Bernstein Lab

Automated Measurement Control for TECNA Characterization

Joshua Kannappilly

Introduction: The TECNA Device

Developed in the Bernstein Lab, the Thermoelectrically Coupled Nanoantenna (TECNA) is an unbiased, frequency-selective detector designed for long-wave infrared (LWIR) sensing. This device consists of a nanoantenna, which captures Incident IR radiation. The radiation induces high-frequency alternating currents in the antenna, which are dissipated as heat at a tiny junction. A thermocouple, formed by the junction of two dissimilar metals or a single-metal geometry with varying widths, converts this localized temperature rise into a measurable DC open-circuit voltage. The primary characterization requirements for these devices include determining their polarization sensitivity; the antenna should show a $\cos^2(\theta)$ response.

Problem Statement

Before this project, the data acquisition process in the lab had a significant lack of synchronization between motion control and data logging. To characterize polarization, a researcher had to operate the SR50CC rotational stage, which houses the Half-Wave Plate, using the standalone Newport SMC100 Utility GUI. Simultaneously, the researcher had to manually trigger the lock-in amplifier's logging software on a separate system. It was difficult to ensure the lock-in was recording exactly when the stage was at a stable, specific angle. Furthermore, manually moving between two software interfaces to align hundreds of potential data points is inherently prone to error and data misalignment.

Objectives and Plan

The objective of this project was to develop a unified, automated software that would allow the lab to move away from manual characterization. The plan to complete the objective was structured into three phases:

1) Interface with the Newport SMC100 controller using Python.
2) Program the controller to actuate the Half-Wave Plate
3) Record data with the lock-in amplifier during rotation.

Following the implementation of the rotational sweep, the project scope was expanded to include the development of a 2D Raster Scan framework. This framework is discussed in detail in the Raster Scan section of this report.

Initialization

The first major hurdle involved the initialization of the SMC100 controller. I utilized an existing Python 2 library (pySMC100) and updated it for Python 3. However, the reset_and_configure() function, designed to bring the controller into a "ready" state, consistently timed out.

Through the addition of debug statements, I discovered that the controller was entering State ‘10’ after a reset. The library was hard-coded to wait only for State ‘0A’ (Not Referenced from Reset). Research into the Newport manual revealed that State ‘10’ indicates a “NOT REFERENCE ESP stage error.” While it sounds like a failure, it actually indicates the controller has successfully recognized an “Electronic Stage Program” (ESP) stage but hasn’t yet loaded its onboard configuration.

To solve this problem, I modified the library’s wait condition to accept both ‘0A’ and ‘10’ as valid post-reset states. This allowed the configuration sequence to proceed, enabling the controller to load the stage’s specific parameters from its internal memory.

After successfully initializing the SMC100 controller, I was almost fully done with the actuation implementation for the SR50CC rotational stage. A critical step in establishing precision control was mapping the motion controller’s internal logic to the physical resolution of the stage. The Newport SMC100 is inherently unit-agnostic; it executes movements based on discrete encoder counts. I configured the controller’s scaling factor to match the SR50CC stage’s native hardware resolution of 0.001 degrees per encoder count, which is shown in Figure 1.

By defining the user unit within the driver, I effectively translated the controller’s raw counts into degrees. The calibration ensures that a software command to move 1 unit corresponds to a 1-degree physical rotation.

DC-Motor Performance Specifications and Characteristics

	Resolution (°)	Speed (°/s)	Nominal Voltage (V)	Max RMS Current (A)	Max. Peak Current (A)	Resistance (Ω)	Inductance (mH)	Tachometer Const. (V/krpm)
SR50CC	0.001	4	12	0.075	0.15	117	0.95	–

Figure 1. Operating Specifications for the SR50CC Stage

QCoDeS Integration

For the data acquisition phase of the project, I was tasked with integrating the customized SMC100 driver into the lab’s existing measurement framework: QCoDeS. The framework is designed to create a standardized way to communicate with various lab instruments, regardless of the manufacturer, by treating hardware components as class objects known as Parameters.

In QCoDeS, a parameter is a class that represents a physical quantity you want to either set (like a motor position) or measure (like voltage). The framework acts as the primary backend for hardware abstraction, ensuring all data points are recorded into a centralized SQL database.

To integrate the motion control into the lab's existing workflow, I created a custom QCoDeS class: RotationAngle(Parameter). The class inherits from the standard QCoDeS Parameter class and acts as the interface for the Half-Wave Plate. The core of this parameter consists of two vital functions: get_raw() and set_raw(). The get_raw() function is linked to the driver's get_position_deg() method. Whenever QCoDeS needs to know where the motor is, it automatically calls get_raw(). The set_raw() function is linked to the driver's move_relative_deg() method. During a measurement sweep, QCoDeS automatically calls this function to move the stage to the next coordinate. I also created an additional function to set the speed, which I manually invoke after instantiating the object. Figure 2 shows the implementation of the class.

```python
class RotationAngle(Parameter):
    def __init__(self, name, stage_controller = controller):
        self.stage = stage_controller
        Parameter.__init__(self, name = name, label = "Rotational Angle")

    def get_raw(self):
        return self.stage.get_position_deg()

    def set_raw(self, angle):
        self.stage.move_relative_deg(angle, waitStop=True)

    def set_speed(self, speed):
        self.stage.sendcmd('VA', speed)
```

Figure 2. Implementation of the RotationAngle parameter

Once the RotationAngle parameter was defined, I integrated it into the measurement environment in our lab. This environment utilizes a module called Measuring, developed by Dr. Orlov. Our data acquisition scripts inherit the functionality of this module to simplify the execution of experimental runs.

To implement the polarization characterization, we first instantiate the RotationAngle object. We then use the Measuring module's functions to define the experimental space:

meas.add_measurand(self, *meas): This uses Python's *args syntax to allow for a flexible number of dependent variables. In our configuration, we pass in the lock-in amplifier's X, Y, and R parameters.

meas.add_variable(self, Var, Vmin, Vmax, Numb, delay): This defines the independent variable of the experiment. We pass our custom rotation_angle object and specify a start position, end position, total number of points, and a delay. The delay is the pause (in seconds) between the completion of a motor step and the recording of the voltage measurement. This ensures stability in readings before data is committed to the database. The motor will only continue to move after this measurement is logged.

meas.run(): This final command triggers the automated sequence.

```
time.sleep(1.0)
meas = Measuring('5sdelay measurand1')

meas.add_measurand(X, Y, R)
meas.add_variable(rotation_angle, zero_pos, zero_pos + 90, 401, 3, 0)
meas.run()
```

Figure 3. Python Implementation of the Automated Sweep using the Measuring Module

Through this implementation, the Measuring module iterates through the specified angular steps, calling our set_raw function at every increment.

Experimental Validation

To evaluate the performance of the automation script, I executed a series of polarization sweeps from 0 to 90 degrees with a resolution of 201 data points. These measurements were performed using various stabilization delays to investigate the impact of settling time on data quality. Figure 4 shows that the measured voltage consistently exhibits the $\cos^2(\theta)$ dependence across the different delay settings. The result confirms that both the dipole antenna and the automation script are working as expected.

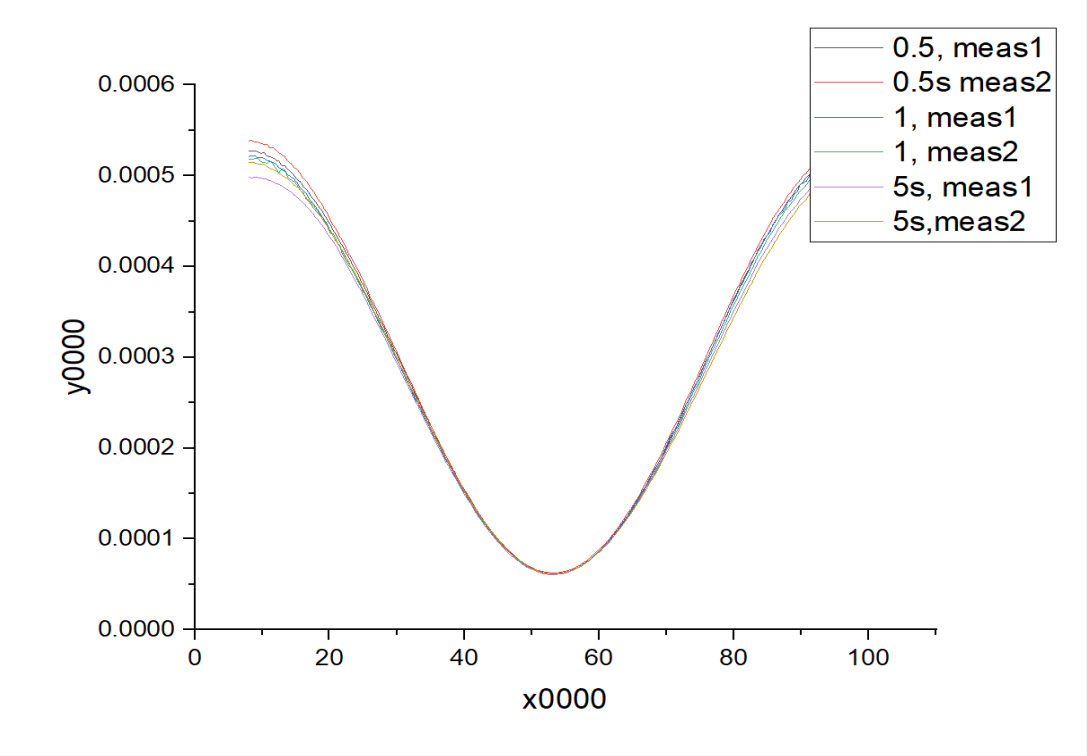

Figure 4. Polarization-Dependent Response (0°-90°) Captured over 201 Data Points

Additional Task: Raster Scan

After implementing the rotational sweep, I was tasked with developing a 2D Raster Scan for X-Y motion stages. The goal was to prepare the lab for image processing in the future.

Unlike the Newport-controlled rotation stage, the X-Y linear stages were actuated by Thorlabs Kinesis motors. The lab group had already created a basic Python script capable of establishing communication with these motors and performing basic movement commands. My objective was to evolve this actuation into an automated raster scanning motion.

The primary challenge here was moving from a procedural loop to an abstracted QCoDeS measurement. In a basic script, I was able to get the raster scan working with a nested loop. However, to benefit from QCoDeS's automatic database logging system and plotting, I had to integrate the raster scan into the functions from the measuring module.

The problem with a standard nested sweep is that the fast axis (X) usually resets to the start after every row (Y). This is inefficient and causes mechanical wear. Figure 5 illustrates the "Snake" pattern that was needed.

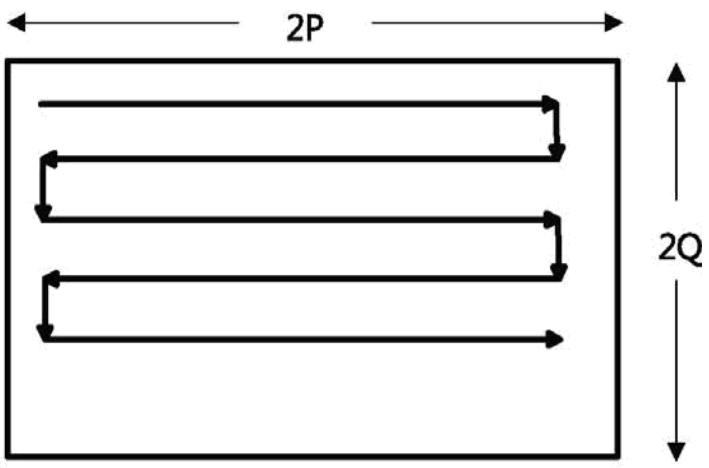

Figure 5. Conceptual Diagram of a Snake Raster Scan

Implementing the Snake Pattern via post_action

I discovered that the add_variable method in the measuring module includes an optional post_action parameter. This allowed me to pass a callback function that triggers every time a Y-row is completed.

I created a y_post_action function that increments a row counter, shown in Figure 6. If the row is even, it calls X_pos.activate_forward(); if odd, it calls X_pos.activate_reverse(). This "flipped" the X-vector logic inside the SnakeX parameter class before the next row started.

```
row = {"i": 0}

def y_post_action():
    row["i"] += 1

    if row["i"] % 2 == 0:
        X_pos.activate_forward()
    else:
        X_pos.activate_reverse()
```

Figure 6. Custom y_post_action() Function

Dummy Runs to Validate Raster Scan

To validate the software framework while awaiting the installation of the focusing lens, multiple dummy runs were conducted. A 2D scan was performed over an 8 mm by 8 mm area with a step size of 2 mm in both the X and Y directions.

Because no IR light was projected onto the detector during these tests, the resulting data represent noise in the system. Figure 7 shows the axes range from -1 to 9 mm; this shift occurs because the plotting software centers the pixels at the recorded coordinates (0 to 8 mm) and extrapolates the boundaries to create a continuous visual grid. This run successfully confirms that the snaking logic and data synchronization are ready for live imaging once the required lens is manufactured.

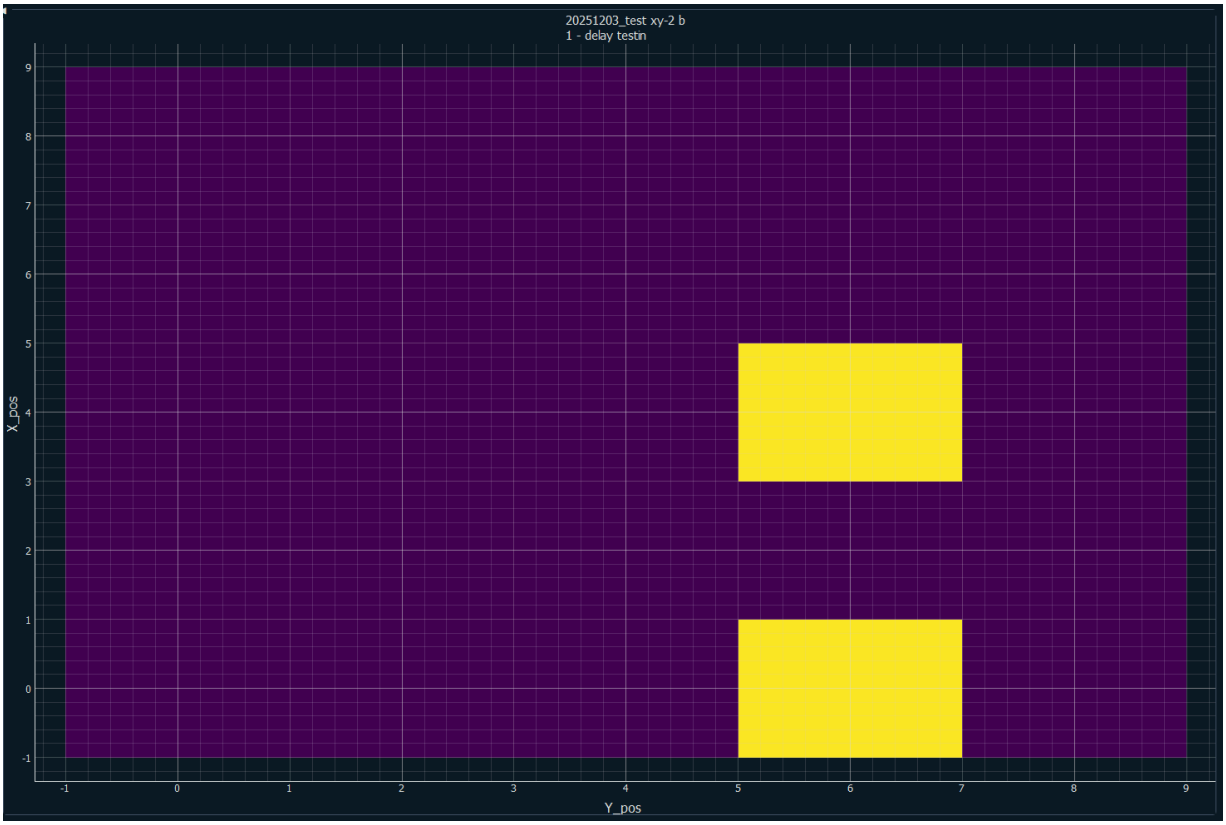

Figure 7. Visualization of a Dummy Raster Scan (0-8 mm)

Summary and Future Work

Throughout this semester, I successfully transitioned the TECNA characterization process from a fragmented, manual workflow into a cohesive, automated system.
Key Achievements:

- Converted and modernized the SMC100 driver, adding ESP-error handling and degree-based rotation.
- Integrated the driver into the existing QCoDeS measurement system.
- Validated the system with polarization data.
- Developed a "snaking" 2D raster scan framework for imaging.

Future work includes looking into defining the delay as a multiple of the lock-in time constant (e.g., 3τ). The thought is that it ensures signal stability while reducing unnecessary wait times.

The software framework for the 2D scan is complete and validated through "dummy" runs. The immediate next step is to perform live characterization once the focusing lens hardware for the X-Y setup is finalized.