;

}
/ * not for general use. Get and segment the PVCAM frame for use in stage calibration. */ 

void retrieve_pvcam(struct Images* pImg1){

    // Shorten variables
    int W = FLUO_IMAGE_CLR->width;
    int H = FLUO_IMAGE_CLR->height;

    // Get video frame
    cv::Mat m = cvCreateMat(FLUO_IMAGE_CLR->height,FLUO_IMAGE_CLR->width,CV_16U); // temporary Mat used for PVCAM pixel manipulations
    PVCAM_acquire( pImg1->hCam, pImg1->frame, pImg1->region, pImg1->size, pImg1->PVCAM_ACQUIRE_MODE);
    frame2Ipl(pImg1->frame,pImg1->FluoImgGray,&m,&pImg1->region);

    // Exclude border areas from consideration
    cvDrawRect(pImg1->FluoImgGray,cvPoint( W * 0.0, H * 0.0 ),cvPoint( W * 1.0, H * 0.2),ww,-1,8); // Exclude top border
    cvDrawRect(pImg1->FluoImgGray,cvPoint( W * 0.0, H * 0.0 ),cvPoint( W * 0.2, H * 1.0),ww,-1,8); // Exclude left border
    cvDrawRect(pImg1->FluoImgGray,cvPoint( W * 1.0, H * 0.8 ),cvPoint( W * 1.0, H * 1.0),ww,-1,8); // Exclude right border
    cvDrawRect(pImg1->FluoImgGray,cvPoint( W * 0.0, H * 0.0 ),cvPoint( W * 0.8, H * 0.0 ),cvPoint( W * 1.0, H * 1.0),ww,-1,8); // Exclude bottom border

    // Convert the frame for use
    cvCvtColor(pImg1->FluoImgGray,pImg1->FluoImg,CV_GRAY2RGB);
    cvCvtScale(pImg1->FluoImgGray,pImg1->FluoImgGray8bit,1./110);

    // Segment video frame

    // Threshold image
    cvThreshold(pImg1->FluoImgGray8bit,pImg1->FluoImgThresh,pImg1->fluoThresh,255,CV_THRESH_BINARY_INV);

    // Check that at least 1 pixel has a value >0. if not, set a dummy region to 255 to prevent crashes
cvRectangle(pImg1->FluoImgThresh,cvPoint(pImg1->fluo_img_ctr->x-1,pImg1->fluo_img_ctr->y-1),cvPoint(pImg1->fluo_img_ctr->x+1,pImg1->fluo_img_ctr->y+1),ww,-1);

// Delete small objects
isolate_largest_object(pImg1->FluoImgThresh,pImg1->FluoImgThresh,pImg1->tempMem);

// find the outer contours
cvFindContours(pImg1->FluoImgThresh,pImg1->contourMem,&pImg1->Contours);

//Draw the contours on the user's image
cvDrawContours(FLUO_IMAGE_CLR,pImg1->Contours,yy,yy,1);

// Get the centroid of the object
fluo_moments(pImg1);

} /* Automatically calibrate the stage, using PVCAM (fluorescence image). */

void StageCalibrate(struct Images* pImg1, SimpleSerial *mySerial) {

    // Setup some variables
    int c, ct;
    double xc1, yc1, xc2, yc2, w;
    CvFont font1;
    cvInitFont(&font1, CV_FONT_HERSHEY_SIMPLEX,.5,.5,0,1,8);
    pImg1->OlFlag = 1;
    cvNamedWindow("Live feed",0);
    w = 0.08; // move step in mm

    // STEP 1: Move to the first wire (horizontal wire)
    ct = 0;
    for(;;) {
        ct = ct + 1;
        // Retrieve and segment video frame
        retrieve_pvcam(pImg1);

        // Apply text labels to video frame
        cvPutText(FLUO_IMAGE_CLR,"Find HORIZONTAL WIRE location",cvPoint(2,15),&font1,rr);
        cvPutText(FLUO_IMAGE_CLR,"ESC accepts the current frame for calibration",cvPoint(2,35),&font1,rr);
// Show video frame if valid frame address
if (FLUO_IMAGE_CLR != 0) {cvShowImage("Live feed", FLUO_IMAGE_CLR);
    c = cvWaitKey(1);

    // Allow user to change threshold
    if (c==56) {pImg1->fluoThresh += 5; printf("FluoImg Threshold = %d\n", pImg1->fluoThresh);
        if (c==55) {pImg1->fluoThresh -= 5; printf("FluoImg Threshold = %d\n", pImg1->fluoThresh);
    }

    // break loop if esc is pressed
    if (c==27) {break;}
}

// STEP 2: Get Y pixel calibration

    // Get the centroid before motion
    retrieve_pvcam(pImg1);
    cvSaveImage("calib_y_before.jpg", FLUO_IMAGE_CLR);
    xc1 = pImg1->Centroid->x;
    yc1 = pImg1->Centroid->y;

    // Move the stage some mm up and watch it
    StageGoRel(mySerial, 0, w*1000);
    retrieve_pvcam(pImg1);
    cvShowImage("Live feed", FLUO_IMAGE_CLR);
    cvWaitKey(500);
    retrieve_pvcam(pImg1);
    cvShowImage("Live feed", FLUO_IMAGE_CLR);
    cvWaitKey(500);

    // Get the centroid after motion
    retrieve_pvcam(pImg1);
    cvSaveImage("calib_y_after.jpg", FLUO_IMAGE_CLR);
    xc2 = pImg1->Centroid->x;
    yc2 = pImg1->Centroid->y;

    pImg1->mm_per_pix_y = w / abs(yc2-yc1);
    cout << "Moved " << w << " mm up : moved " << abs(yc2-yc1) << " pixels up : Ratio = " << pImg1->mm_per_pix_y << endl;

// STEP 3: Move to the second wire (vertical wire)
ct = 0;
for (;;) {
    ct = ct + 1;
    // Retrieve and segment video frame
    retrieve_pvcam(pImg1);
// Apply text labels to video frame
cvPutText(FLUO_IMAGE_CLR,"Find VERTICAL WIRE location",cvPoint(2,15),&font1,rr);
cvPutText(FLUO_IMAGE_CLR,"ESC accepts the current frame for calibration",cvPoint(2,35),&font1,rr);

// Show video frame if valid frame address
if(pImg1->MainImg != 0){cvShowImage("Live feed",FLUO_IMAGE_CLR);
    c = cvWaitKey(1);
}

// Allow user to change threshold
if(c==56){pImg1->fluoThresh += 5; printf("FluoImg Threshold = %d\n",pImg1->fluoThresh);}
if(c==55){pImg1->fluoThresh -= 5; printf("FluoImg Threshold = %d\n",pImg1->fluoThresh);}

// break loop if esc is pressed
if(c==27){break;}

// STEP 4: Get X pixel calibration

// Get the centroid before motion
retrieve_pvcam(pImg1);
cvSaveImage("calib_x_before.jpg",FLUO_IMAGE_CLR); xc1 = pImg1->Centroid->x; yc1 = pImg1->Centroid->y;

// Move the stage some mm right and watch it
StageGoRel(mySerial,w*1000,0); retrieve_pvcam(pImg1); cvShowImage("Live feed",FLUO_IMAGE_CLR); cvWaitKey(500);
retrieve_pvcam(pImg1); cvShowImage("Live feed",FLUO_IMAGE_CLR); cvWaitKey(500);

// Get the centroid after motion
retrieve_pvcam(pImg1);
cvSaveImage("calib_x_after.jpg",FLUO_IMAGE_CLR); xc2 = pImg1->Centroid->x; yc2 = pImg1->Centroid->y;

pImg1->mm_per_pix_x = w / abs(xc2-xc1); cout << "Moved " << w << " mm right : moved " << abs(xc2-xc1) << " pixels left : Ratio = " << pImg1->mm_per_pix_y << endl;
cvDestroyWindow("Live feed");
system("PAUSE");
return;
}

/*
   * Automatically center the desired point along the worm centerline OR
fluorescence centroid. Requires segmentation performed first on this
frame.
   */

void StageAutomate(CvPoint* fluo_img_ctr, int cauto, CvPoint* max_loc,
SimpleSerial *mySerial, bool joystickflag, int *stagespeed){

    // Force fluorescence targeting for now.
    // Determine the difference between the object centroid and the
image focus point along the worm, in millimeters.
    double diffx, diffy;

    // If using fluorescence targeting
    int xbar = fluo_img_ctr->x;
    int ybar = fluo_img_ctr->y;
    double scalefactor = 0;

    // Use approximate scalefactor appropriate to the
fluo camera in use (6.67 um/pixel with 4x binning and image in small
corner of sensor, 20x, 11/19/14)
    if(PVCAM_ACQUIRE_MODE_DEF<0){scalefactor = 0.75;}
    else{scalefactor=STAGE_SCALE_20X;}

    diffx = STAGE_DIRECTION*((double) max_loc->x - (double) xbar)* *stagespeed/100;
    diffy = STAGE_DIRECTION*((double) max_loc->y - (double) ybar)* *stagespeed/100;

    // Find out if we should move
    int ifmove = (abs(((double) max_loc->x - (double) xbar)) > STEP_MIN ||
abs(((double) max_loc->y - (double) ybar)) > STEP_MIN);

    //if (!ifmove){diffx=0;diffy=0;};

    // IF we need to move by more than 1 um, Order the
stage to move in that direction
    if(joystickflag==0){
        StageJoyStick(mySerial,diffx,diffy);
    }
//StageGoRel(mySerial,(int)-diffx/4,(int)-diffy/4); // Go at quarter speed for Jumpy mode
}

// Go back to the camera after the order is issued (not necessarily after the order finishes executing)
return;
}

/*
 * Force the stage to halt
 */

void StageHalt(SimpleSerial *mySerial){
    StageJoyStick(mySerial,0,0);
}

// stdafx.h
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//
#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>

/* HARDCODED PARAMETERS USED BY STAGEBOOST */

// Specify whether using PVCAM variables (16bit) or ICC variables (8bit)
#define PVCAM_ACQUIRE_MODE_DEF 1
// --1 use imagingsource FL, 0=standard , 1=CircularOverwrite , 2=CircularNoOverWrite
#define WHICH_PVCAM 0
// 0 for Cascade 1k (DEPRECATED, MAY NOT WORK). 1 for optiMOS

311
```c
#define FLUO_IMAGE_CLR pImg1->FluoImg
    // imagingsource returns 8bit images. PVCAM returns 16bit images
#define FLUO_IMAGE_GRY pImg1->FluoImgGray
#define FLUO_THREAD_PARAMETERS &GetPvcamFrame, pImg1->hCam, pImg1->frame, pImg1->region, pImg1->size, pImg1->PVCAM_ACQUIRE_MODE, &rflag2
    // &GetDarkfieldFrame, cv_cap_fl, img_color_fl, &rflag2

    // Specify whether FITC subtractions should be performed during stimuli. FITC leaks into the mCherry channel
#define FITC_MODE 0

    // Specify whether to use "large save images" (also writes high quality fluo image stream to disk)
#define SAVE_LARGE 0

    // Specify whether the stage movements should be inverted (-1) or normal (1)
#define STAGE_DIRECTION 1

    // Specify whether to use the inverse threshold when segmenting brightfield, or normal threshold when segmenting darkfield. Only used for tracking modes 31-39
    // For dark field: use CV_THRESH_BINARY and 0
    // For brightfield: use CV_THRESH_BINARY_INV and 0
    // Inverting the image ("INVERT_MODE") is not very effective but inverting the threshold mode works perfectly. INVERT_MODE should always be 0.
#define THRESH_MODE CV_THRESH_BINARY // CV_THRESH_BINARY_INV
#define INVERT_MODE 0

    // Parameters loaded from disk
#define PARAM_FILE_PVCAM_THRESH "YourDirectory/StageBoost/Parameters/parameter_pvcam_threshold.txt"
    // Fluorescence threshold
#define PARAM_FILE_STAGE_SPEED "YourDirectory/StageBoost/Parameters/parameter_stage_speed.txt"
    // Stage speed for manual joystick

    // Specify how often to record images to disk (if applicable) and display them to user. This slows down the system! A value of 1, means show every frame. 2 means show every other. 3 means every third
#define SAVE_INTERVAL 1
#define SHOW_INTERVAL 2

    // Specify the maximum allowed framerate (useful to stabilize the framerate). Mode 12 uses 18.
```
#define MAX_FRAME_RATE 40

// Specify the puncta forecast interval (milliseconds)
define DT_PRED 0

// Specify the relative position of the fluorescent image center, in IMAGE COORDINATES. absolute center would be (0.5,0.5)
define CENT_X 0.55
define CENT_Y 0.55

// Specify the minimum step size, in micrometers, that the stage is allowed to make if the worm deviates from the image center
define STEP_MIN 10

// Specify the stage scaling, in microns / pixel, calculated by option "l" in the main menu (calibrate stage)
define STAGE_SCALE_20X 8 // 6.67 for 20x on Cascade 1k

/*
   20X, PVCAM, small spot on sensor -- 6.67
   20X, ICcam, --
   ~0.75
*/

// Specify the stage speed, in pixels per frame, calculated manually by moving a paralyzed worm around and letting the stage recenter
define STAGE_SPEED_X_20X 2.85
#define STAGE_SPEED_Y_20X 4
BIBLIOGRAPHY


Kiehn, O. (2016). Decoding the organization of spinal circuits that control locomotion. Nat Rev Neurosci 17, 224-238.


Sørenson, T. (1948). A Method of Establishing Groups of Equal Amplitude in Plant Sociology Based on Similarity of Species Content and Its Application to Analyses of the Vegetation on Danish Commons (I kommission hos E. Munksgaard).


