# Remote Access and Automation at BioSAXS

*Sam Barton*[1] *and Richard Gillilan*[2]

*1 Math and Engineering Science Department, Hudson Valley Community College, Troy, New York*

*2 Cornell Laboratory for Accelerator-Based Sciences and Education, Cornell University, Ithaca, New York*

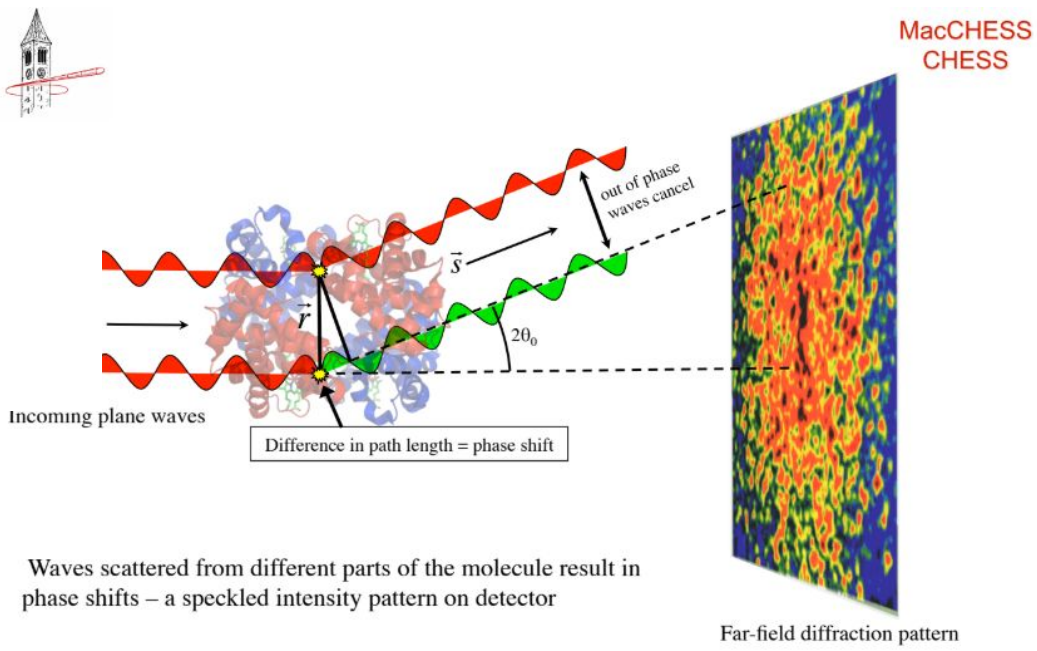
**Abstract:**

At the time of writing, as our country experiences the Covid-19 pandemic, we as a society have come together to prevent the spread and minimize the health impact of the virus: stores and workplaces are closed, masks are required, and social distancing is in full effect. Cornell's Laboratory for Accelerator-based Sciences and Education (CLASSE) is no exemption to this; thus, at the BioSAXS station within CLASSE's synchrotron radiation laboratory (CHESS), we have worked towards finding new and innovative ways to allow for the continued conduction of research while the laboratory remains largely closed. Our efforts have been focused on developing a remote access model, in which users mail in samples and control the station from their home computers. In order to set up an effective remote access system, our efforts were focused on testing and optimizing different remote desktop programs, setting up cameras to view important components of the station, and modifying the existing software to allow for greater automation.

**Introduction:**

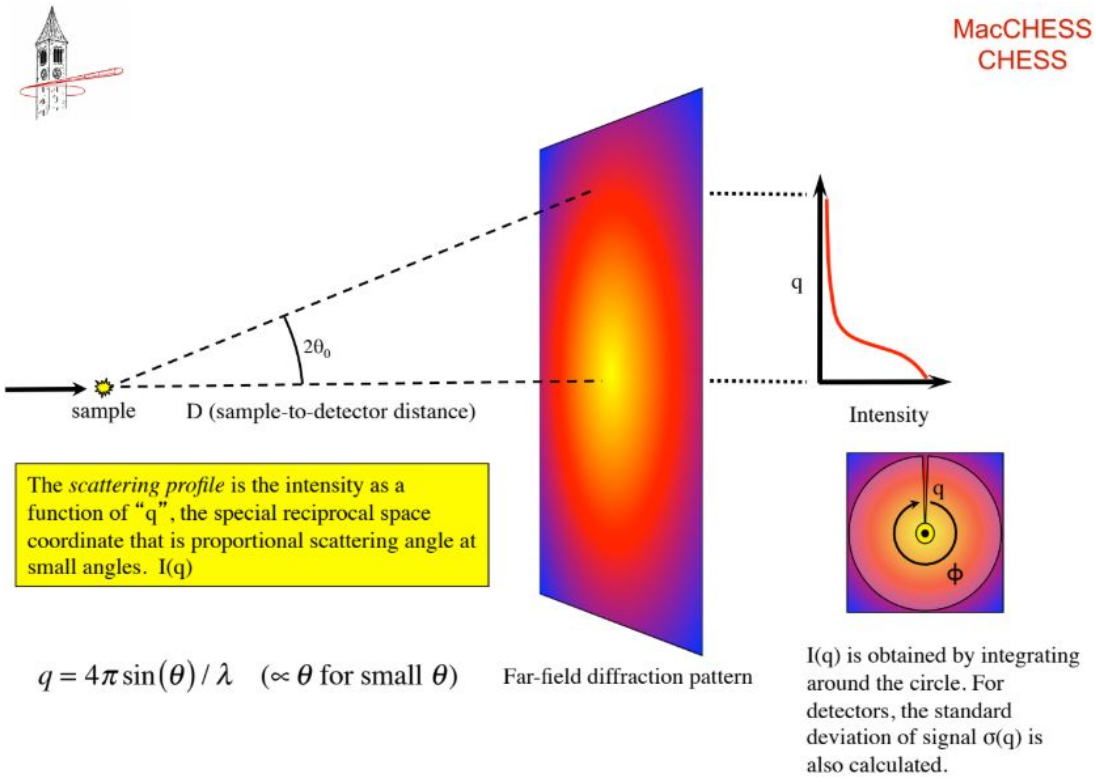
BioSAXS stands for biological small-angle x-ray scattering, and it's the method used at the station to study the structures of biomolecules. At Cornell's High Energy Synchrotron Source (CHESS), positrons are accelerated in order to produce high energy x-ray beams. These beams are utilized at multiple different stations, such as BioSAXS, where they're shone through samples containing a specific biomolecule. The electrons within a biomolecule interact with the x-ray waves that make up the beam, causing elastic scattering. These scattered waves interfere with each other, creating a diffraction pattern, as seen below in Fig. (1). Since different biomolecules have different structures and consequently different electron positions, every biomolecule creates a unique diffraction pattern.

Figure 1: X-ray Diffraction of a Single Biomolecule



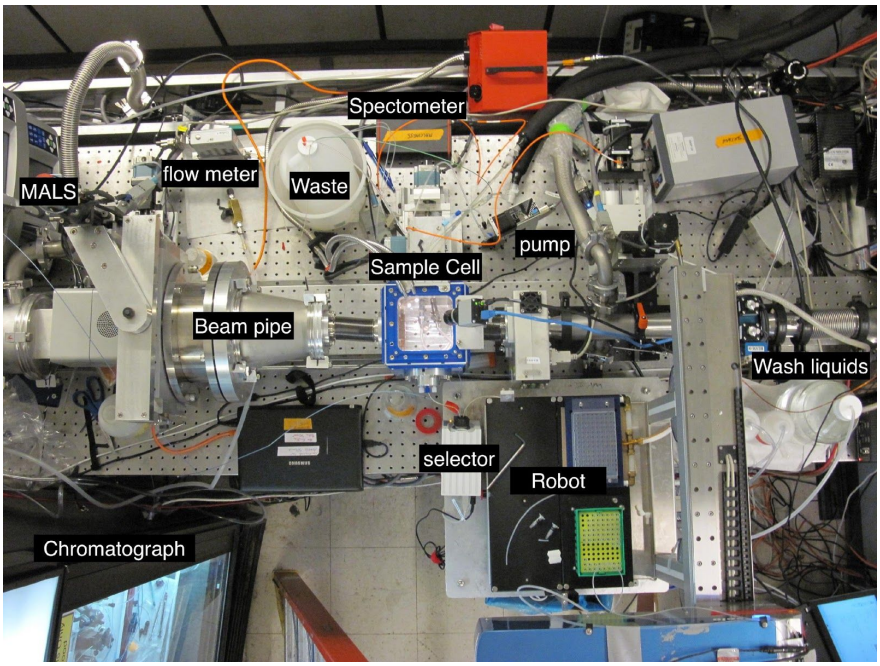

For a dilute sample including many biomolecules of a single type (which is more realistically seen in the laboratory since it's impossible to isolate a single biomolecule), a different diffraction pattern is obtained, which can be seen below in Fig. (2). This pattern is equivalent to the scattering pattern of a single rotationally-averaged biomolecule of the same type. A scattering profile is obtained from this pattern, as seen below in Fig. (2). This profile is used to generate Guinier and Kratky plots, which are used for further structural analysis [1].

Figure 2: Scattering Profile of a Rotationally-Averaged Biomolecule



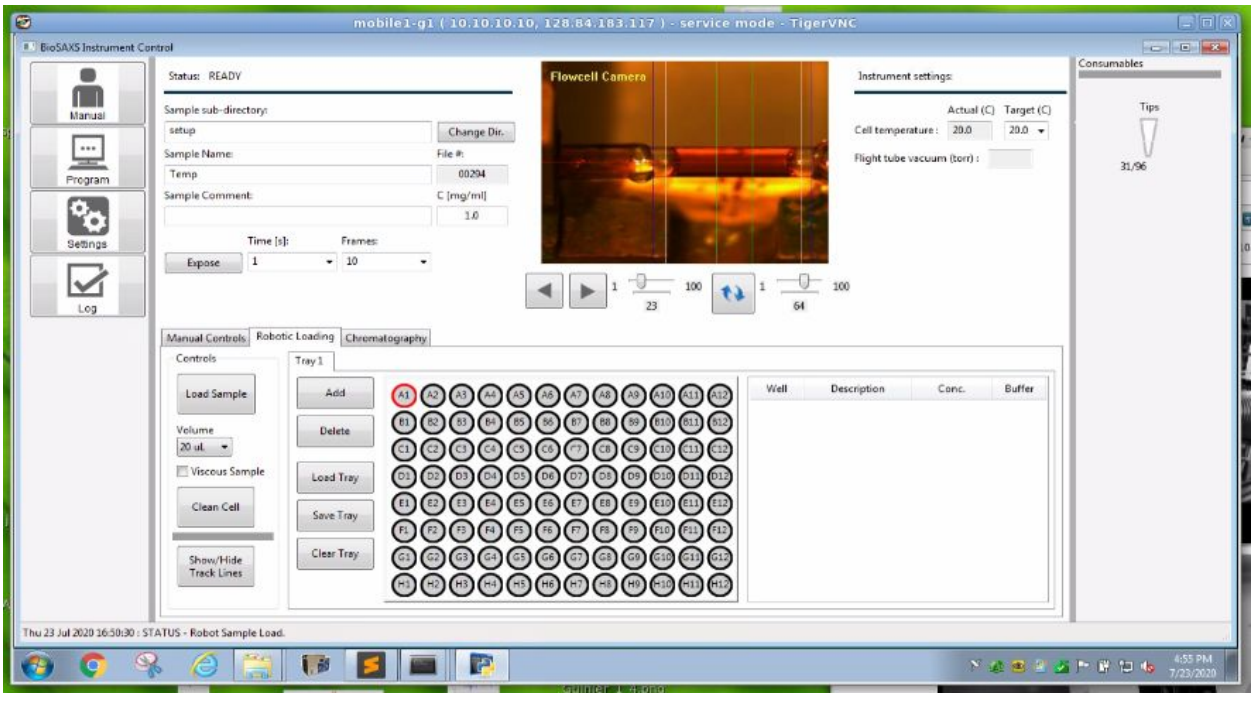

The BioSAXS station relies on a combination of hardware and software components working together in order to make data collection efficient and accurate. The hardware setup for the station can be seen below in Fig. (3). First, samples are loaded into a tray by the user. In the case of remote access, samples will be mailed in, and a station scientist/technician will load them into the tray. For each trial, a robot arm loads a new pipette, pumps the contents of a tray cell into the pipette, and pumps the sample into a funnel. This funnel is connected to capillary tubing; tubing that eventually crosses the x-ray beam perpendicularly. A syringe pump is used to move the sample into the beamline, where the sample is then oscillated (in order to prevent x-ray damage) and diffraction scattering data is collected using a detector set up behind the sample. After data is collected, a valve switches the tubing connection from the funnel to a peristaltic pump that's connected to cleaning solutions and a dryer. The peristaltic pump is then used to run a cleaning cycle so that the capillary tubing is clean and dry. This process is then repeated for every cell within the tray.

Figure 3: Hardware Setup of BioSAXS



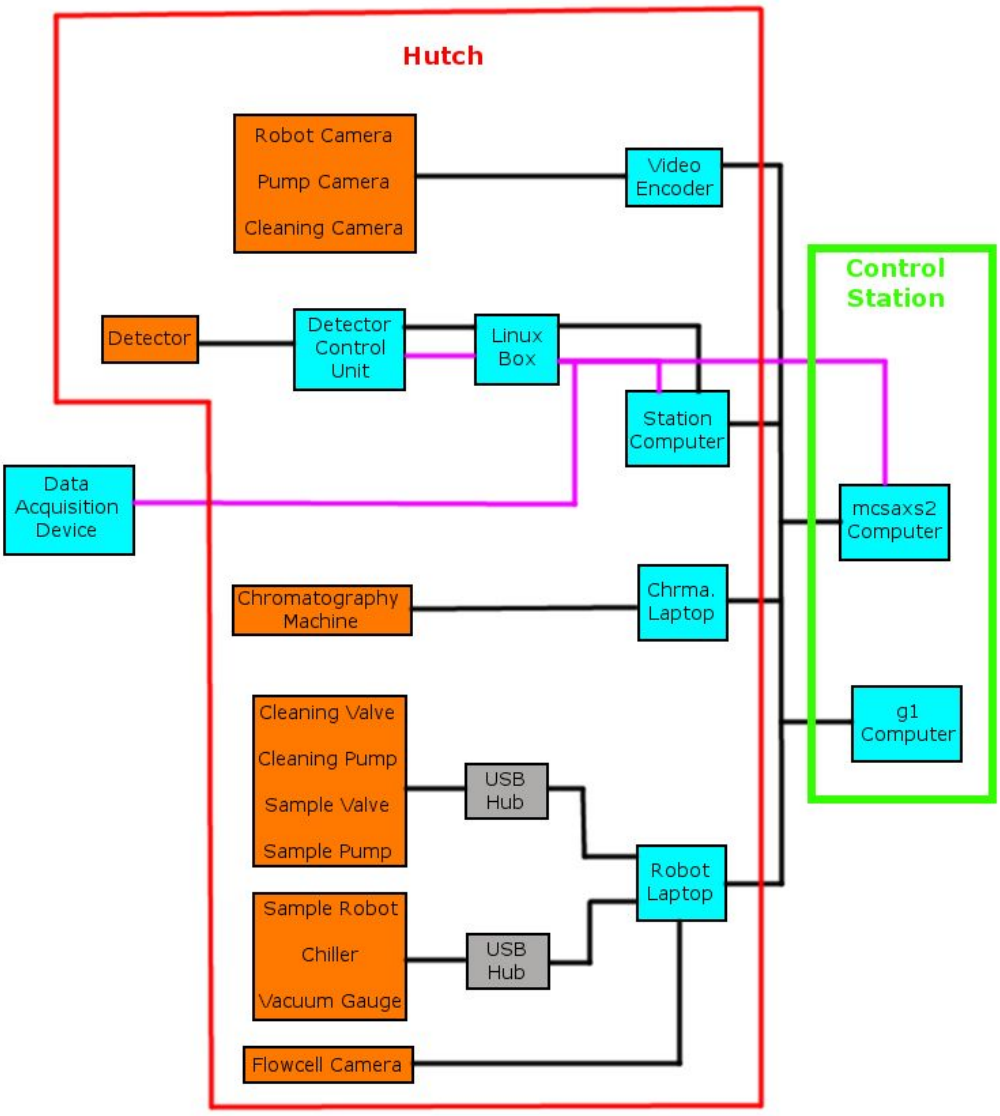

In order to tie all the hardware components of the station together, multiple pieces of software are used. Arguably, the most important piece of software is biosaxs.py, which creates the graphics user interface (GUI) of the station, which allows users to control all the hardware components on a single desktop (mcsaxs2). In Fig. (4) below, a screenshot of the biosaxs.py GUI can be seen.

Figure 4: BioSAXS GUI



There are also conn.py, pumpserver.py, and v2l_server.py which deal with hardware communication, the syringe pump, and video tracking, respectively. All of these programs, including biosaxs.py, run on a laptop within the hutch. This laptop is connected to all the hardware components, and it's accessed outside the hutch by the station computer (called mcsaxs2) through the use of a program called VNCViewer. A diagram of the network setup of the station can be seen below in Fig. (5).

Figure 5: BioSAXS Network Diagram



**Remote Desktop Software:**

As aforementioned, in order to set up an effective remote access system, a good remote desktop program is key. Remote desktop programs allow users to connect and control other computers from their own. In the case of the BioSAXS station, users need to connect to mcsaxs2, where from there, they can connect to the robot laptop and control the station through the use of the biosaxs.py GUI. We needed a program that would accomplish this at a high frame rate and resolution; otherwise, remote access would be unusable for users. We found three programs called X2Go, NoMachine, and ConnectWise, and we undertook extensive testing in order to determine which program would be the best option for the station. On the next page is a pros-and-cons list for each program transferred from Sam Barton's laboratory notebook.

X2GO:

- Pros:
    - Open-source
- Cons:
    - Slow frame rate
    - Does not scale resolution automatically
    - Includes very few options and features
    - Less user-friendly
    - Outdated GUI
    - Requires the use of a VPN, which means additional set up for users

NoMachine:

- Pros:
    - Very high frame rate
    - Scales resolution properly
    - Includes the most display options
- Cons:
    - Less user-friendly
    - Requires the use of a VPN, which means additional set up for users

ConnectWise:

- Pros:
    - Best resolution scaling
    - Most user-friendly
    - Includes the most features (screen sharing, mic capabilities, drawing)
    - Cleanest looking GUI
    - Does not require the use of a VPN; barely any user set up required
- Cons:
    - Runs laggier than NoMachine
    - Often glitches creating two mouse cursors
    - Not as many display options as NoMachine

After our testing, we determined that NoMachine would be the best program to use. X2Go had lots of problems, so it wasn't even competitive with NoMachine and ConnectWise. ConnectWise was by far the nicest looking program, and its user friendliness and abundance of features made it a very viable option. Ultimately, we decided on NoMachine because it was the best in terms of display options and performance. Nonetheless, X2Go and ConnectWise remain on mcsaxs2, so users can still use these programs if they choose to. Extensive documentation was written up that guides users step by step through the process of connecting to the station from their home

computer for all three programs. This makes it extremely easy for new users to connect to BioSAXS without prior knowledge of computers and how the station's network is set up.

**Remote Camera System:**

After setting up an effective remote desktop program, we focused on installing and connecting cameras in the station. Under normal circumstances, users in the lab are able to see cleaning/waste liquid levels, valves, pumps, the capillary tubing, etc. In transitioning to a remote access model, users still need to be able to see these important components, so that they can diagnose problems in real time and keep track of material levels. Our solution to this problem was to install multiple cameras within the hutch. These cameras are connected to a video encoder, which is connected to the CHESS public network as seen in Fig. (5) above. As seen below in Fig. (6), cameras were set up facing the cleaning liquid levels and syringe pump as well as on the loading robot arm.

Figure 6: Remote Camera System


The robot loading arm camera is especially important, since it allows users to see if samples are loaded into the funnel properly. It's also important to mention that a camera facing the beam exposure area of the capillary tubing was already in use prior to station modifications (the view of this camera can be seen above in the GUI in Fig. (4)). This camera is especially important for remote access, since it allows users to guide samples into place to perform proper data collection.

The flow cell camera is also very important for video tracking and automation; its importance is outlined in the next section.

**Automation and Video Tracking:**

After installing and connecting various cameras around the station, we worked on modifying the existing code to allow for greater automation. By automating components and processes at the station, remote control becomes significantly easier, since users need less real-time information; information that sometimes can only be obtained by being at the station in-person. In addition, increased automation makes the station easier to run for new users. In the past, when new users have operated the station, they've been given an in-person tour and shown demonstrations. Now that things are being operated remotely for the time-being, it's much harder to show new users the ropes of the station. By automating some of the station's processes, the learning curve is lessened. Automation also has applications outside of the remote access model. Once things are back in-person, the automatic systems can continue to be used, freeing up operating time for users and reducing the number of user errors present at the station.

The first and most important order of action was fixing the video tracking software already present at the station. Prior to modification, samples were loaded into the tubing using either the sample robot or chromatography system. These samples were then positioned in the beamline through the use of manual pump movement buttons in the GUI.

This process was very inefficient and is difficult for remote users to use, since they can't check the position of the sample within the tubing at any given time. In order to make things easier, we decided upon implementing an automatic loading system. This system stops the sample automatically near the beamline. Code already existed to undertake this process, but it was very buggy and unreliable at tracking plug samples. In addition, the length of the tubing had been increased at an earlier date, and upon the command of a single load cycle, the sample wasn't pumped far enough to reach the beamline. This is important because the video tracking software stops the sample as soon as the camera sees it. It does not add additional pump load steps, so if the plug doesn't reach the beamline, the automatic stopping mechanism doesn't work.

The maximum number of load steps for the syringe pump was increased from 3000 to 4500, since it takes approximately 3500 steps for the sample to reach the beamline (a max value of 4500 ensures the plug will always move past the beamline, even for viscous materials). This created a new problem. The syringe pump could not accept a single load of higher than 3000 steps from its initial position (less than 3000 if it's farther away from its initial position). Thus, the code seen in Fig. (7) was added to the onLoadSampleButton() method within biosaxs.py, which is called when the 'Load Sample' button in the GUI is clicked.

Figure 7: Loading Fix Code (onLoadSampleButton() within biosaxs.py)

```
comm.controlQueue.put([('B', 'INIT')])

if steps >= 3000:
    steps -= 3000
    comm.controlQueue.put([('B', 'LOAD ' + str(3000) + ' ' + str(speed))])
    comm.controlQueue.put([('B', 'LOAD ' + str(steps) + ' ' + str(speed))])
else:
    comm.controlQueue.put([('B', 'LOAD ' + str(steps) + ' ' + str(speed))])
```

After fixing the loading issue, we modified the video tracking code to make it more accurate and reliable. Multiple lines of redundant and overly complex code were removed, leaving the following code within the trackSample() method within V2L_Server.py.

Figure 8: Video Tracking Code (trackSample() within V2L_Server.py)

```
xl = min(np.trim_zeros(over_threshold))
xr = max(np.trim_zeros(over_threshold))

if xr-xl > self.parameters['MIN_TRACKSIZE']:
    self.right_trackpos = xr
    self.left_trackpos = xl

    if self.show_sample_tracking_lines:

        if self.left_trackpos != 0:
            #Draw tracking line
            pt1 = ( self.left_trackpos, 0 )
            pt2 = ( self.left_trackpos, self.frame_height )
            cv2.line( frame, pt1, pt2, color = (255, 255, 255) )

        if self.right_trackpos != 0:
            #Draw tracking line
            pt1 = ( self.right_trackpos, 0 )
            pt2 = ( self.right_trackpos, self.frame_height )
            cv2.line( frame, pt1, pt2, color = (0, 0, 0) )
```

Once video tracking was fixed, we moved on to further station automation. Every time a new video frame is processed, the above code in Fig. (8) is run, so that the tracking position is always being updated. In addition, the following code (Fig. (9)) within the loadAndStopSample() method is also run every time a new frame is processed, as long as the automatic load stop function is activated (this is done by checking a box within the GUI).

Figure 9a: Oscillation and Exposure (loadAndStopSample() within V2L_Server.py)

```
if self.first_time:
    if self.left_trackpos < self.parameters['ROI_RL'] and self.left_trackpos != 0:
        self.pump_socket.send("STOP")
        self.pump_socket.recv()
        self.time1 = time.time()
        self.count = False
        print "Move left"
        time.sleep(0.5)
        bio_socket.send("Expose")
        bio_socket.recv()
        self.pump_socket.send("RELMOVE 400 200")
        self.pump_socket.recv()
        self.first_time = False
        self.second_time = True
elif self.second_time:
    self.time2 = time.time()
    if (self.time2 - self.time1) >= 4:
        # bio_socket.send("Start Exposure")
        # bio_socket.recv()
        self.pump_socket.send("STOP")
        self.pump_socket.recv()
        time.sleep(0.5)
        self.pump_socket.send("RELMOVE -600 200")
        self.pump_socket.recv()
        self.time2 = time.time()
        print "Move right"
        self.second_time = False
        self.third_time = True
```

Figure 9b: Oscillation and Exposure (loadAndStopSample() within V2L_Server.py)

```
elif self.third_time:
    self.time3 = time.time()
    if (self.time3 - self.time2) >= 4:
        self.pump_socket.send("STOP")
        self.pump_socket.recv()
        time.sleep(0.5)
        self.pump_socket.send("RELMOVE 400 200")
        self.pump_socket.recv()
        self.time3 = time.time()
        print "Move left"
        self.third_time = False
        self.fourth_time = True
elif self.fourth_time:
    self.time4 = time.time()
    if (self.time4 - self.time3) >= 4:
        self.pump_socket.send("STOP")
        self.pump_socket.recv()
        time.sleep(0.5)
        self.pump_socket.send("RELMOVE -600 200")
        self.pump_socket.recv()
        self.time4 = time.time()
        print "Move right"
        self.fourth_time = False
        self.fifth_time = True
elif self.fifth_time:
    self.time5 = time.time()
    if (self.time5 - self.time4) >= 4:
        self.pump_socket.send("STOP")
        self.pump_socket.recv()
        bio_socket.send("Clean")
        bio_socket.recv()
        self.first_time = True
        self.fifth_time = False
        self.load_and_stop = False
```

This code within V2L_Server.py is responsible for stopping the sample once it's in view of the flow cell camera. Afterwards, the sample is oscillated in the beamline, and the station's detector starts taking exposures. Oscillation is important so the sample does not become overly damaged by the x-ray beam. After the detector is done taking exposures, oscillation is stopped, and the cleaning cycle of the system is run. Within V2L_Server.py, commands can be sent directly to the syringe pump, but not to the detector or cleaning system: biosaxs.py is responsible for this communication. Thus, a server/client communication was established between the two programs. The server and client code can be seen below in Figs. (10) & (11).

Figure 10: Server (beginning of V2L_Server.py)

```
context = zmq.Context()
bio_socket = context.socket(zmq.REP)
bio_socket.bind("tcp://*:1337")
bio_socket.recv()
```

Figure 11: Client (beginning of biosaxs.py)

```
context = zmq.Context()
bio_socket = context.socket(zmq.REQ)
bio_socket.connect("tcp://localhost:1337")
bio_socket.send(" ")
```

The following code within the Tracking() thread of biosaxs.py was then written.

Figure 12: Exposure and Cleaning Communication (Tracking() class within biosaxs.py)

```
class Tracking(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        #start exposure (timed so no need to have stop command)
        bio_socket.recv()
        bio_socket.send("Got")
        print "Exposure started."
        ExposureControls.takeExposure(ExposureControls)
        #run cleaning cycle
        bio_socket.recv()
        bio_socket.send("Got")
        print "Cleaning cycle initiated."
        mainframe = wx.FindWindowByName('MainFrame')
        comm.controlQueue.put([('D', ['CLEAN', mainframe.all_settings])])
        pub.sendMessage('log.update', data = ['STATUS', 'Cleaning started.'])
```

This code essentially sits and waits for commands from the code in Fig. (9). When sample oscillation is begun, a command is sent from loadAndStopSample() within V2L_Server.py to Tracking() within biosaxs.py to turn on the detector. This process also takes place when the cleaning system is initiated.

Prior to code modification, loading, oscillating, exposing, and cleaning were all handled by different buttons in the GUI. Users would have to monitor the station and click the correct buttons at the correct times. Now, users only have to press one button, and the code takes care of the rest. Additional information about the modified code is included within the software in the form of comments.

**References:**

[1] Kikhney, Alexey G. and Svergun, Dmitri I. (2015), A practical guide to small angle X-ray scattering (SAXS) of flexible and intrinsically disordered proteins, *FEBS Letters*, 589, doi: 10.1016/j.febslet.2015.08.027

**Acknowledgement:**
Special thanks to Dave Schuller, Keara Soloway, Robert Miller, Jesse Hopkins, and Soren Skou for their help. This work is based upon research conducted at the Center for High Energy X-ray Sciences (CHEXS), which is supported by the National Science Foundation under award DMR-1829070, and the Macromolecular Diffraction at CHESS (MacCHESS) facility, which is supported by award 1-P30-GM124166-01A1 from the National Institute of General Medical Sciences, National Institutes of Health, and by New York State's Empire State Development Corporation (NYSTAR).