Distribution Agreement

In presenting this thesis as a partial fulfillment of the requirements for a degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis in whole or in part in all forms of media, now or hereafter now, including display on the World Wide Web. I understand that I may select some access restrictions as part of the online submission of this thesis. I retain all ownership rights to the copyright of the thesis. I also retain the right to use in future works (such as articles or books) all or part of this thesis.

Sam Miller April 12, 2020

Integration of the Control Software for the Electromagnetic and Permanent Magnet Tweezers

by

Sam Miller

Laura Finzi

Advisor

David Dunlap

Advisor

Physics

Laura Finzi

Advisor

David Dunlap

Advisor

Shun Cheung

Committee Member

Effrosyni Seitaridou

Committee Member

2021

Integration of the Control Software for the Electromagnetic and Permanent Magnet Tweezers

By

Sam Miller

Laura Finzi

Advisor

David Dunlap

Advisor

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Physics

2021

Abstract

Integration of the Control Software for the Electromagnetic and Permanent Magnet Tweezers
By Sam Miller

` Magnetic tweezers are single-molecule instruments that are often used to impart forces and torques on DNA molecules. They typically have a pair of permanent magnets attached to a motor that can be vertically translated and rotated to adjust the magnitude and orientation of the magnetic field, respectively. Then, DNA molecules are chemically attached to the microscope stage and a paramagnetic bead, which translates a magnetic force from the magnetic field onto the molecule. An objective captures images of the samples and computer software uses diffraction pattern analysis to determine the position, magnetic force, and tether length of the DNA molecule. The electromagnetic tweezers developed by the Finzi-Dunlap lab improve upon the existing technology by generating a magnetic field with current-carrying wires. Although this set-up offers numerous advantages over traditional magnetic tweezers, the program that drives the electromagnetic tweezers existed independently of the permanent magnet program. As a result, users of the electromagnetic tweezers were required to launch and engage with two separate, non-communicative programs. In order to solve this problem, I introduced a new “Axis-type” class that mimicked the behavior of the existing classes that represented the motors in the permanent magnet program. The ElectromagnetAxis class wraps all of the functionality required to set the current values that map to the strength and orientation of the resulting magnetic field. Additionally, this class collects all of the relevant data from the electromagnetic tweezers program into a single place. As such, ElectromagnetAxis objects allow the electromagnet program to access the particle-tracking and data-processing components of the permanent magnet program. The result is a single, unified program that launches and closes simultaneously, allows for the manipulation of the magnetic field through the design introduced with the electromagnetic tweezers program, and tracks beads and processes data through the software from the permanent magnetic tweezers program. Finally, the unified software shows deference for the tenets of the object-oriented programming

paradigm through increased encapsulation of methods and properties. Therefore, users of the Finzi-Dunlap lab's electromagnetic tweezers only need to operate a single program to investigate the effects of magnetic forces and torques on DNA molecules.

Integration of the Control Software for the Electromagnetic and Permanent Magnet Tweezers

By

Sam Miller

Laura Finzi

Advisor

David Dunlap

Advisor

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Physics

2021

Acknowledgements

A special thanks to:

Laura Finzi

David Dunlap

Joe Piccolo

Josh Mendez

Shun Cheung

Effrosyni Seitaridou

Nancy, Andrew, and Jacob Miller

Table of Contents

Transcription and Supercoiling

Transcription of the deoxyribonucleic acid (DNA) (Figure 1a) is the process by which the DNA double helix is unwound, and one of the two strands is used as a code to synthesize ribonucleic acid (RNA) (Figure 1b) (Campbell, 2016). RNA can then be processed by ribosomes to assemble proteins (messenger RNA, mRNA), used to deliver the correct amino acid for mRNA polymerization (transfer RNA, tRNA), integrated as part of large molecular machines, such as the ribosome (ribosomal RNA, rRNA), exploited as catalyst (Figure 1c, Top) (Campbell, 2016). Transcription is therefore a fundamental process for the life of a cell and is highly dependent upon the physical structure of the DNA double helix. This is because transcription is carried out and facilitated by a number of enzymes which can only bind to DNA in specific configurations. In particular, RNA Polymerase is the enzyme which unzips the double helix, generates the transcription bubble, and polymerizes an RNA strand in the 5' to 3' direction which is complementary to the 3'-5' template, or coding, DNA strand (Campbell, 2016). Unzipping naturally engenders the unwinding of DNA behind and winding ahead of the transcription fork, resulting in contortions of the molecule known as supercoils (Figure 1c, Bottom) (Wang, 1987). In order to alleviate this problem, topoisomerases are employed ahead of the RNA Polymerase molecule. These enzymes relax the DNA by relieving the molecule of the supercoils through the severing of the phosphate backbone. Then, the DNA molecule is relaxed either partially or completely and DNA transactions, such as transcription, DNA replication (duplication of the double stranded DNA) can continue.

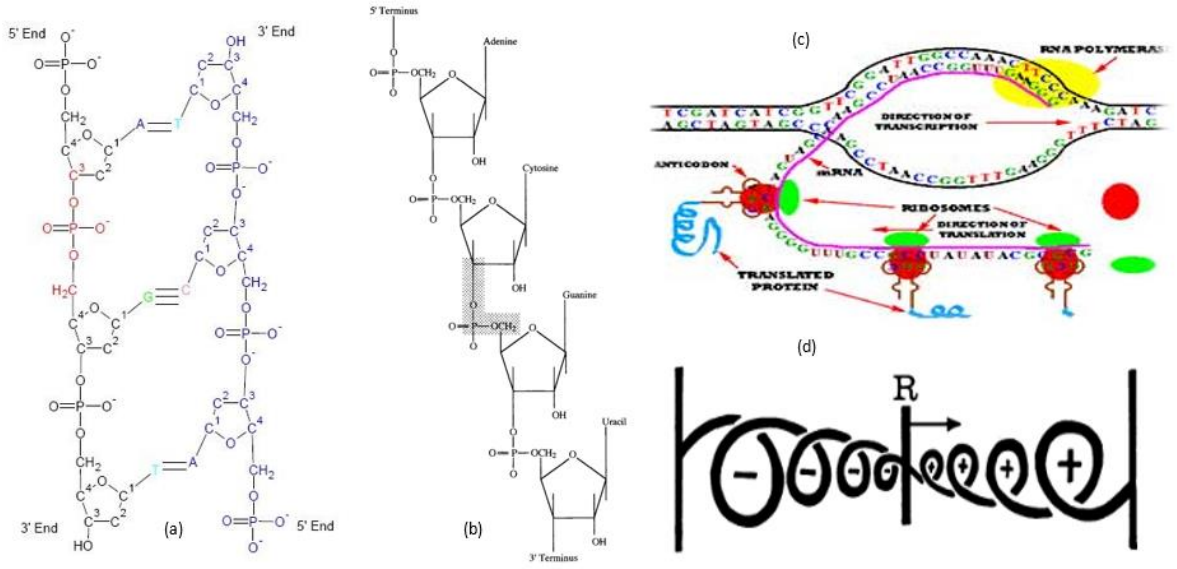
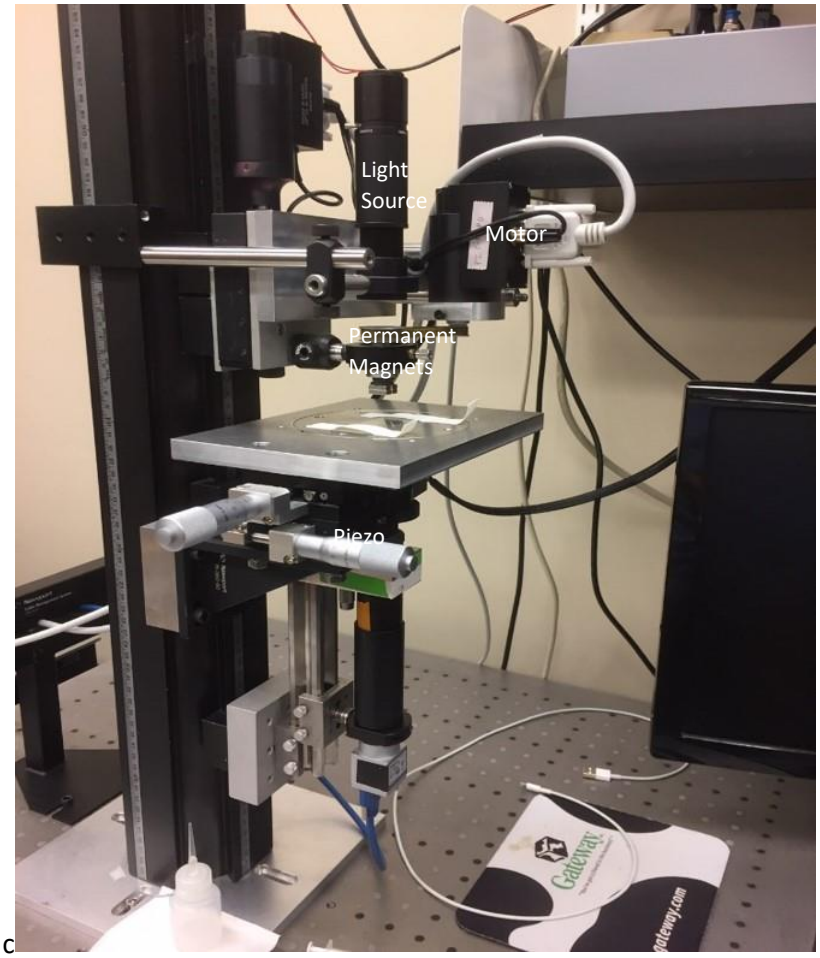


Figure 1: (a) (Campbell, 2017) Nucleic acids are biomolecules composed of repeating units known as nucleotides. Each nucleotide has a phosphate group, a pentose, and a nitrogenous base. In DNA, the pentose is deoxyribose, and the bases may be either adenine, guanine, cytosine, or thymine. Bonds between these bases form a double helix structure. (b) (Campbell, 2017) RNA molecules are also nucleic acids with ribose as their 5-carbon sugar. The nitrogenous bases are the same as those in DNA except that thymine is replaced with uracil. RNA molecules, unlike DNA, are single-stranded. (c) (Aryal, 2019) Transcription is the production of mRNA, a copy of a DNA strand that is used to assemble proteins in ribosomes. The DNA double helix is unwound, and RNA Polymerase enzymes assemble the mRNA. (d) (Wang, 1987) The transcription complex involved in unwinding the DNA and forming the nascent mRNA naturally engenders positive supercoils in the DNA ahead of the transcription bubble and negative supercoils behind it.

Supercoils also play a major role in gene expression (Charles Dorman, 2016). The overwinding and underwinding of DNA regulates the ability of certain enzymes, such as topoisomerases, to bind to the strand in the first place. Indeed, supercoiling can be either free or topologically constrained by proteins, such as the histone octamer around which eukaryotic DNA tightly winds to form a nucleosome, the building unit of chromatin. In this form, DNA cannot be transcribed, nor replicated. Therefore, as biological processes like transcription and replication are sensitive to the topological state of the DNA (Dorman, 2019), DNA supercoiling becomes an important parameter for the regulation of such processes (Finzi & Olson, 2016).

Magnetic Tweezers

Magnetic tweezers usually consist of a pair of permanent magnets -- often alloys of rare earth elements that are ferromagnetic -- mounted above the stage of an optical microscope (Figure 2) (Finzi & Dunlap, JMB 2010). The magnetic field established between the two magnets is used to exert forces and torques on paramagnetic beads attached to single DNA molecules attached to the surface of a glass microchamber. The pair of magnets are connected to a stepper motor or voice coil actuator, that is used for vertical translation along the microscope optical axis, and a different stepper motor for rotation. In a magnetic tweezers experiment, the molecules of interest are chemically anchored to the glass surface of a flow chamber at one end and to a super-paramagnetic bead with a typical diameter of 1 micron through digoxigenin-anti-digoxigenin or streptavidin-biotin bonds, respectively. The field from the pair of magnets induces a force on the tethered bead which stretches the DNA attached to the surface of the flow chamber, changing the bead's diffraction image, and the amplitude of its thermal fluctuations. Thus, by tracking the bead motion and recording images with a camera and appropriate computer software, one can infer its distance from the surface and, consequently, the end-to-end distance of the tethering molecule of interest (Kovari, 2019).



c

Figure 2: Magnetic tweezers. The pair of permanent magnets can be vertically translated and rotated with motors to create a magnetic field of variable strength and orientation.

An important part of the magnetic tweezers set up is the motor. The vertical translation of the permanent magnets allows for the direct manipulation of the magnetic field strength. Likewise, the rotation of the permanent magnets rotates the associated magnetic field. The rotation of the bead is an effective method for twisting and inducing supercoils in a DNA molecule. Each turn of the magnetic field increases or decreases, depending on the direction, the twist of the DNA. Once the DNA curls to form a loop, further turns of the magnetic field contribute to the writhe, wherein the axis of the double helix crosses over itself. (Wenxuan Xu et al, 2021).

By modelling the bead-tether complex in a thermal bath under the influence of the magnetic force as an inverted pendulum, its motion can be succinctly summarized with the equipartition theorem, which relates the average energy of a particle to the temperature of the bath. The following equation is the result of modelling the average energy of a particle as a three-dimensional harmonic oscillator:

$$E_{tot} = \frac{1}{2}k_x x^2 + \frac{1}{2}k_y y^2 + \frac{1}{2}k_z z^2 \ (1)$$

Then, the equipartition theorem says that the average of the square of the particle's position in each dimension is equal to half the Boltzmann constant times the temperature:

$$\frac{1}{2}k_x < x^2 > = \frac{1}{2}k_b T$$

$$\tfrac{1}{2}k_y < y^2 > = \tfrac{1}{2}k_b T \ (2)$$

$$\frac{1}{2}k_z < z^2 > = \frac{1}{2}k_b T$$

If the anchor point is the origin, the average extension of the DNA molecule is:

$$L_{avg} = \sqrt{< x^2 > \ + < y^2 > \ + < z^2 >} \ (3)$$

Finally, the magnetic force on the bead can be calculated by treating the bead-tether complex as an inverted pendulum:

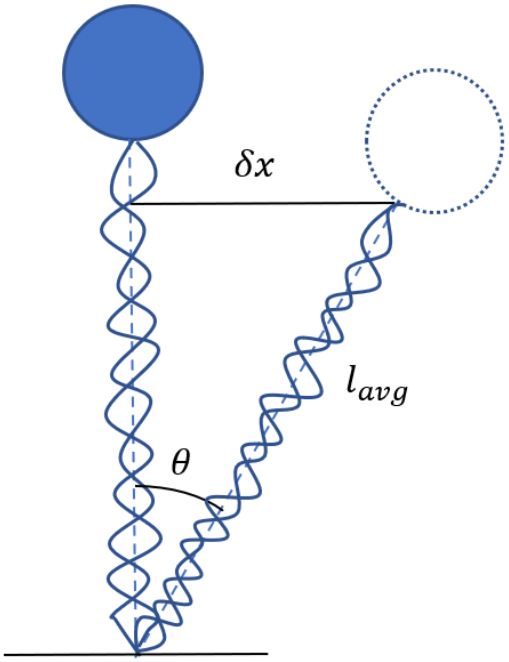


Figure 3: Tethered beads in a magnetic tweezer can be treated as inverted pendula in a simple model with which the magnetic force can be determined. In this case, δx is the fluctuation of the bead in the x direction, θ is the angle from the horizontal, and l_{avg} is the average tether length.

From Figure 3, an expression for the force in the x-direction can be found:

$$F_x = \big||F|\big|sin\theta = \frac{\big||F|\big|\delta x}{l_{avg}} (4)$$

As expected, the spring constant in the x-direction of an inverted pendulum is the force on it divided by its length. Therefore, this expression can be substituted into the preceding equipartition theorem equation, yielding the desired expression for the magnetic force:

$$\frac{\frac{1}{2}F < \delta x^2 >}{l} = \frac{1}{2}k_bT$$

$$F = \frac{k_bTl}{< \delta x^2 >} (5)$$

Therefore, a magnetic tweezers program is able to calculate the average tether length of the molecule and the force felt by the bead simply by averaging the x, y, or z positions of the bead over time.

Electromagnetic Tweezers

A limitation of the most common implementation of magnetic tweezers is that the motors can be costly because their movement must be extremely precise. What is more, the physical vibrations produced by the motor during translation may disturb the sample. A magnetic tweezer measurement depends on accurate tracking of the position of beads. Vibrations from the motor artificially translate the beads and interfere with accurate tracking. Furthermore, the rate of change of the magnetic field is limited by the minimum and maximum speeds of the motors (Piccolo, 2021). These issues motivated the development of the electromagnetic tweezers which utilize current-carrying wires to produce the magnetic field.

Hardware

The electromagnetic tweezers in the Finzi lab use an Arduino functioning as a current controller to split an input current from a DC voltage source into two output currents. This current controller uses pulse width modulation to map the input current on a scale of 0-255 arbitrary units. The two resulting currents flow through two pairs of solenoids which generate two magnetic fields (Figure 4, Right). The magnitude of this resulting magnetic field can be changed by adjusting the amplitudes of the input currents. The orientation of the magnetic field can be specified by modulating the amplitudes of the two currents, since the resultant field is the vector sum of the two component magnetic fields.

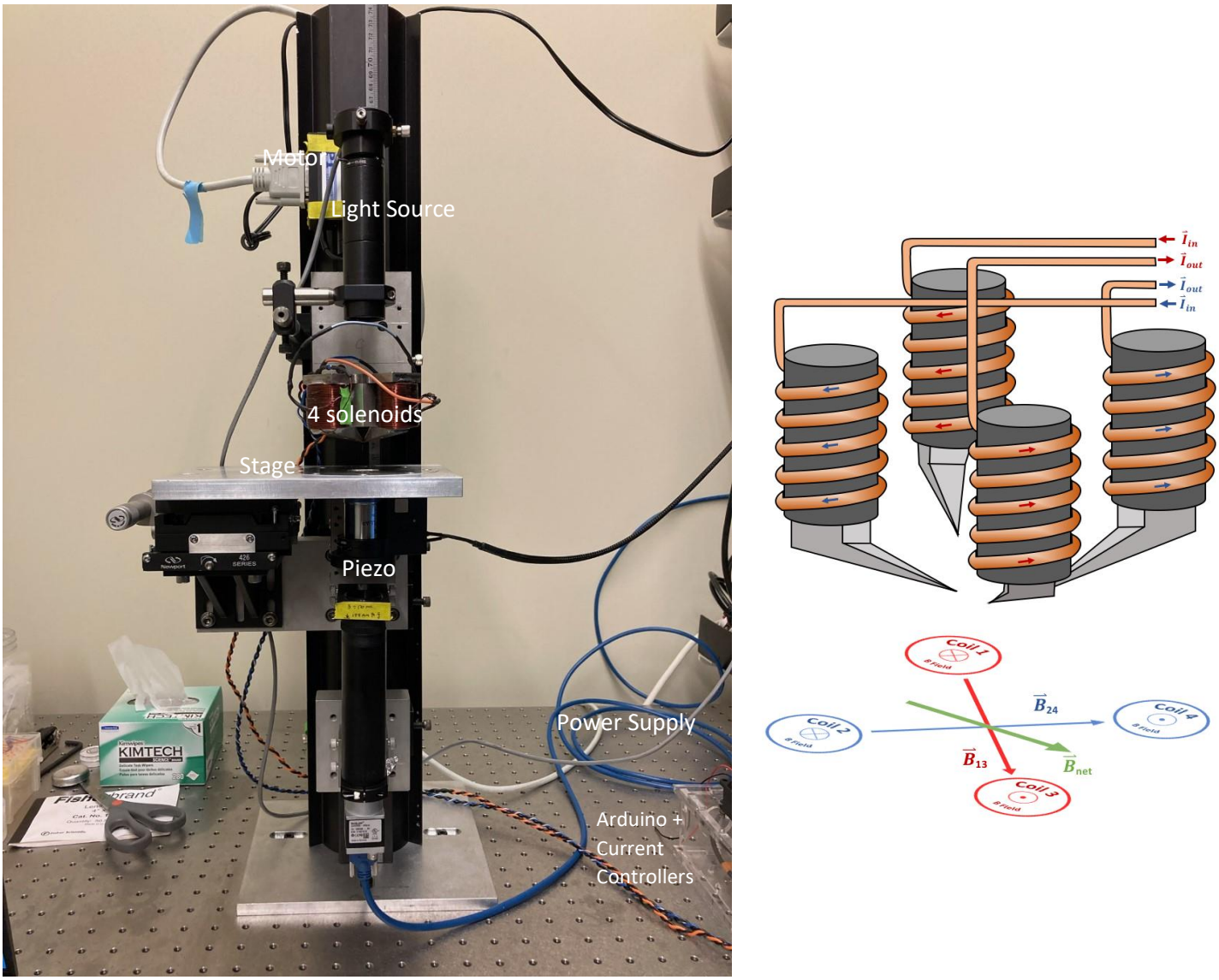


Figure 4: Electromagnetic tweezers. Left: photograph of the electromagnetic tweezer microscope in the Finzi/Dunlap lab. Right: Generation of a magnetic field from 2 input currents. Each current runs through a pair of solenoids, creating a magnetic field. The vector sum of these two magnetic fields determines the orientation of the net magnetic field. (Piccolo, 2021)

Software

There exist two separate programs for operating the magnetic tweezers in the Finzi/Dunlap lab in the physics department at Emory University. The first program operates the permanent magnetic tweezer with stepper motors for translation and rotation of the permanent magnets. It is well developed and has the user interface (UI) shown in Figure 5:

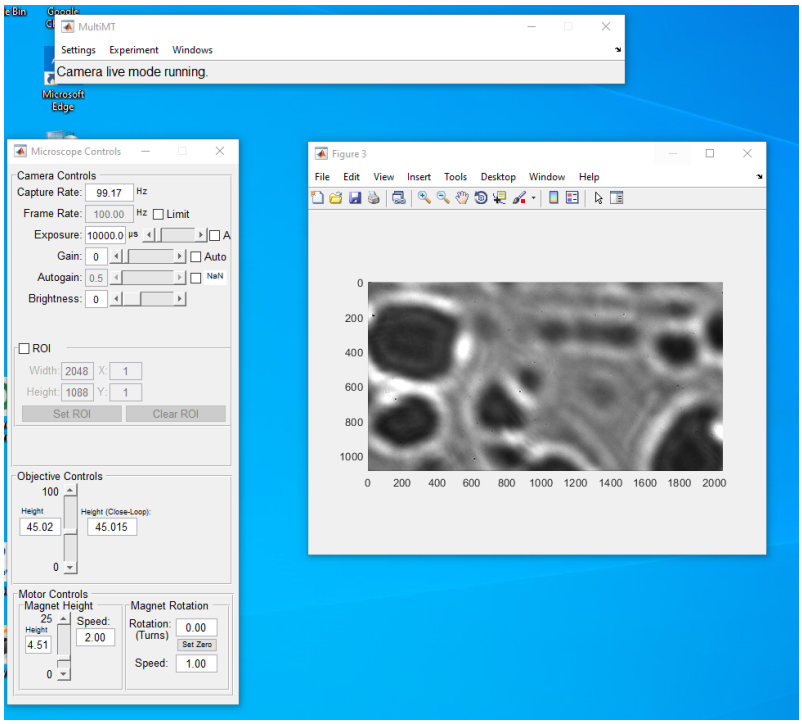


Figure 5: UI of the permanent magnet program.

In Figure 5, the uppermost panel is the parent window of all other windows in the program. This is where the user accesses different simple experiments, tracking controls, and other settings. The leftmost window contains the camera controls and the motor controls. The user can adjust the strength and orientation of the magnetic field from here. Finally, the rightmost window is the image displayed by the video camera. Regions of interest are drawn with small rectangles on this image to specify locations in which to search for tethered beads. Throughout the operation of the magnetic tweezers, the camera produces a video stream that pauses during the instances in which the program needs time to think,

process data, or move the motor. In each video frame, the program searches the regions of interest for the particles and records the positions. It is then able to employ equations (3) and (5) to produce the average tether length of the DNA molecule and force on the paramagnetic bead.

Meanwhile, in a separate program, the electromagnetic tweezers have the UI shown in Figure 6:

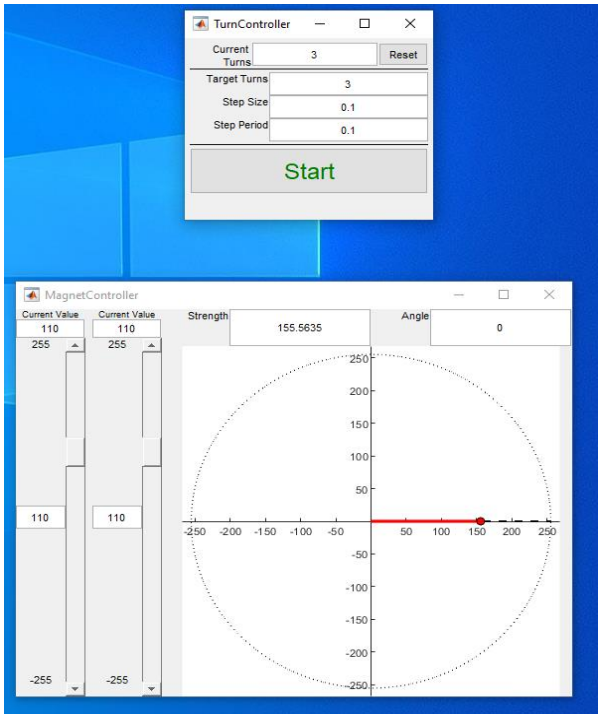


Figure 6: UI of the electromagnet program. The bottom window is the Magnet Controller. Users can adjust the strength and orientation of the magnetic field with the data fields on the top or by dragging the dot anywhere within the central circle. The magnetic field can also be adjusted through the sliders on the left that correspond to the input currents. The Turn Controller is the top window. A user can easily rotate the magnetic field by entering a value in the Target Turns field and pressing start. Then, the magnetic field is rotated until the Current Turns field matches Target Turns. The rotation of the magnetic field proceeds with equally sized steps determined by Step Size. A step is taken after every period, determined by the Step Period parameter.

Problem and Rationale

The main issue with the existing software for the Finzi-Dunlap lab's electromagnetic tweezers is that the graphical user interfaces (GUIs) for the permanent magnet and the electromagnet do not communicate. The permanent magnetic tweezers are currently controlled by software developed by former post-doc in the lab, Daniel Kovari. His program was developed in MATLAB with the outdated app-design environment GUIDE (Graphical User Interface Design Environment) which is used to create GUIs. GUIDE allows a programmer to create interactable windows with buttons, sliders, and data fields, and more through a drag-and-drop interface that automatically generates the associated code afterwards. Kovari's software allows the user to directly specify the position (height above the sample) and orientation of the permanent magnets. For the electromagnetic tweezers, however, these parameters are no longer relevant; they are supplanted by the relative amperages of the input currents.

Meanwhile, the electromagnetic tweezer software that was developed by another former post-doc, Josh Mendez, was not created through GUIDE. The Magnet Controller allows the user to directly specify the strength and orientation of the magnetic field, while the Turn Controller can be used to automate rotation. However, the electromagnetic software was not integrated with the particle-tracking and data-recording aspects of the permanent magnet program.

As such, a user of the electromagnetic tweezers will be forced to launch both programs. Then, the user must operate both programs at the same time for the duration of the experiment. This is a cumbersome and inefficient way to take data with the electromagnetic tweezers that requires tedious synchronization of the particle tracking data stream with electromagnet settings recorded by the user for analysis.

In order to solve this problem, I unified the two programs by creating a new "Axis-Type" class that mimics the behavior of the C862 and C843 motor classes that are used to set the strength and

orientation of the resulting magnetic field. Classes are fundamental tenets of object-oriented programming. They are blueprints for objects which wrap important data and functions into manipulable variables. As such, objects derived from my ElectromagnetAxis class compact most of the user's input data from both programs into a single manageable instance that can interact with the particle-tracking and data-collection components of the former permanent magnet program.

As a result, the automated simple experiments will be available to the user. This automates many of the typical data collection experiments that the user will likely want to perform. In a force extension experiment, the current – and consequently, the magnitude of the magnetic field – is gradually increased by a constant step size over time. The rotational analogue of this experiment is known as a Chapeau curve. A record XYZ experiment simply tracks the location of the bead over time. In the previous version of the software, where both programs were isolated from one another, these simple experiments were only available through the permanent magnet GUI. With the ElectromagnetAxis class, this range of capabilities will become available to users of the electromagnetic tweezers as well. These simple experiments are crucial in plotting data, including force-tether length curves and tether length-turn curves, which typically have the forms displayed in Figure 7. Finally, the user will no longer have to launch and manage two separate programs in order to work with a single machine.

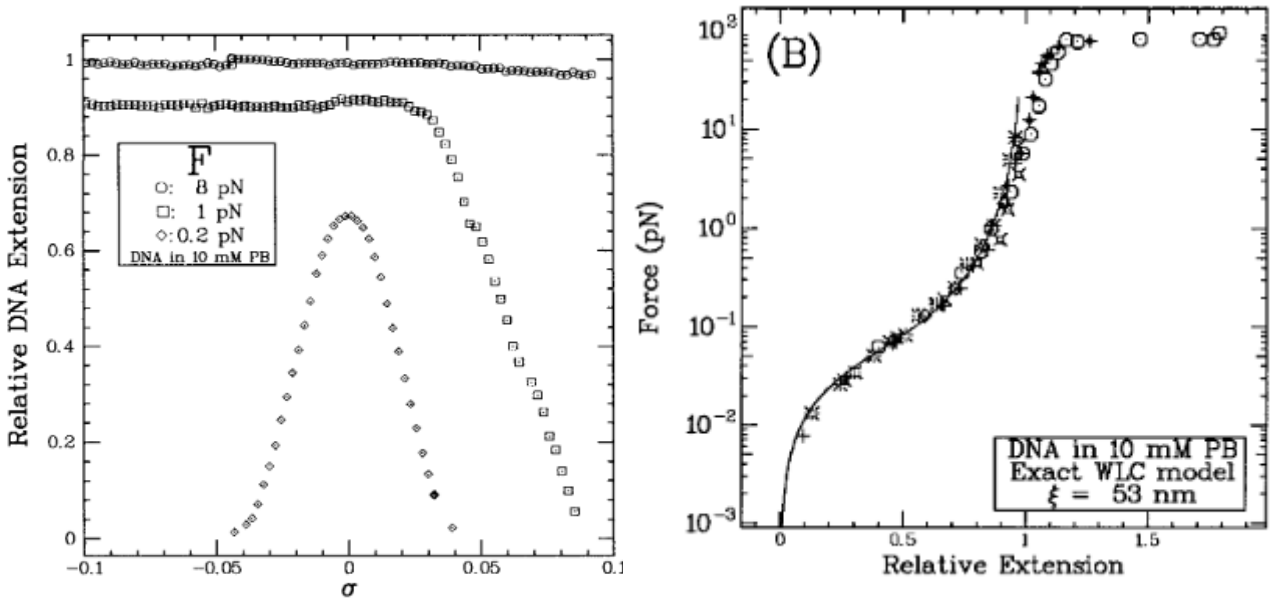


Figure 7 (Strick, 1998): Typical plots of interest in a magnetic tweezers experiment. The figure on the left plots tether length against supercoils. At lower forces, a DNA molecule will contract as more supercoils are formed (either positive or negative). However, at intermediate and higher forces, the molecule's extension is not reduced with negative supercoils. The figure on the right plots tether length against forces. Generally, as the force grows, the molecule's extension increases. Then, when the force is reduced, the tether shortens, with minimal hysteresis.

Simultaneous Launching and Closing of the Two Programs

The permanent magnet program is driven by the file MultiMTgui_main.m which begins an opening cascade that initializes the GUI. Similarly, the electromagnet GUI is launched by a driver file called launch_Controller.m which instantiates the Magnet Controller and Turn Controller objects that are used to set the magnetic field but do not save any settings. As shown in Figure 8, the user must run both of these files individually in order to operate the electromagnetic tweezers. When they are done, they must close the parent windows for each program.

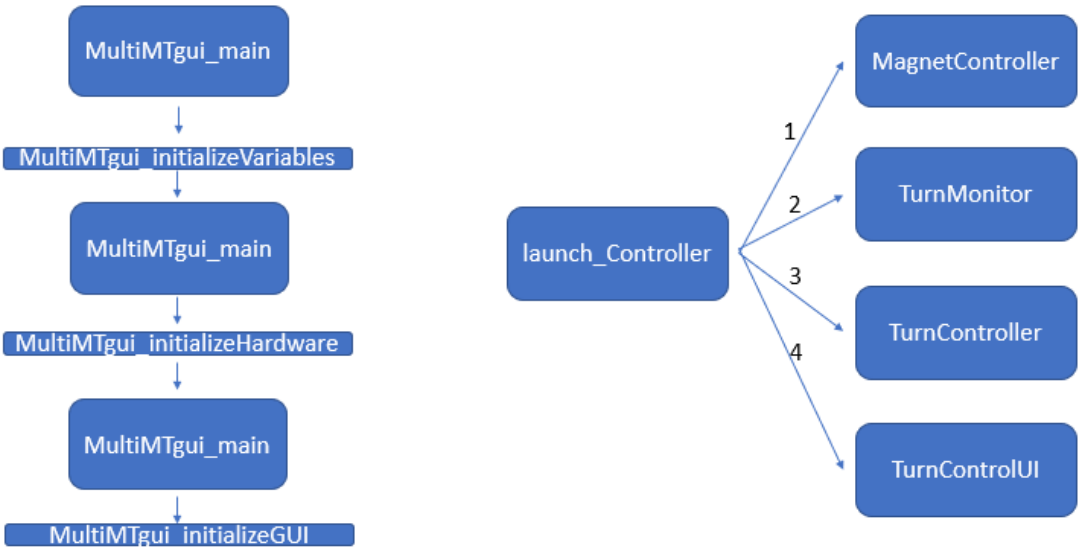


Figure 8: Opening code cascades for both the permanent magnet and electromagnet programs. The permanent magnet program (left) is launched through the driver file MultiMTgui_main.m which makes calls to various initialization files that open communication with the hardware and launch the GUI. The electromagnet program (right) is similarly launched by a driver file called launch_Controller.m that instantiates Magnet Controller, Turn Monitor, and Turn Controller objects. These cascades occur independently of one another, and no communication between the two programs takes place.

By wrapping launch_Controller in a function that returns the Magnet Controller and Turn Controller, I gained access to those objects in the MultiMTgui_initializeHardware file. After instantiating each object, I assigned them to a data field in the global MultiMTgui_main handles structure. This achieved the simultaneous launching of both programs because now the Turn Controller, Magnet Controller, and Turn Monitor (which listens for turns in the Magnet Controller) point directly to the parent window of the entire program.

Then, in order to get the programs to terminate together, I simply included several delete statements within a try-catch block in MultiMTgui_closeGUI.m. This callback is executed whenever the user attempts to close the parent window. It invokes the proper delete files for the Turn Controller, Turn

Monitor, and Magnet Controller and closes the serial and COM ports that are used to interface with the hardware.

Development of the Current Object

MATLAB configuration files contain multiple variables and values that can be loaded into a program. In the MultiMTgui_initializeVariables file, the configuration file that is loaded into the program determines which motor is used, represented by the Motor Object variable in the global handles structure. This Motor Object wraps two "Axis-Type" objects, which represent the motors in the traditional permanent magnet program. Methods such as setPosition(), setVelocity(), and setAcceleration() allow for the manipulation of the motor and thus the magnetic field. Each Motor Object represents a direction of motion: one for translation and one for rotation. The properties of the C862 Mercury Motor Controller class – called C862class – default to:

```
    properties (SetAccess = private)
        scom = [];
        PORT = '';
        BAUD = 9600;
        connected = false;
        nAxes = 0;
        ConnectedAxes = [];
        Axis=C862Axis.empty;
    end
```

The C862class properties are set through a method which takes a port as an input and attempts to create a serial object. Then, the method scans for Mercury Motor Controllers along the different COM ports in the computer. There are two motors that are connected in the traditional permanent magnet setup; a C862Axis object is instantiated for each one.

C862Axis objects are the interfaces through which most of the software will communicate with the hardware. They have the following default properties:

```
    properties (SetAccess = private)
        AxisType = '';
        IsRotary = false;
        HasLimitSwitch = false;
        initialized = false;
        referenced = false;
        Acceleration = 1;
        Deceleration = 1;
        Velocity = 0;
        Limits = [];
        Position = 0;
        TargetPosition = 0;
        MotorScale = 1; %steps/unit
        units = '';
    end
```

AxisType is a string loaded in from the configuration file that specifies the motor driving a particular plane of motion. For translation, the C862 Motor Controller operates the M-126.PD motor. Rotation is done with the C-150.PD motor. The string that is loaded in determines many of the other properties, such as *IsRotary* – *true* for rotation, *false* for translation – *Limits* – 0 to 25 for translation, -infinity to infinity for rotation, *units* – millimeters for translation, degrees for rotation, and so on. The methods for the C862Axis class are mostly *setters* and *getters.* These are publicly available methods that a programmer can invoke to update and retrieve private class attributes. In the case of the C862Axis class, these methods directly interact with the hardware. For instance, public methods such as setVelocity(val), setPosition(val), and setAcceleration(val) are higher level motor functions that facilitate setting the velocity, position, and acceleration values of a motor.

However, because the electromagnetic tweezers set the magnetic field through the magnitude and vector sum of the input currents rather than the position and orientation of the motor, a new "Axis-Type" class must be developed in order to fully integrate the two programs. This class must perform many of the same functions as a traditional motor class in that it should contain convenient functions to set the magnitude and orientation of the magnetic field.

First, I created a new configuration file – MultiMTgui_config_ELECTROMAGNET.mat -- that had many of the same data fields as the present configuration file except that the 'MotorController' field is set to the string value 'ELECTROMAGNET.' Then, in the MultiMTgui_initializeHardware file, I added an extra case to the opening switch statement:

```
switch handles.MotorController
    case 'PI C-843 (PCI)'
        handles.MotorObj = C843class.getInstance();
    case 'PI C-862 (RS232)'
        handles.MotorObj = C862class.getInstance();
        handles.MotorObj.ConnectCOM(handles.MotorCOM);
    case 'ELECTROMAGNET'
        [MC, TM, TC, ~] = launch_Controller;
        handles.MC = MC;
        handles.TM = TM;
        handles.TC = TC;
        disp("Electromagnetic Set Up is On Its Way!");
        handles.MotorObj = C862class.getInstance();
        handles.MotorObj.ConnectCOM(handles.MotorCOM);
        handles.CurrentObj = ElectromagnetClass(handles);
    otherwise
        error('%s is not a valid motor controller',handles.MotorController);
end
```

When the Motor Controller is initialized to 'ELECTROMAGNET' in MultiMTgui_initializeVariables, an ElectromagnetClass object is instantiated as a *Current* object in the program. The C862 motor must still be instantiated, so that the experimentalist can vertically translate the motor, because the distance between the four solenoids and the sample still impacts the strength of the magnetic field. By creating instances of both the *Motor* object and the *Current* object, the unified program has access to both the traditional motor controls and the current controls provided by the electromagnet program.

I created the ElectromagnetClass and ElectromagnetAxis classes with structures that mimic those of the C843 and C862 packages. Much like C862class and C843class, the ElectromagnetClass exists primarily to wrap the two ElectromagnetAxis instances. However, because the Magnet Controller, Turn Controller, Turn Monitor, and motor are already instantiated and linked to their COM ports, the

hardware is already connected, and no further initialization is necessary. Thus, the ElectromagnetClass only requires constructor and a single property: an array called Axis. When the ElectromagnetClass is instantiated, the constructor instantiates two *ElectromagnetAxis* objects to represent magnitude and direction of the magnetic field. These two objects are assigned to the observable property Axis where they can be easily accessed. The ElectromagnetClass object is assigned to the *Current* object data field in the global handles structure associated with the main GUI window. Therefore, the *Current* object and the two *ElectromagnetAxis* objects can be easily accessed by any window in the entire program.

The ElectromagnetAxis class has the following properties and methods:

ElectromagnetAxis Properties (Access = private)	
Property (Default Values)	*Purpose*
Limits = [-255, 255]	Limits of current
Velocity = 0	How quickly the magnitude of the magnetic field changes
Ang_Velocity = 0	How quickly the orientation of the magnetic field changes
Step_Duration = 0	Time required to move one step of current; used as input for WaitForOnTarget()
Num_Voltages = 0	For changes in magnitude of magnetic field only; number of currents to step through from start to stop
Num_Turns = 0	For changes in orientation of magnetic field only; number of turns to step through from start to stop

AxType = ‘’	Either ‘z’ or ‘r’; represents whether this object changes the magnitude or direction of the magnetic field. Modifies the functionality of certain methods

Table 1: Private properties of the ElectromagnetAxis class

ElectromagnetAxis Methods (Access = public)	
Method	*Purpose*
setAxisType(int)	Sets AxType property to ‘z’ if input is two, ‘r’ if input is one
setCurrent()	Accesses MagnetController.Controller.Target and sets it equal to the input current
setTurn()	Accesses TurnController.TargetTurns and sets it equal to the input turn. Starts the TurnController.StepTimer
setVelocity()	Sets either Ang_Velocity or Velocity
WaitForOnTarget()	Waits an appropriate amount of time, as calculated by calculateStepDuration(), to approximate a current speed.
CalculateStepDuration()	Determines how long WaitForOnTarget() must wait from the user’s desired speed

Table 2: Public setters for the ElectromagnetAxis class

setAxisType() is called in the constructor of the ElectromagnetAxis class. When it is passed a 1, the AxType property is set to ‘r’; if it is passed 2, then AxType is set to ‘z’. This property transforms much of the functionality of the methods through if/else statements that test the objects AxType.

Adding Speed Parameters

Another issue is that the speed parameter that specifies how quickly the motor moves in the permanent magnet set-up does not have an easily identifiable analogue for the electromagnetic set-up. The C862 and C843 motors can move at a minimum speed of 0.1 mm/s and a maximum speed of 50 mm/s . As they move along this vertical axis, the distance between the sample and the source of the magnetic field increases. Therefore, specifying the speed of the motor is related to how quickly the magnitude of the magnetic field will change. This is important because molecules that are analyzed with magnetic tweezers are often delicate; abrupt changes in the strength of the magnetic field can be jarring and may potentially damage the sample. Similarly, the rotational speed of the motors specifies how quickly the magnetic field is turning.

Electromagnetic tweezers do not use the motors to change the strength nor orientation of the magnetic field; this is accomplished through changing the input currents. However, the input current can essentially be changed instantaneously. Therefore, I created a speed parameter that approximates the continuous changes in magnetic field strength and orientation achieved by the permanent magnet motors.

My approach takes advantage of the mapping of the input currents onto a 0-255 scale. The current can only change by whole numbers, so the smallest possible step size when moving from an initial magnitude to a final magnitude is 1. This means that the number of steps required to move from $||I_i||$ to $\left|\left|I_f\right|\right|$ is simply equal to the difference between the two values. Once the user specifies a speed, the total duration of the transition between the two magnetic field strengths can be calculated from the definition of velocity.

$$t_{tot} = \frac{\Delta\left|\left|I\right|\right|}{v} \quad (6)$$

Finally, the time required to complete a single step is the total duration of the transition divided by the number of steps.

$$t_{step} = \frac{1}{v} \quad (7)$$

These equations, and their rotational analogues, are employed in the private method CalculateStepDuration(). This method is called after the user enters a speed so that the t_{step} value is immediately available to the rest of the program. This value can then be used to simulate smooth transitions in the magnetic field strength and orientation.

These t_{step} values are used in the public method WaitForOnTarget() derived from the function of the same name found in the C862Axis and C843Axis classes. In the permanent magnet program, these functions are invoked immediately after the position (rotational or translational) of the magnets has been specified. This allocates time for the magnets to move to the required position before the execution of the code continues.

If the ElectromagnetAxis type is 'z', then the WaitForOnTarget() method simply utilizes the built-in pause() function to stall the program execution for t_{step}. This means that, when the magnitude of the magnetic field is changed by the user, it will increment or decrement in steps of 1 and pause for a short length of time that is determined by whatever speed the user selected.

WaitForOnTarget() uses an analogous approach to handle rotation when the ElectromagnetAxis type is 'r'. In this case, the x and y components of the initial magnetic field strength must be modulated from their current values to the values that fit a vector with the same magnitude but different angle. This was implemented by applying the following equations in increments of 10° where θ refers to the new orientation of the magnetic field:

(8)

$$B_x = \big|\big||B|\big|\big|\cos(\theta)$$

$$B_y = ||B||\sin(\theta)$$

Using these two equations, the x and y components of the magnetic field can be changed simultaneously to rotate the field while keeping the magnitude constant. WaitForOnTarget(), for objects with AxTypes of 'r', will apply the two preceding equations in 10 degree increments to simulate the smooth rotation of a magnetic field that was possible with the permanent magnet setup.

WaitForOnTarget() implements these ideas with a while loop with a guard condition that asks whether the target –either the Turn Controller property TargetTurns for rotation or the MagnetController.Controller property Target for magnitude -- is sufficiently close to the current value. In the case of changing the magnetic field's magnitude, the current is discretized into whole number values, so that the while loop ends when the target equals the current value. However, in the case of changing the magnetic field's orientation, rounding errors when converting from angles to turns and turns to angles can cause the target and current turn values to not be exactly equal. Therefore, if the difference is less than 1e-5, the rotation is said to be completed. In the body of the while loop, either setTurn() or setCurrent() are appropriately called with each iteration, using Δθ values of 10° or change in current values of 1. Then, the built-in pause function is called with t_{step} as its argument. The result is an approximated speed feature that simulates the continuous changing of the magnetic field found in the permanent magnet set-up.

Integration of Simple measurements

There are three measurement sequences predisposed in the permanent magnet program: the force *vs*. extension curve, the Chapeau curve, and the temporal recording of the XYZ coordinates of the

bead. A force *vs*. extension measurement gradually increases the strength of the magnetic field by gradually moving the motor up and down the microscope optical axis and taking data after every step. Similarly, the Chapeau curve feature turns the magnetic field by some angle and takes data after each step. Finally, the record XYZ experiment just tracks the position of the beads of interest over time. During this experiment, the magnetic field strength and orientation can still be manually operated. All 3 of these experiments are features of the permanent magnet program but not the electromagnet program. However, the record XYZ measurement did not need to be entirely overhauled because it merely records the positions of the beads of interest and plots them against time; there is no required interaction with the Current Object nor the Motor Objects.

Force Extension

In order to incorporate the force *vs*. extension measurement into the unified electromagnetic tweezers program, I first made several edits to the window that the user will interact with.

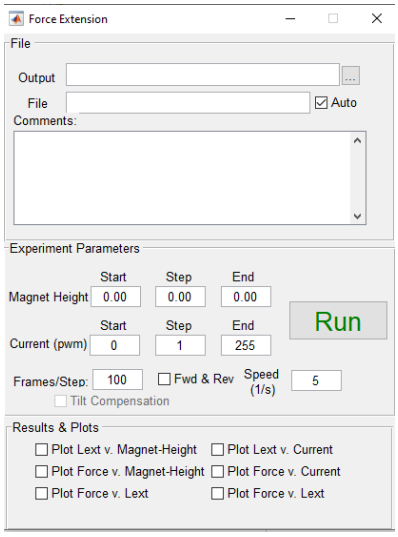


Figure 9: Force extension window through which the user interacts

I added four data fields to the new GUI shown in Figure 9. The first three are parameters of the force extension experiment: the initial current, the step size (in whole numbers), and the final current.

The fourth parameter is the speed which specifies how quickly the current will change. Then, depending on the configuration file that is loaded into MultiMTgui_initializeVariables, some of these data fields are hidden. If the configuration file specifies a motor as the Motor Controller object, then the program defaults to the traditional permanent magnet GUI and the modifications that I made are grayed out. Otherwise, the old data fields (initial height, step size, and final height) are hidden and my modifications are present.

This was achieved through a simple if/else statement in MultiMTgui_setupForceExtension, which executes immediately as the window initializes for the first time. Then, when the user presses 'Start', MultiMTgui_startForceExtension is executed. In this file, the parameters specified by the user are grabbed and saved in the handles to the force extension window.

Finally, I made a parallel file MultiMTgui_FE_ProcessFrame_EM.m that interacts with the ElectromagnetAxis object to step along the array of currents. If an experiment is running, ProcessFrame files will execute immediately after the camera captures an image. The purpose of these files is to record the positions of each bead and perform the appropriate calculations for each experiment. In a force extension experiment, these calculations include the average magnetic force on the bead – given by equation (5) – and the average tether length of the DNA molecule – given by equation (3). In order to do this, MultiMTgui_FE_ProcessFrame_EM initializes persistent variables that have a global scope; they continue to exist even after the function has finished executing. This way, I kept track of the number of images the Piezo has taken as well as the current value in the array of currents that we must step through. After a certain number of frames have been captured – specified by the user – the plots are updated and the index for the array of currents is incremented and the ElectromagnetAxis setCurrent() method is called with the new current value. In order to keep the current from changing too rapidly, WaitForOnTarget() is called to ensure that the magnetic field is changing at the specified rate.

Chapeau Curve

As in the case of the force vs. extension measurement, I began my modifications to the Chapeau curve experiment by changing the GUI to fit both the permanent magnet and electromagnet setups.

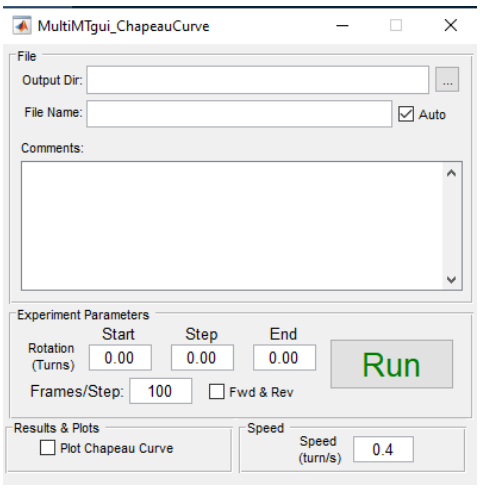


Figure 10: Chapeau curve window that the user interacts with

In the preceding Figure 10, I only needed to add the angular speed data field to the Chapeau curve window. This is because a Chapeau curve experiment is performed in universal units of turns of the magnetic field.

Once again, I created a parallel MultiMTgui_CC_ProcessFrame_EM.m file that records data from each captured frame, performs the necessary calculations, and updates the plots. In this case, we are only interested in how the average height of the bead changes with additional turns, so no further calculations are necessary. Persistent variables are used to keep track of the number of frames captured as well as the number of turns of the magnetic field. After a predetermined number of images have been snapped (specified by the Frames/Step field), the index corresponding to the array of turns is incremented and the ElectromagnetAxis setTurn() method is invoked. Again, WaitForOnTarget() is called to ensure that the orientation of the magnetic field changes at the rate requested by the user.

Object Oriented Design

The modifications that I made to the code also improved the object-oriented structure of the program. Although there was not much of a need to employ the tenets of inheritance and polymorphism, which define how a subclass may derive and implement its methods from parent classes, the edits that I made took advantage of abstraction and encapsulation to further promote the modularity, readability, and reusability of the code. Abstraction and encapsulation refer to the hiding of complicated details from the user by wrapping functionality in methods and applying the appropriate access level.

Much like the existing C843 and C862 packages, the ability to edit the properties of an instance of my ElectromagnetClass and ElectromagnetAxis classes is limited to code within that package itself. These properties, such as velocity, current, AxType, and more, are observable but not editable outside of the ElectromagnetClass directory. As a result, most of the interaction between an ElectromagnetAxis object and the rest of the magnetic tweezer software must utilize the public methods setTurn() and setCurrent(). These methods allow the outside program to easily set the strength and orientation of the magnetic field without worrying about the internal details of how this is achieved. Therefore, this level of visibility is critical because it maximizes encapsulation and abstraction.

The technical details of how the turns and currents are actually being set are compacted into single methods where they can be readily accessed and used. Furthermore, these methods have succinct names that make their purposes and functions easily identifiable to anyone who may be interested in the code in the future.

Robustness

With the edits that I have made to the permanent magnet program, the resulting code has become significantly more robust. The original code was already relatively capable of handling errors

and exceptions, especially those engendered by hardware failures. However, the previous version of the software did not appropriately handle many of the edge cases of the operating parameters. In other words, the program broke down in the instances where the user would attempt to operate the electromagnetic tweezers with either extreme or invalid values.

For example, the Turn Controller does not permit step sizes larger than 0.5 turns. If a user attempted to run an experiment with a step size of 0.6 turns, the program must be able to handle this exception and default to 0.5 turns.

Below is a list of all the edge cases that have been tested and patched to ensure that the code does not run into an error when the user specifies either an extreme or invalid parameter:

Motor Controls

- Minimum motor height below 0 mm
- Maximum motor height above 25 mm
- Motor speed above 50 mm/s
- Motor speed below 0.1 mm/s

Electromagnet Controls

- Current above 255
- Current below -255
- Current non-whole number
- Magnetic field strength above 255 arb units
- Magnetic field strength below 0
- Angle above 360 degrees
- Angle below 0 degrees

- Turn step size above 0.5 turns
- Turn step size below 0.1 turns
- Current Speed above 510 arb units / s[1]
- Current Speed below 1 arb unit / s
- Angular Speed above 100 turns / s
- Angular Speed below 0.01 turns / s

Conclusion

Magnetic tweezers are an important tool for single-molecule experimentation that work by inducing magnetic forces on beads attached to molecules of interest. Electromagnetic tweezers work through similar principles, except that the magnetic field is generated by current carrying wires rather than a pair of permanent magnets. They offer many advantages over the traditional magnetic tweezers' setup, including the lack of physical reverberations due to the translation and rotation of a motor as well as faster adjustments of the magnetic field's strength and orientation. However, Emory University's current electromagnetic tweezer prototype had a glaring software issue: 2 separate, non-communicative programs were required to operate a single instrument. With the modifications made to the permanent magnet GUI, I was able to fully integrate the existing software with the alternative electromagnet software. Now, both programs launch together, can communicate with one another, and close together as well. The user has full access to the particle-tracking and data processing components of the permanent magnet program as well as the magnetic field controls given by the electromagnet program. This was achieved through the development of a Current Object that is modeled off of the existing classes that represent the traditional permanent magnet motors. By interacting with this object,

[1] Derived from the proportion relating top motor speed and motor height to current speed and magnitude of current: $\frac{50\frac{mm}{s}}{25\ mm} = \frac{current\ speed}{255\ arb\ units}$

I made it easier to set the strength and orientation of the magnetic field through methods such as setCurrent() and setTurn(). Consequently, both programs are able to communicate with each other through the properties and methods of this class. The result is a single, unified program that can be used to operate the electromagnetic tweezers in the Finzi-Dunlap lab. What is more, the final program was developed with increased deference towards object-oriented design through encapsulation and abstraction. Finally, after rigorous stress-testing of the code, I concluded that all of the potential edge-cases stemming from the user's input parameters have been accounted for. As such, the code functions as intended and can be used to easily operate Emory's electromagnetic tweezers setup. With a more compact and user-friendly design, it will be significantly easier to utilize the electromagnetic tweezers to study DNA transcription, supercoiling, and topography in the future.

In the future, this software could be further improved by porting the entire program to App Designer, the successor to GUIDE. MATLAB is currently in the process of phasing out GUIDE and any GUIs built through it may no longer be supported. Therefore, it would be prudent to look into mechanisms for migrating programs built with GUIDE over to App Designer. In a similar vein, the program should be version-controlled through Git. During this project, I did not work on a parallel version of the code. This occasionally created some conflicts with other members of the lab who were utilizing the electromagnetic tweezers. With version-control, these situations could be sufficiently minimized. Other areas of improvement include the development of a bead simulation routine – discussed in the appendix – that simulates the calibration process required to interpret the z-position data taken by the tweezers during an experiment. This simulation can be integrated with the main software by prompting the user to either select a simulation or real-world experiment when the program boots up. Then, the simulation could be further advanced by adding the ability to perform simple experiments – such as force extension and Chapeau curve – on the simulated beads, giving new users of the software the ability to play around with the features of the program with mock data. Finally,

yet another prompt should be added that asks the user whether or not they would like to use the electromagnet software or the traditional permanent magnet software. With this prompt, the unified version of the software could be implemented with other magnetic tweezers setups across Emory's labs.

References

Campbell, N. A. (2017). *Biology*. In *Biology* (pp. 335-363). Upper Saddle River: Pearson.

Liu, L. F., & Wang, J. C. (1987). Supercoiling of the DNA template during transcription. *Proceedings of the National Academy of Sciences, 84*(20), 7024-7027. doi:10.1073/pnas.84.20.7024

Finzi, Laura, and Wilma K Olson. "The emerging role of DNA supercoiling as a dynamic player in genomic structure and function." *Biophysical reviews* vol. 8,Suppl 1 (2016)

Daniel T. Kovari, David Dunlap, Eric R. Weeks, and Laura Finzi, "Model-free 3D localization with precision estimates for brightfield-imaged particles," Opt. Express 27, 29875-29895 (2019)

Finzi, L., & Dunlap, D. D. (2010). Single-molecule approaches to probe the Structure, kinetics, and thermodynamics Of Nucleoprotein complexes that Regulate transcription. *Journal of Biological Chemistry, 285*(25), 18973-18978. doi:10.1074/jbc.r109.062612

Charles Dorman, M. D. (2016). DNA supercoiling is a fundamental regulatory principle in the control of bacterial gene expression. *Biophysical Reviews*.

Dorman, C. (2019). DNA Supercoiling and Transcription in Bacteria: A Two-Way Street. *BMC Molecular and Cell Biology*.

Finzi, L. (2020). *Introduction to Magnetic Tweezers.*

Piccolo, J. (2021). Solid State Magnetic Tweezer Development.

Appendix A – Bead Simulation

Due to the challenges posed by online learning and online research, it was often impossible to go into the lab and determine if the hardware was properly responding to the software I have developed. Since I was working remotely for most of the duration of this project, I developed a MATLAB routine to simulate the presence of a bead in a video frame. BeadSimulation.m was developed in close association with Dr. Dunlap in order to confirm the validity of the software.

This bead simulation routine utilizes the sinc function, defined as:

$$sinc(x) = \frac{\sin(\pi x)}{\pi x} \quad (9)$$

This formula approximates the point spread function of the microscope and was used to approximate the image of a "bead" , to calibrate the program. During calibration, a piezoelectric positioner moves the objective along the vertical axis and snaps a number of images at each height. As it does this, the tethered beads go further in or out of focus creating an expanding or contracting ring patterns versus height. A bead without a tether that is fixed to the surface is used as a control to indicate the focal plane of the glass surface.

```
function R = getimg(A, B1, B2, X, Y)
 R = zeros(1088, 2048);
for i = 1:1088
    for j = 1:2048
        bead1 = sqrt((X(i)-544)^2 + (Y(j)-1044)^2);
        bead2 = sqrt((X(i)-272)^2 + (Y(j)-522)^2);
        R(i,j) = A*sin(pi*bead1/B1)/(pi*bead1/B1) + A*sin(pi*bead2/B2)/(pi*bead2/B2);
    end
end
end
```

In the preceding code, X and Y are pixel arrays that span the area of the window that displays the camera's field of view, A is the amplitude of the oscillation of the sinc function, and B1 and B2 are

the periodicities. The argument of the sinc function is the square root of the sum of the squares of each position. This results in an oscillating circular pattern that is visually comparable to the diffraction patterns that appear during an actual calibration. Bead 1 is a typical bead with a DNA tether that goes in and out of focus as the Piezo steps through different heights. This is accomplished by passing the getimg() function a periodicity that scales with the index of the piezo height array. By contrast, B2 is a constant periodicity meant to simulate the control bead that is stuck to surface.

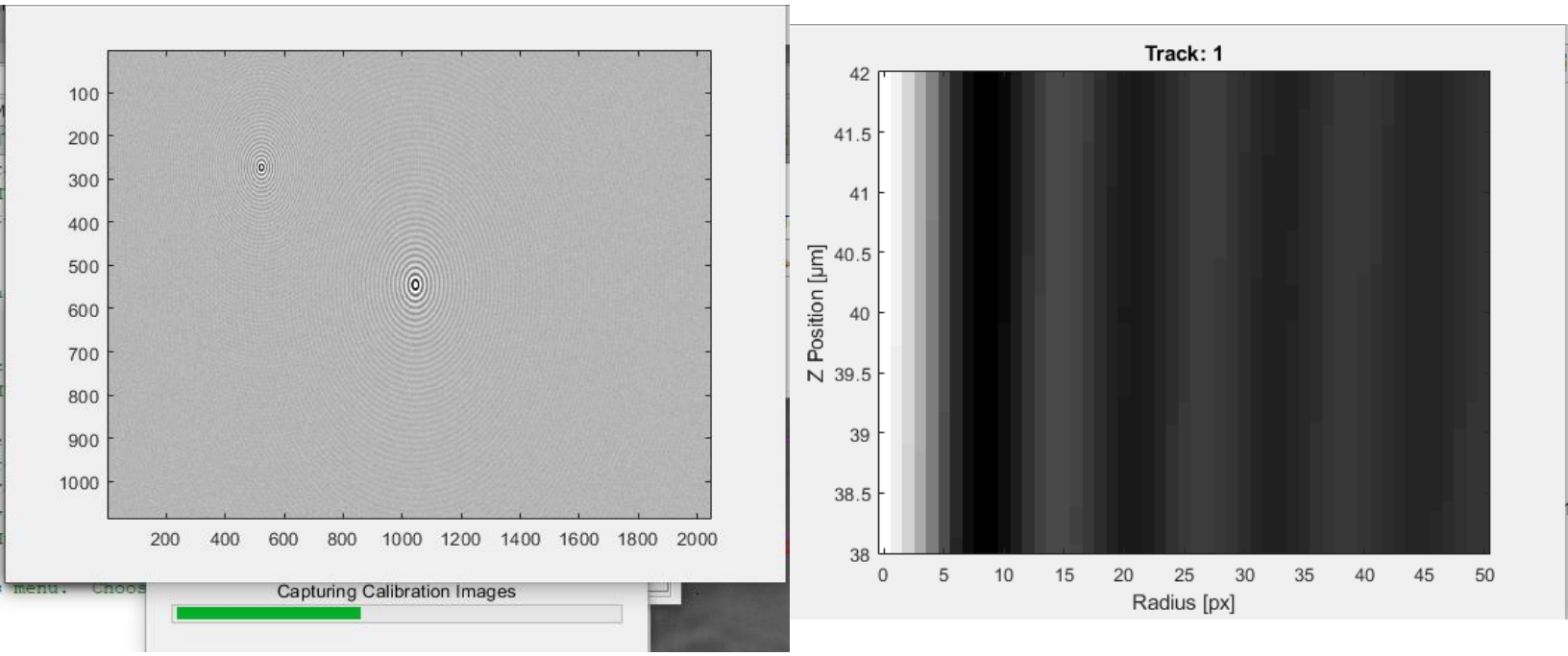


Figure 11: Left: Simulated calibration. The reference "bead" is on the upper left and the measurement "bead" is in the center. Right: A sample diffraction pattern versus displacement scan. In a real experiment, the program uses this data to interpret z values of the measurement beads.

In a legitimate experiment, the data produced by calibration (Figure 11, Right) is utilized to determine how far a bead is out of focus from the camera. Then, the program is able to use this information as a look-up table to provide information about the relative and absolute heights of the beads during data processing.

Appendix B – Relevant Code

ElectromagnetAxis.m

```
classdef ElectromagnetAxis < handle
    properties (Access = private)
        Limits = [];
        Velocity = 0;
        Ang_Velocity = 0;
        Current = 0;
        Turns = 0;
        Step_Duration = 0;
        Num_Voltages = 0;
        Num_Turns = 0;
        AxType = '';
    end

    methods (Access = public)
        function this=ElectromagnetAxis(ax_id) % constructor
            this.setAxisType(ax_id);
        end
        function setAxisType(this, ax_id) %set object to correspond to B
field strength or orientation
            if ax_id == 1
                this.AxType = 'r';
                this.Limits = [-inf, inf];
            elseif ax_id == 2
                this.AxType = 'z';
                this.Limits = [-255, 255];
            end
        end
        function WaitForOnTarget(this, hMain) %used to approximate speed
            handles = guidata(hMain);
            target = this.TargetPosition(hMain);
            if this.AxType == 'r'
                current = handles.TM.Turns;
                while abs(target - current)>= 1e-5 %vals are sufficiently
close
                    current = handles.TM.Turns;
                    pause(handles.TC.StepPeriod);
                end

            elseif this.AxType == 'z' %current already discretized to
smallest possible unit
                pause(this.Step_Duration);
            end
        end

        %setters
        function SetVelocity(this, val)
            if this.AxType == 'r'
                this.Ang_Velocity = val;
            elseif this.AxType == 'z'
                this.Velocity = val;
            end
        end
```

```
        function SetCurrent(this, val, hMain)
            handles = guidata(hMain);
            current = max(this.Limits(1), min(val, this.Limits(2))); %ensure
valid current
            handles.MC.Controller.Target = [current, current];
            this.Current = current;
            pause(this.Step_Duration);

        end
        function SetTurn(this, hMain, turn)
            handles = guidata(hMain);
            handles.TC.TargetTurns = turn;
            this.Turns = turn;
            if ~handles.TC.Running %start TurnController timer
                handles.TC.start();
            else
                return;
            end

        end
        %getters
        function target=TargetPosition(this, hMain) %get target value
            handles = guidata(hMain);
            if this.AxType == 'r'
                target = handles.TC.TargetTurns;
            elseif this.AxType == 'z'
                target = handles.MC.Controller.Target;
            end
        end
        function current = getCurrent(this)
            current = this.Current;
        end
        function turn = getTurns(this)
            turn = this.Turns;
        end
        function velocity = getVelocity(this)
            if strcmpi(this.AxType, 'r')
                velocity = this.Ang_Velocity;

            elseif strcmpi(this.AxType, 'z')
                velocity = this.Velocity;
            end
        end
    end
    methods (Access = public)
        function Step_Duration = CalculateStepDuration(this, varargin)
            %discretize current/turn to smallest possible value and
            %calculate time per step

            if strcmp(this.AxType, 'r')
                Step_Duration = 1 / Ang_Velocity;
                this.Step_Duration = Step_Duration;
            elseif strcmp(this.AxType, 'z')
                Step_Duration = 1 / this.Velocity;
                this.Step_Duration = Step_Duration;
```

```
            end
        end

    end
end
```

ElectromagnetClass.m

```
classdef (Sealed) ElectromagnetClass < handle
    properties
        Axis = []
    end
    methods (Access = public)
        function this=ElectromagnetClass()
            [MC, TM, TC, ~] = launch_Controller;
            handles.MC = MC;
            handles.TM = TM;
            handles.TC = TC;
            disp("Electromagnetic Set Up is On Its Way!");
            handles.MotorObj = C862class.getInstance();
            handles.MotorObj.ConnectCOM(handles.MotorCOM);
            this.Axis = [ElectromagnetAxis(1), ElectromagnetAxis(2)];
        end

    end

end
```

SimulateBead.m

```
function [CalStack, CalStackPos] = SimulateBead(hMain, nPos, nFrames,
CalStack, hBar, CalStackPos)

%% constants
handles = guidata(hMain);
disp(nPos);
pxNoiseSD = 1;
pxNoiseMean = 0;
kT = 4.1;
pN = 2000;
bp = 3000;
cL = 0.34*bp;
extension = 0.7*cL;
pixelCalibration = 7;
fluctuations = sqrt(kT*extension/pN)/pixelCalibration;
x = linspace(0, 1088, 1088);
y = linspace(0, 2048, 2048);
a = 125;
b = 5;
Simulation = figure();
pause('on');
colormap('gray');
colorbar;
```

```
handles.Simulation = Simulation;
ax = Simulation.CurrentAxes;
for p=1:nPos
    drawnow;
    handles = guidata(hMain);
    img = getimg(a, b*(1+(p/nPos)), b, x, y);
    img = rescale(img, 0, 256);

    for n=1:nFrames
        handles = guidata(hMain);
        CalStackPos(n,p) = p;
        tmp = circshift(img,round(fluctuations*randn),1);
        tmp = circshift(tmp,round(fluctuations*randn),2);
        CalStack{n,p} = tmp + pxNoiseSD.*randn(1088,2048) + pxNoiseMean;
        image(ax, CalStack{n,p});
    end
    waitbar((p/nPos), hBar);
end
delete(hBar);
close(Simulation);
guidata(hMain, handles);



end


function R = getimg(A, B1, B2, X, Y)
R = zeros(1088, 2048);
for i = 1:1088
    for j = 1:2048
        bead1 = sqrt((X(i)-544)^2 + (Y(j)-1044)^2);
        bead2 = sqrt((X(i)-272)^2 + (Y(j)-522)^2);
        R(i,j) = A*sin(pi*bead1/B1)/(pi*bead1/B1) +
A*sin(pi*bead2/B2)/(pi*bead2/B2);
    end
end
end
```

MultiMTgui_FE_ProcessFrame_EM.m

```
function MultiMTgui_FE_ProcessFrame_EM(hMain,X,Y,Z_REL,Z_ABS,dZ,UsingTilt)

handles = guidata(hMain);
fehandles = guidata(handles.hFig_ForceExtension);

if ~handles.ExperimentRunning
    return;
end

%% init persistent vars
persistent CurrentFrame;
persistent CurrentStep;
persistent Xacc;
persistent Yacc;
persistent dZacc;

%errorbar handles for figs
persistent hEBs_LvCurrent;
persistent hEBs_FvCurrent;
persistent hEBs_FvL_EM;

persistent Fx;
persistent avgL;
persistent stdL
persistent FxErr;

if isempty(CurrentFrame)
    CurrentFrame = 1;
end
if isempty(CurrentStep)
    CurrentStep = 1;
end
if isempty(Xacc)
    Xacc = NaN(handles.FE_FrameCount,handles.num_tracks);
end
if isempty(Yacc)
    Yacc = NaN(handles.FE_FrameCount,handles.num_tracks);
end
if isempty(dZacc)
    dZacc = NaN(handles.FE_FrameCount,handles.num_tracks);
end

if isempty(Fx)
    Fx = NaN(handles.FE_NumVoltages,handles.num_tracks);
    avgL  = NaN(handles.FE_NumVoltages,handles.num_tracks);
    stdL  = NaN(handles.FE_NumVoltages,handles.num_tracks);
    FxErr  = NaN(handles.FE_NumVoltages,handles.num_tracks);
end

%% update title
handles.MMcam.haxImageAxes.Title.String = ...
    sprintf('Current (PWM): %0.2f; Frame Count %i/%i',...
                        handles.FE_Voltage(CurrentStep),...
```

```
                        CurrentFrame,...
                        handles.FE_FrameCount);

%% xyz data
Xacc(CurrentFrame,:) = X;
Yacc(CurrentFrame,:) = Y;

%RefID = [handles.track_params.ZRef];
dZacc(CurrentFrame,:) = dZ;%Z_ABS(RefID) - Z_REL;


%% setup record
thisRecord.Date = handles.MMcam.clkImageTime;
thisRecord.Step = CurrentStep;
thisRecord.FrameCount = CurrentFrame;
thisRecord.ObjectivePosition = handles.obj_zpos;
thisRecord.MagnetHeight = handles.mag_zpos;
thisRecord.MagnetRotation = handles.mag_rotpos;

thisRecord.Current = handles.CurrentObj.Axis(2).getCurrent();
thisRecord.Turns = handles.CurrentObj.Axis(2).getTurns();
thisRecord.Velocity = handles.CurrentObj.Axis(2).getVelocity();

thisRecord.X= X;
thisRecord.Y = Y;
thisRecord.Z_REL = Z_REL;
thisRecord.Z_ABS = Z_ABS;
thisRecord.dZ = dZ;
thisRecord.UsingTilt = UsingTilt;


mtdat_writerecord(handles.FE_FileID,handles.FE_Record,thisRecord);
CurrentFrame = CurrentFrame + 1;

if CurrentFrame > handles.FE_FrameCount
    stopCamera(hMain);
    CurrentFrame = 1;
    %captured all frames in this step

    Xacc = Xacc*handles.PxScale;
    Yacc = Yacc*handles.PxScale;
    varX = nanvar(Xacc,0,1);
    %mean shift
    Xacc = bsxfun(@minus,Xacc,nanmean(Xacc,1));
    Yacc = bsxfun(@minus,Yacc,nanmean(Yacc,1));
    %tether length
    L = sqrt(Xacc.^2 + Yacc.^2 + dZacc.^2);
    %stats

    %varY = nanvar(Yacc,0,1);
    avgL(CurrentStep,:) = nanmean(L,1);
    stdL(CurrentStep,:) = nanstd(L,0,1);

    %RefTrks = find(strcmpi('Reference',{handles.track_params.Type}));
```

```
    %avgL(CurrentStep,RefTrks) = NaN;
    %stdL(CurrentStep,RefTrks) = NaN;

    %force
    kBT=1.380648813e-23*(273.15+handles.Temperature)*10^6;
    Fx(CurrentStep,:) = kBT*avgL(CurrentStep,:)./varX*10^12;
    FxErr(CurrentStep,:) = kBT*stdL(CurrentStep,:)./varX*10^12;

    %% Plot Data
    [~,filename,~] = fileparts(handles.FE_File);
    MeasTrks = find(strcmpi('Measurement',{handles.track_params.Type}));
    MeasTrkNames = cell_sprintf('Trk %d',MeasTrks);
    nMeas = numel(MeasTrks);
    if fehandles.hChk_FE_PlotLvCurrent.Value
        if isempty(hEBs_LvCurrent) || any(~isvalid(hEBs_LvCurrent)) ||
numel(hEBs_LvCurrent)~=nMeas
            try
                delete(hEBs_LvCurrent);
            catch
            end
            [hEBs_LvCurrent,hAx,~,hFig] = errorbar_selectable(...
                repmat(reshape(handles.FE_Voltage,[],1),1,nMeas),...
                avgL(:,MeasTrks),...
                [],[],...
                stdL(:,MeasTrks),stdL(:,MeasTrks),...
                MeasTrkNames);
            hAx.Title.String = 'Length vs Current';
            xlabel(hAx,'Current [PWM]');
            ylabel(hAx,'Avg. Tether Length [µm]');
            hFig.Name = [filename,' Length v Current'];
            hFig.NumberTitle = 'off';
            hold(hAx, 'on');
        else
            for n=1:nMeas
                hEBs_LvCurrent(n).YData = avgL(:,MeasTrks(n));
                hEBs_LvCurrent(n).XData = reshape(handles.FE_Voltage,[],1);
                hEBs_LvCurrent(n).YLowerData = stdL(:,MeasTrks(n));
                hEBs_LvCurrent(n).YUpperData = stdL(:,MeasTrks(n));
            end
        end
    end
    if fehandles.hChk_FE_PlotFvCurrent.Value
        if isempty(hEBs_FvCurrent) || any(~isvalid(hEBs_FvCurrent)) ||
numel(hEBs_FvCurrent)~=nMeas
            try
                delete(hEBs_FvCurrent);
            catch
            end
            [hEBs_FvCurrent,hAx,~,hFig] = errorbar_selectable(...
                repmat(reshape(handles.FE_Voltage,[],1),1,nMeas),...
                Fx(:,MeasTrks),...
                [],[],...
                FxErr(:,MeasTrks),FxErr(:,MeasTrks),...
                MeasTrkNames);
            hAx.Title.String = 'Force vs Current';
            xlabel(hAx,'Current [PWM]');
```

```
            ylabel(hAx,'Force [pN]');
            %set(hAx,'yscale','log');
            hFig.Name = [filename,' Force v Current'];
            hFig.NumberTitle = 'off';
            hold(hAx, 'on');
        else
            for n=1:numel(hEBs_FvCurrent)
                hEBs_FvCurrent(n).YData = Fx(:,MeasTrks(n));
                hEBs_FvCurrent(n).XData = reshape(handles.FE_Voltage,[],1);
                hEBs_FvCurrent(n).YLowerData = FxErr(:,MeasTrks(n));
                hEBs_FvCurrent(n).YUpperData = FxErr(:,MeasTrks(n));
            end
        end
    end
    if fehandles.hChk_FE_PlotFvL_EM.Value
        if isempty(hEBs_FvL_EM) || any(~isvalid(hEBs_FvL_EM)) ||
numel(hEBs_FvL_EM)~=numel(MeasTrks)
            try
                delete(hEBs_FvL_EM);
            catch
            end
            [hEBs_FvL_EM,~,~,hFig] = ForceExtension_selectable(...
                    avgL(:,MeasTrks),...
                    Fx(:,MeasTrks),...
                    stdL(:,MeasTrks),...
                    FxErr(:,MeasTrks),...
                    MeasTrkNames);
            hFig.Name = [filename,' Force v Length'];
            hFig.NumberTitle = 'off';
            hold(hAx, 'on');
        else
            for n=1:numel(hEBs_FvL_EM)
                hEBs_FvL_EM(n).XData = avgL(:,MeasTrks(n));
                hEBs_FvL_EM(n).YData = Fx(:,MeasTrks(n));
                hEBs_FvL_EM(n).XLowerData = stdL(:,MeasTrks(n));
                hEBs_FvL_EM(n).XUpperData = stdL(:,MeasTrks(n));
                hEBs_FvL_EM(n).YLowerData = FxErr(:,MeasTrks(n));
                hEBs_FvL_EM(n).YUpperData = FxErr(:,MeasTrks(n));
            end
        end
    end


    %increment step
    CurrentStep= CurrentStep +1;
    if CurrentStep >handles.FE_NumVoltages
        MultiMTgui_stopForceExtension(hMain);
    else
        if handles.FE_Voltage(CurrentStep) ~=
handles.CurrentObj.Axis(handles.magzaxis).TargetPosition(hMain)
            if handles.FE_Voltage(CurrentStep) >
handles.FE_Voltage(CurrentStep-1)
                for
i=handles.MC.Controller.Target+1:handles.FE_Voltage(CurrentStep)
                    handles.CurrentObj.Axis(handles.magzaxis).SetCurrent(i,
hMain);
```

```
                end
            elseif handles.FE_Voltage(CurrentStep) <
handles.FE_Voltage(CurrentStep-1)
                for i=handles.MC.Controller.Target-1:-
1:handles.FE_Voltage(CurrentStep)
                    handles.CurrentObj.Axis(handles.magzaxis).SetCurrent(i,
hMain);
                end
            end
        end
    end
    startCamera(hMain);
end

end
```