FERMI NATIONAL ACCELERATOR LABORATORY
GQUEST PROJECT

Controlling Optical Cavities using Machine Learning

Author:
Edoardo Fazzari
edoardo.fazzari@santannapisa.it

Supervisor:
Chris Stoughton
stoughto@fnal.gov

Monday 25th September, 2023

Contents

1 Introduction **3**

2 Materials and Setup **5**
2.1 Mephisto Laser 5
2.2 Red Pitaya 6
2.3 Optics Layout 9
2.4 Red Pitaya's Connections 10
2.4.1 Red Pitaya's Radio Frequency (RF) Inputs 10
2.4.2 Red Pitaya's Radio Frequency (RF) Outputs 12
2.4.3 Red Pitaya's Analog Outputs 13
2.5 Oscilloscope's Connections 13
2.6 Reading Temperature 13
2.6.1 TMP35 Temperature Sensor 14
2.6.2 LabJack U3-HV 14

3 Experiments & Results **15**
3.1 Initial Temperature 16
3.2 Lock Maintaining 17
3.3 Software Control for Lock Maintaining 17
3.4 Reinforcement Learning & Temporal Difference Learning for Lock Maintaining 19
3.4.1 Q-Learning Control 21
3.4.2 Double Q-Learning Control 25

4 Python Code **26**
4.1 RedPitayaController Class 27
4.1.1 Constructor 27
4.1.2 Setting and Resetting Red Pitaya's Outputs 27
4.1.3 Reading the Scope 29
4.1.4 Temperature Scanning 30
4.1.5 Piezo Handling 30
4.1.6 PHD Lock 30
4.1.7 Maintain Lock: Software Control 32
4.2 RedPitayaQLearning Class 32
4.2.1 Constructor 32
4.2.2 Utils 33
4.2.3 Learning 33
4.2.4 General Utils 35
4.3 RedPitayaDoubleQLearning Class 35

4.3.1 Constructor 36
4.3.2 Utils 36
4.3.3 Learning 37

5 Conclusion 39

1 Introduction

The **GQuEST** experiment endeavors to investigate the unification of Quantum Mechanics and General Relativity within the realm of quantum gravity. This pursuit constitutes a well-established conundrum in the field of physics, characterized by its formidable nature due to the infinitesimal effects manifesting at the Planck scale, which measures a mere 10^{35} meters. A remarkable facet of the GQuEST experiment is its capacity to predict observable manifestations of Quantum Gravity on a larger and more accessible scale.

The fundamental concept underlying this experiment entails the transmission of a laser beam to a beam splitter. The incident light is then divided into two paths, subsequently returning along those paths. Unlike a conventional interferometer, which primarily monitors changes in mirror positions induced by room vibrations or the passage of gravitational waves, the GQuEST experiment pursues a more subtle objective. Lie et al.[5] present a model operating in four dimensions, wherein the metric properties manifest as oscillations in the dimensions of a sphere, dictated by a scalar field denoted as ϕ:

$$ds^2 = -dt^2 + (1 - \phi)(dr^2 + r^2 d\Omega^2) \tag{1}$$

The influence of the ϕ term modulates the spatial dimensions, introducing minute variations. This scalar field, ϕ, is an exceedingly small quantity that orchestrates the dynamic expansion and contraction of the sphere (see Figure 1), leading to alterations in distances. The quantum fluctuations in this scale subsequently engender fluctuations in the round-trip time taken by a photon to traverse the distance between mirrors in an interferometer. By constructing an exceptionally sensitive interferometer[13], it becomes conceivable to detect and characterize these fluctuations, offering a pathway to the detection of quantum gravity phenomena.

To achieve the level of laser sensitivity required for our experiment, it is essential to ensure a consistent emission of waves with a fixed wavelength. While it is a well-established fact that lasers typically emit waves of uniform wavelength, it is important to note that certain factors, such as the temperature of the crystal and the physical size of the crystal, can introduce minor variations in wavelength. The central aim of our experiment is to employ a Pound-Drever-Hall[2] technique to stabilize the laser output, maintaining it in an optimal Gaussian mode. Subsequently, the stabilized laser will be directed into an interferometer, specially designed to selectively filter out photons and other signal components, allowing only those photons associated with quantum gravity to reach the photon counter (see Figure 2). The core premise of our approach hinges on a binary outcome: a complete absence of detected photons would cast doubt on the validity of our theoretical framework,

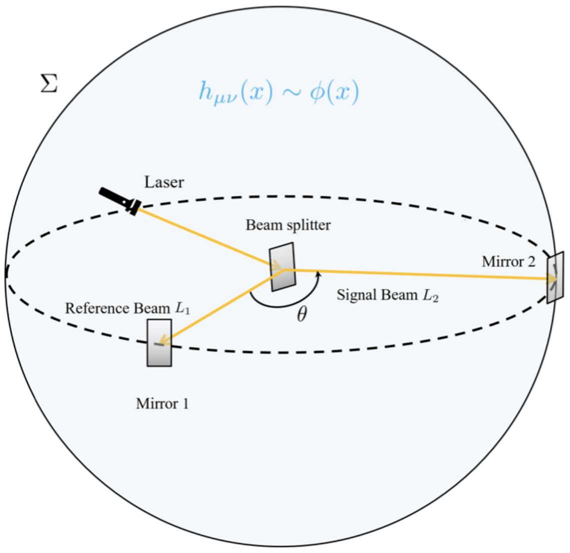


Figure 1: Setup of the interferometer. (Credit Lie et al.[5])

whereas the detection of any photons would serve as a strong indication of the presence of quantum gravity phenomena.

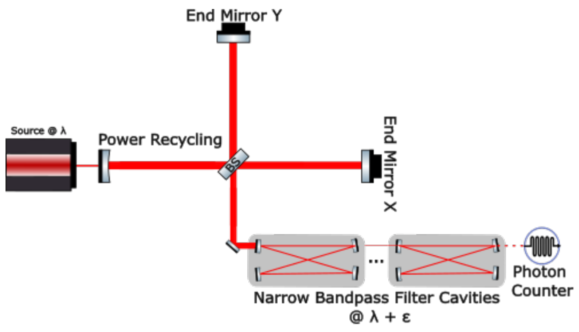


Figure 2: Simplified setup of GQuEST.

To achieve laser output stabilization, it is essential to establish a precise wavelength configuration where $2L/\lambda$ assumes an integer value. Under these conditions, the light entering the optical cavity undergoes repeated constructive interference, resulting in exponential growth within the cavity. Subsequently, a fraction of this amplified light escapes from the rear end of the cavity. Once the desired transverse electromagnetic mode (TEM00, see Figure 3) has been located, the subsequent objective involves maintaining the stability of the cavity. In pursuit of this goal, which constitutes the crux of our research, we have harnessed Reinforcement Learning (RL) algorithms to train an autonomous agent. This agent is tasked with dynamically adjusting the temperature of the laser's crystal in response to fluctuations in the optical signal emanating from the cavity.

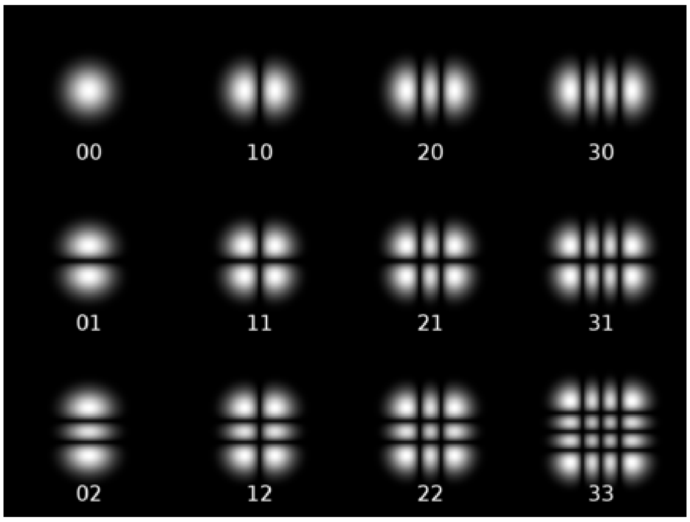


Figure 3: Hermite Gaussian modes.

The primary contributions of this research encompass:

- The development of dedicated software for electronic temperature control of a Mephisto laser.
- The creation of software capable of autonomously identifying the TEM00 mode within the optical cavity.
- The successful training and testing of a Q-Learning Agent, designed to preserve system stability by adapting to variations in the optical output.
- The effective training and testing of a Double Q-Learning Agent, similarly geared towards maintaining system stability by responding to changes in the optical signal emerging from the cavity.

2 Materials and Setup

In this sections all the devices that were used in the experiment are described along with the optical layout.

2.1 Mephisto Laser

The **Mephisto** laser system (documentation) consists of two self-contained units, the laser head and the control electronic unit, as depicted in Figure 4a and Figure 4b.

In order to operate the Mephisto laser system, the control electronics unit needs to be connected to the laser head. The provided electrical power is converted into coherent narrow bandwidth radiation with an efficiency of about 30%. The remaining power heats the diode laser, which must be cooled to prevent overheating. Furthermore, the wavelength of the diode laser depends on the junction temperature. Therefore, *temperature stabilization of the diode laser is essential.*

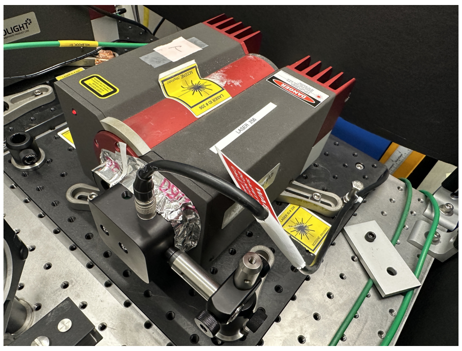
The wavelength of the Nd:YAG laser light also depends strongly on the **crystal temperature**, consequently the Nd:YAG laser crystal must be temperature stabilized as well. The control electronics unit is designed to provide all required subsystems to drive and control the Mephisto laser system, a schematic representation can be found in Figure 4c and Figure 4d:

- a *Laser Diode Driver* that provides a very stable, low noise injection current to the diode lasers up to a value of 3A. This subsystem also contains the protection circuitry that is essential for reliable operation of the laser system and a temperature controller that regulates the diode laser temperature.
- a *Precision Temperature Controller* that stabilizes the Nd:YAG crystal's temperature. Because of an integrated pre-stabilization stage, typical drifts of this controller are only a few 100 μK/min, corresponding to a variation of the laser frequency of less than 1 MHz/mm.
- *analog modulation inputs* (BNC connectors) for diode laser current and *laser crystal temperature* to externally control output power and laser frequency of the **Mephisto** laser system.
- a *diagnostic connector* (D-Sub connector) to monitor all vital signals and voltages of the **Mephisto** laser system without opening the control electronics unit.

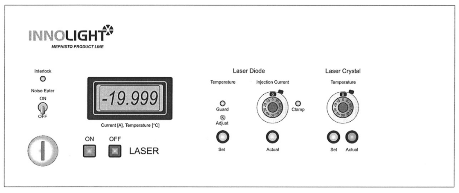
The frequency of the Mephisto laser can be tuned by changing the temperature of the monolithic laser crystal. This can either be done directly at the front panel of the control electronics unit using the appropriate dial or trimmer (see Figure 4), which results to be a tedious task, or by applying a voltage to the Laser Crystal temperature modulation input at rear panel of the control electronics unit, *our objective.*

2.2 Red Pitaya

The Red Pitaya used is a STEMlab 125-14 (`https://redpitaya.com/product/stemlab-125-14/`) with 512MB (4Gb) RAM, the system memory is a Micro SD 10Gb. All specifications are summarized in Table 1.


(a) **Mephisto** laset head

(b) **Mephisto** control electronics unit


(c) Front panel of the **Mephisto** control electronic unit

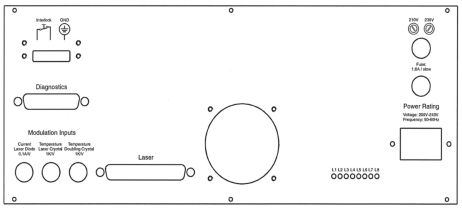

(d) Rear panel of the **Mephisto** control electronic unit

Figure 4: Laser setup

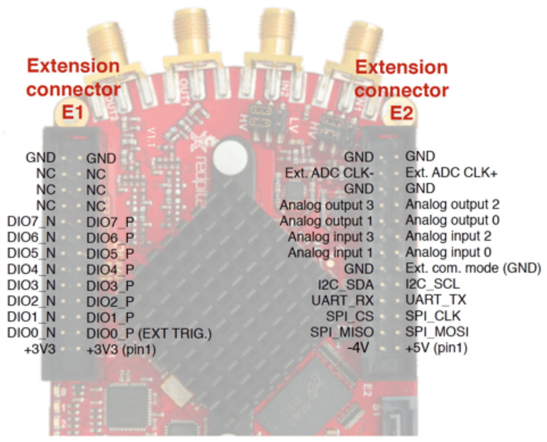


Figure 5: Red Pitaya's extensions

Table 1: Device Specifications

Category	**Specifications**
RF Inputs	
Channels	2
Sample rate	125 MS/s
ADC resolution	14 bit
Full scale voltage range	$\pm 1/\pm 20$ V
Input coupling	DC
Bandwidth	DC-60 MHz
Input impedance	1 MΩ
RF Outputs	
Channels	2
Sample rate	125 MS/s
ADC resolution	14 bit
Full scale voltage range	± 1 V
Load impedance	50Ω
Shortcut protection	Yes
Typical raising/falling time	2 V/10 ns
Bandwidth	DC-60 MHz
Extension Connection	
Digital IOs	16
Analog inputs	4 channels: 0-3.5 V 12 bit
Analog outputs	4 channels: 0-1.8 V 12 bit
Communication interfaces	I2C, SPI, UART
Available voltages	-4 V, +3.3 V, +5 V
Synchronization	
Trigger input	through extension connector
Daisy chain connection	over SATA connection
Ref. clock input	N/A

2.3 Optics Layout

Figure 6 shows the optics layout set at the *CryoModule Test Facility* (CMTF) at Fermilab. The details are the following:

1. **Laser**: 2W Mephisto laser.
2. **Attenuator + beam stop**: attenuating the power to about 65mW; beam stop receives the remaining power. This is done to not damage the phase modulator or the *Faraday isolator*.
3. **Lens**: to ensure that the beam's size is substantially smaller than the phase modulator's aperture.
4. **Phase modulator**: has an aperture about 2mm (diameter) and the lens. Receives an RF signal from the *Red Pitaya* and imprints phase modulation sidebands on the laser light at about 25MHz offset of frequency. The effect is shown in Figure 7.
5. **Lens**: needed to refocus the laser beam and to put a waist in the center of the Faraday.
6. **Lambda/4 waveplate**: change the phase lag between the incident light's electric and magnetic fields.
7. **Lambda/2 waveplate**: change the light's polarization direction. The two waveplates are aligned so that as much power as possible goes through the Faraday isolator.
8. **Faraday isolator**: it is a device which transmits light in a certain direction while blocking light in the opposite direction. There is a power loss, 60mW out of the isolator compared to the 65mW in. The lost power is mostly due to residual polarization mismatch, not due to losses or clipping. The remaining 5mW is dumped on a beam stop.
9. **Pair of steering mirrors**: placed roughly 90 degrees of *Gouy* phase apart, so they are roughly orthogonal actuators.
10. **Three mode matching lenses**: to match the incident laser beam's mode to the optical cavity's fundamental TEM00 mode.
11. **Cavity**: the beam is almost entirely reflected, even on resonance cince the cavity's back mirror is a high-reflector. However, some light does leak out of the cavity, which we monitor with a camera and a photodetector (PD)
12. **Camera and photodetector**: allow to tune up the alignment and to set the laser's temperature such that the TEM00 mode is within the frequency

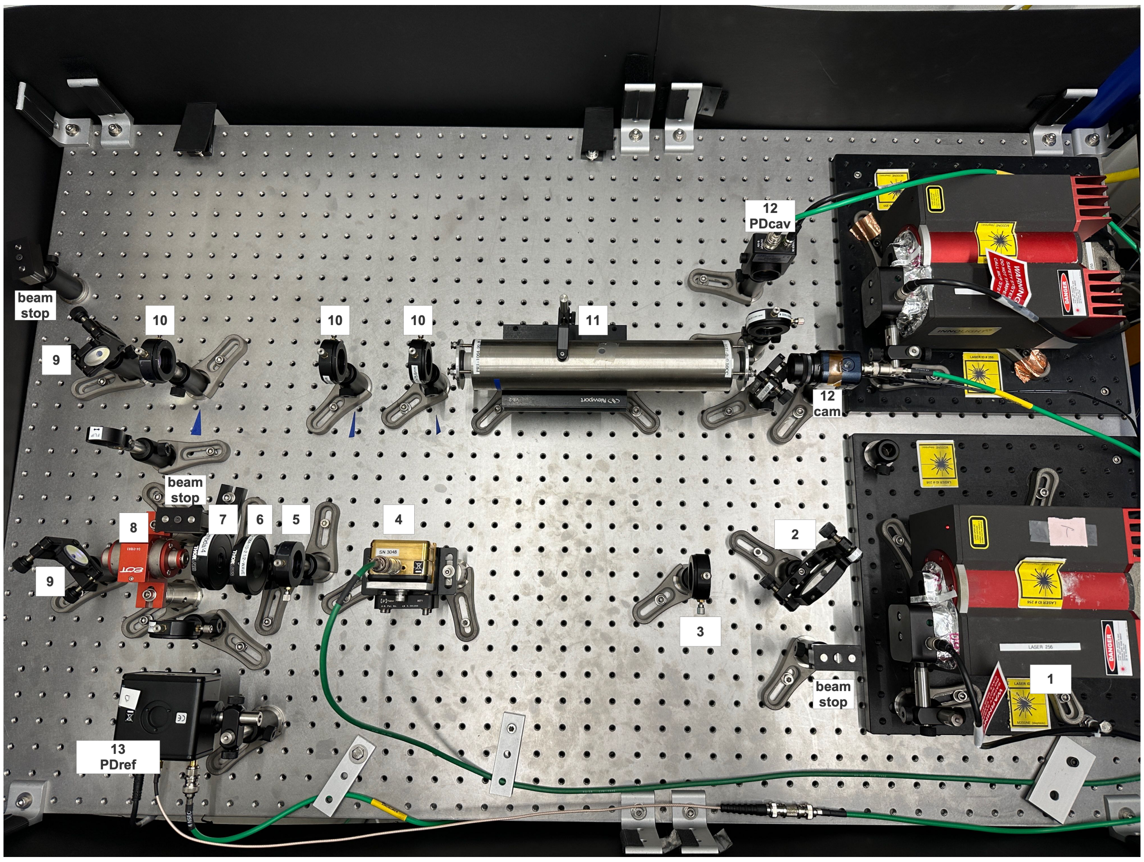


Figure 6: Optics Layout

adjustment range of the laser's piezo-actuated fast frequency control.

The light reflected back towards the mode matching lenses and the alignment mirrors interacts with the circulator (Faraday isolator) and is reflected to a lens and fast (125MHz) photodetector (PDref, 13). There are lens to focus the beam onto the small-area PD where it is sent to the *Red Pitaya*.

2.4 Red Pitaya's Connections

In this section Red Pitaya's connections are described. Figure 8 schematizes all the connections to the Red Pitaya.

2.4.1 Red Pitaya's Radio Frequency (RF) Inputs

Our Red Pitaya have two Radio Frequency (RF) inputs:

1. **Input 1** is the photodetector's output (the one next to the *Faraday*, PDref),

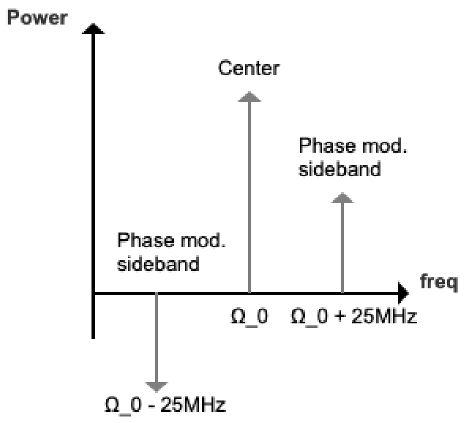


Figure 7: Phase modulator effect

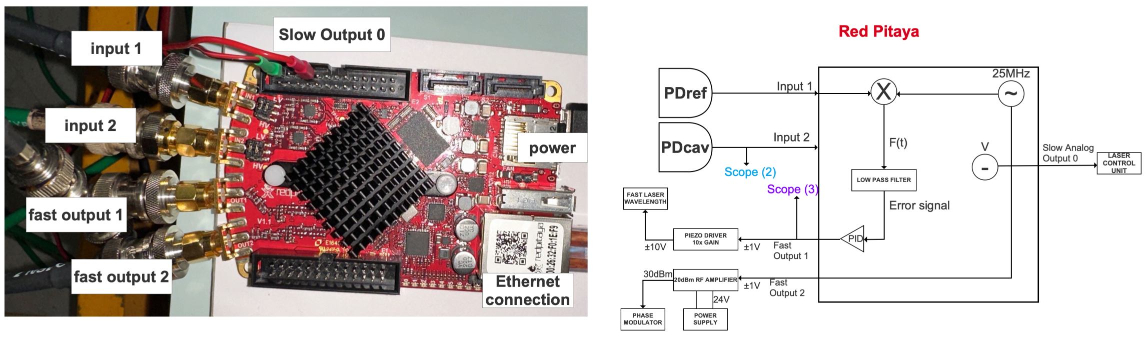


Figure 8: Red Pitaya's layout and connection schema

which measures the optical cavity's reflected power. This input is modulated at 25MHz from the *phase modulator* and it can be written as $A\sin(\omega t)$, where ω is 25MHz. We demodulate this signal in the *Red Pitaya* to generate our *PDH error signal* in the following way:

(a) we multiply input 1 to the signal generated by the*Red Pitaya*, which has the same frequency ω.

$$\begin{aligned} F(t) &= A\sin(\omega t)\cdot B\sin(\omega t) \\ &= AB\sin(\omega t)\sin(\omega t) \\ &= AB\Big(\frac{1}{2i}\Big)^2(e^{i\omega t}-e^{-i\omega t})(e^{i\omega t}-e^{-i\omega t}) \\ &= \frac{AB}{-4}(e^{2i\omega t}-2+e^{-2i\omega t}) \\ &= \frac{AB}{2}-\frac{AB}{2}\cos(2\omega t) \end{aligned} \tag{2}$$

The cosine term is killed by the *low pass filter* and the **error signal** is obtained.

2. **Input 2** is the photodetector's output (the one after the *cavity*, PDcav), which measures the light that goes through the optical cavity. This input is needed to control the temperature as described in section 3.

2.4.2 Red Pitaya's Radio Frequency (RF) Outputs

Our Red Pitaya have two Radio Frequency (RF) outputs:

1. **Output 1** goes to the *laser's fast frequency actuator* and to the *oscilloscope* (purple trace), as described in subsection 2.5). This output is generated by the *error signal* fed to a ***PID block*** in the *Red Pitaya*, which converts the error signal into a control signal. The PID fuction in the *Red Pitaya*, $H[\omega]$ has only a integrator, i.e., there are no proportional and differentiation parts (Figure 9). Since the actuator want signals between -100V and +100V (see documentation on page 5) and the range of the *Red Pitaya* is between -1V and +1V, we send *output 1* to a *Piezo driver*[1] with a $100x$ amplification factor.

2. **Output 2**: *Red Pitaya* outputs 25MHz with 2V peak-to-peak amplitude amplified up to a high-power *RF amplifier*, powered by an external +24 power

[1]A piezo controller or driver is used to control the motion of a piezo positioning device for generating piezoelectricity. Piezoelectricity is the charge created across certain materials when a mechanical stress is applied. Piezoelectric pressure sensors exploit this effect by measuring the voltage across a piezoelectric element generated by the applied pressure.

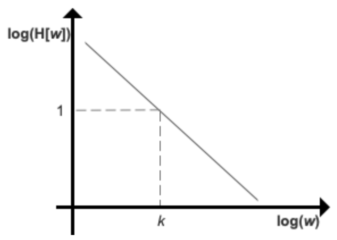


Figure 9: Actual PID in our *Red Pitaya*, k=1e3

supply, before going into the *phase modulator*. This output is needed for PDH lock.

2.4.3 Red Pitaya's Analog Outputs

Our Red Pitaya have four analog outputs as described in subsection 2.2, but only two (i.e., 0 and 1) are actually controllable by the *python package* we use in our code:

- **Output 0** (pin 17 on EC2) is used to control the temperature of the laser crystal, and it is connected to the rear panel of the **Mephisto** control eletronic unit.
- **Output 1** is currently not used.

2.5 Oscilloscope's Connections

We use an oscilloscope, Figure 10 to monitor the cavity's transmitted power obtained from the photodetector located after the optical cavity (blue trace), the voltage sent to the laser's fast frequency tuning port by the *Red Pitaya*'s output 1 (purple trace) in order to monitor the driver voltage, and the DC power on the cavity reflection PD (yellow trace).

We usually turn off the yellow trace once the system is aligned. On resonance, the TEM00 mode has a voltage of 1.44V with the transmission PD's gain set to 0dB gain (blue trace). The purple trace should have a voltage in the 0 to 1V range.

2.6 Reading Temperature

Both the temperature within the optical cavity and the ambient room environment can exert a significant influence on the requisite temperature for maintaining laser stability in the TEM00 mode. To address this consideration, we have chosen to incorporate an external temperature sensor, specifically the **TMP35**, in conjunction

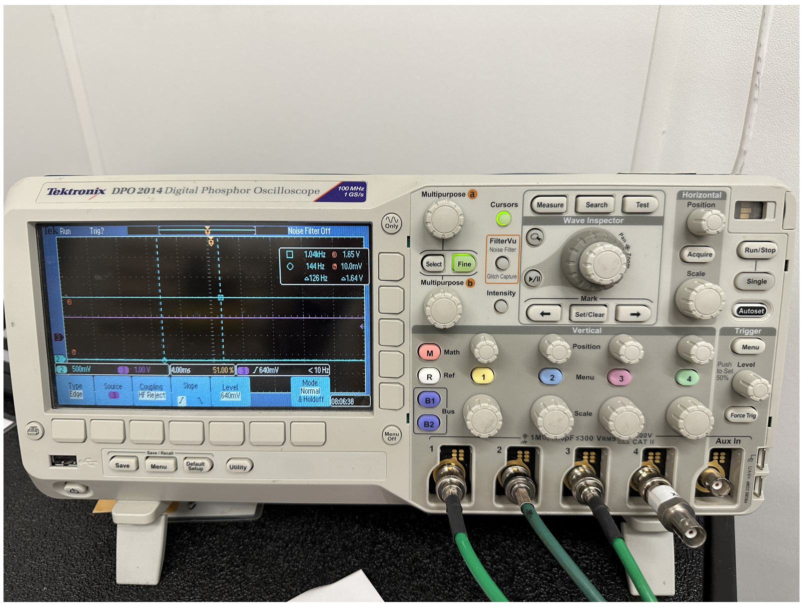


Figure 10: Oscilloscope used in out experiment.

with a data acquisition system known as the **LabJack U3-HV**. This integration enables us to collect temperature data through a Python script utilizing the `u3` Python package (available at this GitHub repository). The justification behind not interfacing the TMP35 sensor with the Red Pitaya lies in the absence of analog input functionality within the pyprl package, as evident in the ams module code.

2.6.1 TMP35 Temperature Sensor

The **TMP35** is a low voltage, precision centigrade temperature sensors. It provides a voltage output that is linearly proportional to the Celsius (centigrade) temperature, and does not require any external calibration to provide typical accuracies of $\pm 1°C$ at $+25°C$ and $\pm 1°C$ over $-40°C$ to $+125°C$ temperature range. The main features are described in Table 2 and IMAGE shows pin configuration.

2.6.2 LabJack U3-HV

To obtain temperature readings to our laptop, we employ a LabJack U3-HV device, which serves as a multifunction Data Acquisition (DAQ) system utilizing a USB interface. In order to utilize this hardware component effectively, it is imperative to install the **LabJack Driver** and configure the Python package as delineated in the documentation available at the following URL: GitHub repository.

FIGURE provides an illustration of the configuration for connecting the temperature

Table 2: TMP35 Sensor Specifications

Feature	**Specification**
Voltage Range	2.7 V to 5.5 V
Calibration Range	Calibrated directly in °C
Scale Factor	10 mV/°C (20 mV/°C on TMP37)
Accuracy Over Temperature	±2°C (typical)
Linearity	±0.5°C (typical)
Stability with Capacitive Loads	Stable
Temperature Range	-40°C to +125°C (operation to +150°C)
Quiescent Current	Less than 50 μA
Shutdown Current	0.5 μA (maximum)
Self-Heating	Low

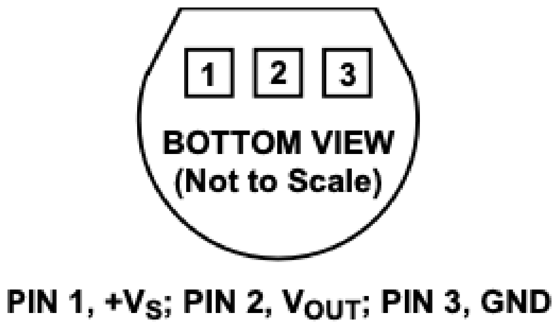


Figure 11: TMP35's pin configuration.

sensors to the LabJack U3-HV device. This setup entails utilizing the `Vs` output, which generates a +5V voltage, along with `AIN0` and `AIN0` to facilitate temperature data acquisition.

3 Experiments & Results

This section comprehensively delineates the experimental procedures and outcomes. It is structured into two distinct segments: the first elucidates the methodology employed for ascertaining the initial temperature requisite for stabilizing the TEM00 mode within the system, while the second expounds upon the strategies employed to sustain the system in a stable state, adhering to the desired mode. The experiments are delineated in a chronological sequence that mirrors the order in which they were conducted.

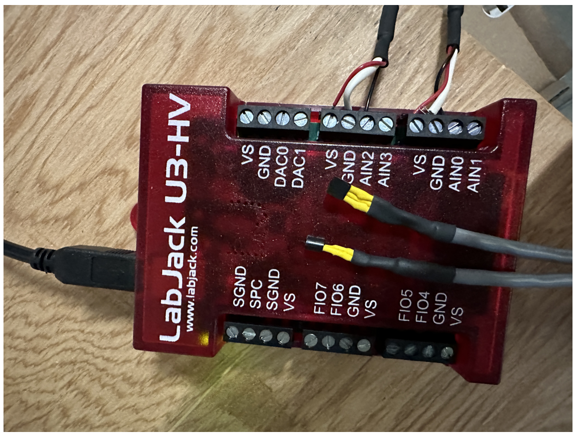


Figure 12: LabJack and TMP35 sensors.

3.1 Initial Temperature

Determining the precise temperature alignment corresponding to TEM00 mode can be a laborious and patience-testing endeavor, as it necessitates the meticulous adjustment of a dial on the Mephisto laser's control unit. To streamline this arduous task, an automated procedure was implemented. This automated method initializes the temperature at 22°C and invokes a computational routine outlined in Algorithm 1. In essence, the Red Pitaya device generates a progressively increasing voltage through its slow analog output channel. With each incremental voltage adjustment, accomplished by setting the `DAC2` value of the Red Pitaya, a data acquisition is performed, capturing two critical parameters: the signal received from the PDcav (*blue_signal*) and the voltage applied to the laser's rapid frequency tuning port (*purple_signal*). The nomenclature "orderedScopeTrace" aptly describes this method because it acquires a scope trace and subsequently restructures the data to ensure that the voltage ramp exhibits continuous progression without abrupt discontinuities, thereby positioning the trace's midpoint at the center of the voltage ramp. The algorithm exclusively relies on the blue signal, discerning its peak position to determine the termination condition of the iteration. If the peak occurs approximately at the midpoint of the ramp, within a specified tolerance threshold denoted as ϵ, the algorithm concludes, yielding a *True* result. Conversely, if the peak lies outside this central range, the algorithm continues to increment the voltage value, augmenting it by a fixed increment of 0.00025. If the voltage value reach 0.3 without satisfying the termination criterion, the algorithm exits the loop and returns *False*. In such cases, auxiliary functions are invoked to address this situation. Typically, the procedure entails a restart, as elucidated in the ensuing subsections.

Algorithm 1 Scanning Temperature

```
1: procedure ScanTemperature($\epsilon$=1000)
2:     for i in np.arange(0, 0.3, 0.00025) do
3:         SetDAC2($i$)
4:         $\_, blue\_signal \leftarrow$ orderedScopeTrace($self$)
5:         $half\_trace \leftarrow$ int($blue\_signal.shape[0]/2$)
6:         $blue\_peak\_index \leftarrow$ np.where($blue\_signal == blue\_signal.max$)
7:         $blue\_peak\_index \leftarrow blue\_peak\_index[0][0]$
8:         if $half\_trace - \epsilon < blue\_peak\_index < half\_trace + \epsilon$ and $blue\_signal.max > 0.95$ then
9:             Return True
10:    Return False
```

3.2 Lock Maintaining

After achieving lock, the subsequent challenge lies in the task of maintaining this locked state while preserving the TEM00 mode. This section provides a comprehensive account of all the experiments and analyses undertaken in pursuit of this particular objective.

It is important to note that the full scope trace duration is calculated as follows: $8, \mathrm{ns} \times 2^{14}$ (which equals 16,384) multiplied by the decimation factor (256) results in approximately 33,554,432 ns, or roughly 33.55 ms. Consequently, the frequency is given by $16384 \times 10^9/33554432 = 488,281.25$ Hz.

3.3 Software Control for Lock Maintaining

One straightforward approach to reestablishing the lock when it is lost is to execute Algorithm 1 each time the lock is disengaged. This procedure is elaborated upon in Algorithm 2, and it functions as follows:

1. Initially, the Red Pitaya's outputs are reset, and the fast input is configured to produce a ramp signal by invoking the function `RampPiezo`, as described in Algorithm 3. This function sets the ASG0 output of the Red Pitaya to a ramp waveform with the specified frequency and IQ0 to a quadrature signal in accordance with the input phase.
2. The `ScanTemperature` function is called, and if it returns *True*, indicating that the correct mode has been identified, a 10-second waiting period ensues to allow for more stable temperature conditions within the crystal and to center the peak of the PDcav signal on the center of the ramp.

3. Subsequently, the cavity is locked using the Pound-Drever-Hall (PDH) technique.
4. At intervals of 10 seconds, the lock status is monitored by analyzing the voltage level of `max_signal`.

Algorithm 2 Automatic Relock Function

```
procedure AutoRelock
    Reset
    RampPiezo
    if ScanTemperature(500) then
        Sleep(10)
        LockCavity
        while True do
            Sleep(10)
            $purple\_signal, blue\_signal \leftarrow$ Scope
            if Max(blue_signal) $< 0.95$ then
                Break
    AutoRelock
```

Algorithm 3 Ramping the Piezo

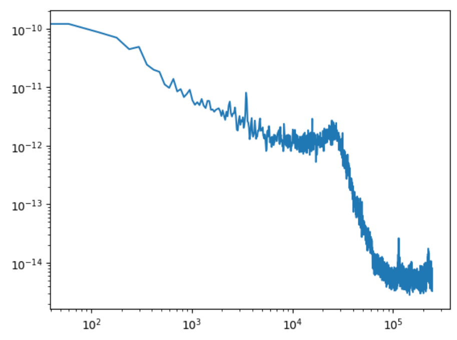
```
procedure RampPiezo(phase=15)
    Reset($self$)
    ScanPiezo($freq = 1/(8E - 9 * (2 * *14) * 256)$)
    setIQ0($phase = phase$)
```

While this method may not maintain the lock but rather initiates a fresh search for it, it can still be employed to conduct valuable analyses. The ensuing experiments were executed consecutively to this end.

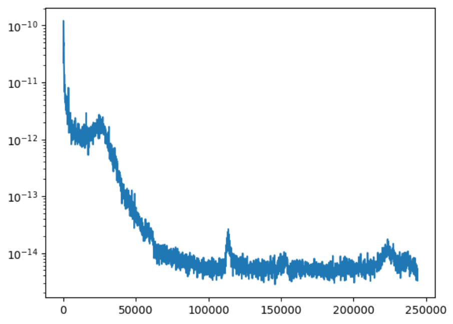
What is the Power Spectrum Density of the fast signal F? To compute the PSD of F the function *scipy.signal.welch(purple_signal, fs, nperseg=len(purple_signal) //2)* was used, where fs is the Hz value indicated above. The PSD of the average of 10 PSD of traces taken in succession is depicted in the following image:

How long lock last? To test how it last the auto_lock function was left running for a total time of 66804.48741364479 seconds (equivalent to 18h33m). Some information were extrapolated:

- *Number of lock events*: 32



(a) Average loglog of F



(b) Average semilogy of fast signal F

Figure 13: PSD of fast signal F

- *Minimum lock duration*: 10.086094617843628 seconds
- *Maximum lock duration*: 7437.1077709198 seconds (2h4m)
- *Average lock duration*: 2087.6402316763997 seconds (34m47s)

Average fast signal plot. Figure 14 shows the plot.

3.4 Reinforcement Learning & Temporal Difference Learning for Lock Maintaining

Reinforcement Learning (RL)[9] is a branch of machine learning that focuses on training agents to make sequences of decisions in an environment to achieve specific goals. Unlike supervised learning, where the model learns from labeled data, and unsupervised learning, where the model identifies patterns in unlabeled data, RL deals with decision-making in dynamic and uncertain settings. In RL, an agent interacts with an environment, takes actions, and receives feedback in the form of rewards or penalties based on its actions. The agent's objective is to learn a policy—a strategy for selecting actions—that maximizes its cumulative reward over time. The whole process is summarized in

Temporal Difference (TD)[1] learning is a fundamental concept in reinforcement learning, particularly in the context of model-free methods. TD learning algorithms bridge the gap between Monte Carlo methods, which require full episode trajectories to estimate returns, and dynamic programming, which requires a model of the environment. TD learning allows agents to learn and update their value estimates and policies incrementally, making it highly suitable for online learning scenarios. The key idea behind TD learning is to estimate the value of a state or state-action

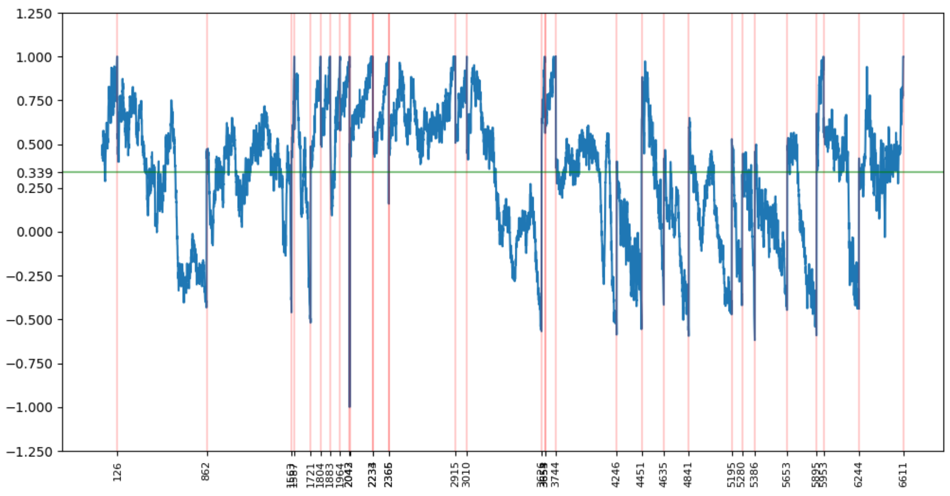


Figure 14: Every single point represents the mean value of the scope trace took every 10 seconds when the system was lock. Every red line indicates the start of a new lock event, and the green line is the average of the overall voltage values. The distances between each red line indicate how much the system was lock, in seconds, by just multiplying the difference times 10. $< F >$ is 0.33892209809867974.

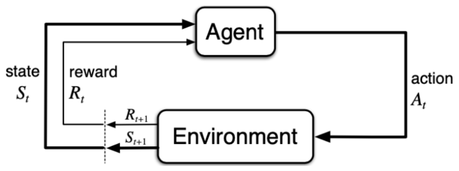


Figure 15: Agent-Environment interaction in Reinforcement Learning.

pair by bootstrapping from the current estimate and incorporating the immediate reward and the estimated value of the next state (or state-action pair). This process of updating value estimates based on temporal differences between successive estimates is what gives TD learning its name.

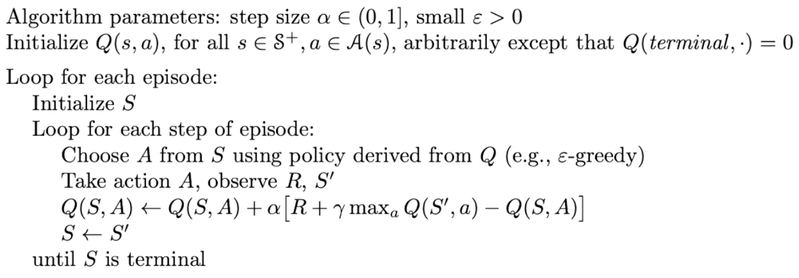
One of the most famous TD learning algorithms is Q-learning, which is used for estimating the quality of taking a specific action in a particular state and is commonly applied in reinforcement learning problems involving discrete state and action spaces. TD learning methods are highly efficient for real-time learning and are crucial in scenarios where episodic or full trajectories are not available or practical to use. In this research, Q-Learning[14] and Double Q-Learning[3] were exploited.

3.4.1 Q-Learning Control

Q-Learning[14] is an off-policy Temporal Difference (TD) algorithm used in Reinforcement Learning (RL) to find an optimal policy for an agent in an environment. It is defined as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(s,a) \right) \tag{3}$$

The learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated, e.g., making advantage of an *epsilon*-greedy strategy. The procedure is described in Figure 16.



```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    $S \leftarrow S'$
  until $S$ is terminal
```

Figure 16: Q-Learning procedure.

Adapting the approach to our specific case necessitated the precise definition of our *states* and *actions*. The voltage values obtained from the PDcav fall within the range of -1 to +1 volts. To facilitate the utilization of Q-Learning, we discretized this voltage range into intervals of 0.1 volts, yielding a total of 21 distinct states. Similarly, the temperature underwent a similar treatment, with a set of actions defined for the slow analog input. These actions induced temperature changes in increments ranging from -0.001 to 0.001, with a step size of 0.0005, resulting in a total of 5 actions. In addition to defining states and actions, other crucial parameters were established. The learning rate, denoted as α, was set to 0.4, while the *discount factor*, represented as γ, was assigned a value of 0.99. To balance exploration and exploitation, an *ϵ-greedy policy* was adopted. Initially, ϵ was set to 0.7 to favor exploration, but after 1000 episodes, it was adjusted to 0.3.

For a comprehensive understanding of the procedure, Algorithm 4 provides a detailed overview. Essentially, it incorporates the Q-Learning technique into Algorithm 2.

The `test` parameter is a Boolean that regularize the agent's configuration parameters to be exclusively greedy.

Following the training of Q-Learning with a total of 5000 episodes, the resultant Q-matrix was subsequently utilized and subjected to comprehensive analysis. This analysis aimed to facilitate a comparative evaluation with the software control strategy. Specifically, it involved an examination of the average performance of the rapid signal as well as the duration of the prevailing lock state. The intent of this analysis was to discern and assess any discernible enhancements or improvements in the system's behavior.

How long lock last? To compare this new strategy to the *software control* solution, we tested it setting the number of episodes, i.e., the number of lock-unlock events to 32. However, do to time and to a problem to the power cable the system stop at episode 4 without completing it. The total run was 210746.8703019619 seconds, equivalent to 2d6h32m. Summarizing:

- *Number of lock events*: 4 (last episode not concluded)
- *Minimum lock duration*: 269.89439964294434 (4m29s)
- *Maximum lock duration*: 195901 seconds (2d6h25m)
- *Average lock duration*: 52686.717575490475 seconds (11h51m)

Even though the first 3 episodes perform poorly, perhaps due to have just turned on the laser, much better results were obtained compered to the software control strategy. The Q-Learning agent proved to be able to maintain lock even after days.

Average fast signal plot. The figure presented in Figure 17 illustrates a segment of the fast signal plot obtained during the training phase of the Q-Learning algorithm. Throughout the training process, a subset of actions is executed randomly, resulting in the signal deviating from its mean value, which is approximately 0.297V. This randomness in action selection often causes the signal to lose lock.

In contrast, Figure 18 showcases the fast signal plot during the testing phase, where the agent exclusively opts for greedy actions. The figure provides a closer examination of the signal's behavior. Notably, the signal exhibits a more concentrated pattern around a mean value of approximately 0.32V during this phase. An interesting observation is the absence of actions that increase the temperature by 0.0025V, indicating that the Q-learning agent likely found such actions to be ineffective. A detailed examination in the zoomed-in section of the figure reveals that whenever the signal deviates, corrective actions are promptly taken to steer it back toward the mean, leading to incremental increases or decreases as needed.

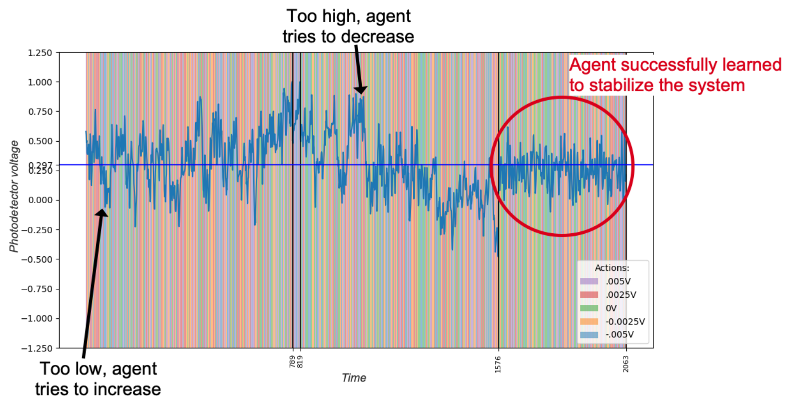


Figure 17: Fast signal during Q-Learning training. The black lines indicate lock events, and the action taken by the agent is highlighted by a specific color at every time step.

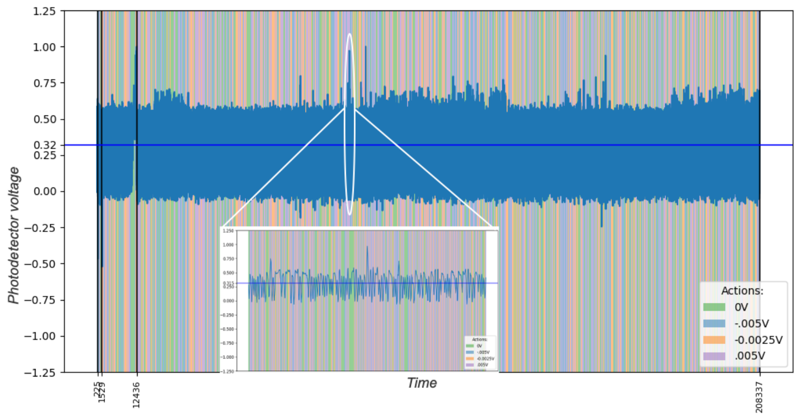


Figure 18: Fast signal during Q-Learning testing.

Assessment of the Q-Matrix The Q-matrix, which is derived from the training of the learning agent, is visually represented using a heatmap as illustrated in Figure 19. The matrix provides valuable insights into the agent's decision-making process. It becomes apparent that certain voltage ranges are never encountered, and the Q-matrix reveals which actions the agent deems most crucial for each voltage state. This determination is made by selecting the action associated with the highest value for a given state, such as a temperature range. Notably, there is relatively little disparity among the various action-state pairs. This uniformity in Q-values may be attributed to the way rewards are defined; specifically, they are binary, with a value of 1 indicating successful lock maintenance and 0 indicating loss, without regard for the episode's temporal progression. The concept of incremental rewards, which accounts for episode progression, is introduced in subsubsection 3.4.2.

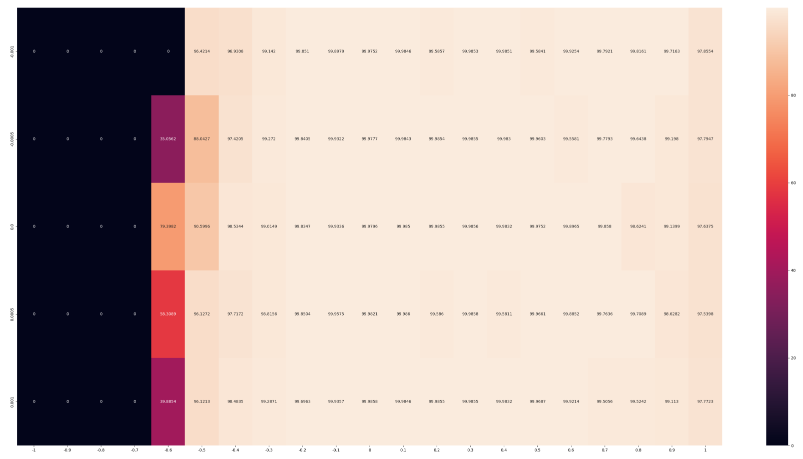


Figure 19: Q-matrix heatmap.

Impact of Actions on the Fast Signal In order to comprehensively ascertain the influence of agent-initiated changes in temperature on the fast signal, we have developed a correlation matrix, as detailed in Table 3. This matrix facilitates the establishment of a connection between the agent's actions and the subsequent state of the system, observed 0.1 seconds later.

I define stability as a state in which the signal undergoes minimal alteration, typically less than 0.03V. This threshold corresponds roughly to one-fifth of the standard deviation of the fast signal, which is calculated as 0.1624. Analyzing the specific impact of individual actions reveals intriguing insights. For instance, interpreting the effect of action -0.005 in isolation can be challenging; however, when juxtaposed with action -0.0025, it becomes apparent that it either sustains or slightly elevates signal stability. Conversely, action -0.0025 carries a higher likelihood of inducing a voltage increase in the photodetector signal downstream of the cavity. Notably, action 0 appears to consistently preserve signal stability, but an interesting observation arises when the temperature remains unaltered—under such circumstances, the signal generally tends to increase. Action 0.0025 is conspicuously absent from the agent's choices, implying its limited impact. In contrast, action 0.005 exerts a substantial influence by significantly reducing signal values. Remarkably, it stands as the most frequently employed action, indicative of a consistent trend where the signal typically inclines towards increase when subjected to action 0, necessitating the application of action 0.005 to maintain system stability.

Table 3: Action and fast signal relation.

	Decrease	**Stable**	**Increase**
-0.005	13165	16440	16432
-0.0025	2689	15555	27850
0	335	27309	19343
0.0025	0	0	0
0.005	65219	2973	1024

Correlation analysis between temperature and action taken. In order to investigate potential correlations between room temperature and the actions undertaken by the agent, we employed statistical analysis techniques. Specifically, we calculated both the Pearson correlation coefficient and the distance correlation to assess the presence of linear and non-linear correlations, respectively.

The *Pearson correlation coefficient*[4, 8] yielded a value of 0.0037, suggesting a lack of linear correlation between temperature and actions taken. Furthermore, the associated p-value of 0.08 indicated that this observed correlation, though small, lacked statistical significance. To explore non-linear relationships, we also computed the *distance correlation*[10, 11, 12], which returned a value of 0.00489. This result implies that the signal and the actions taken are statistically independent of each other.

This finding holds significant implications as it demonstrates that temperature does not exert a direct influence on the rapidity of the signal, contrary to initial assumptions.

3.4.2 Double Q-Learning Control

Double Q-Learning[3] is an analogous to Q-Learning, but it divides the time steps in two, e.g., by flipping a coin on each step). If the coin comes up heads, the update is:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, arg\max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)] \quad (4)$$

If the coin comes up tails, then the same update is done with $Q1$ and $Q2$ switched, so that $Q2$ is updated. The procedure is described in Figure 20.

Double Q-Learning offers several notable advantages over its precursor, Q-Learning. One paramount benefit is its capacity to mitigate the overestimation bias inherent in Q-Learning. By maintaining two distinct Q-value functions, Double Q-Learning ameliorates the issue of overly optimistic action-value estimates that can lead to

```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q_1(s,a)$ and $Q_2(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(terminal,\cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using the policy $\varepsilon$-greedy in $Q_1 + Q_2$
    Take action $A$, observe $R$, $S'$
    With 0.5 probabilility:
      $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S',a)\big) - Q_1(S,A)\Big)$
    else:
      $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S',a)\big) - Q_2(S,A)\Big)$
    $S \leftarrow S'$
  until $S$ is terminal
```

Figure 20: Q-Learning procedure.

suboptimal policies. This reduction in overestimation bias not only enhances policy evaluation but also engenders a more robust and stable learning process, particularly in environments characterized by noisy rewards or uncertainty. Furthermore, the algorithm's improved accuracy in estimating action values facilitates more effective exploration strategies, enabling agents to discover optimal policies with greater efficiency. Thus, these advantages collectively position Double Q-Learning as a compelling choice for addressing the limitations of traditional Q-Learning and enhancing the efficiency and effectiveness of reinforcement learning agents with only the cost of doubling the memory requirements, but without increasing the amount of computation per step.

Algorithm 5 provides a detailed overview. Essentially, it only adds the handling of the two Q-matrices.

The `test` parameter is a Boolean that regularize the agent's configuration parameters to be exclusively greedy. The reward was changed in this case to underline even more the importance to maintain lock longer, although it is not clear if this provides better performance or not.

Unfortunately, do to time and the good performance of the Q-Learning the training and the test have not been performed yet, however the code is present as documented in subsection 4.3.

4 Python Code

Within this section, a comprehensive documentation has been meticulously prepared for all code elements.

4.1 RedPitayaController Class

To facilitate streamlined control of the RedPitaya device, a Python class has been meticulously developed. In this following subsection, a comprehensive exposition of all available functions is presented and discussed in order to elucidate their functionality.

4.1.1 Constructor

The constructor connects to the Red Pitaya using `pyprl`, creating the `redpitaya` object.

```
class RedPitayaController(object):
    def __init__(self, hostname: str, user: str = 'root',
                 password: str = 'root', config: str = 'fermi',
                 gui: bool = False):
        try:
            p = Pyrpl(hostname=hostname, user=user,
                      password=password, config=config, gui=gui)
            self.redpitaya = p.rp
        except Exception as e:
            print(e)
```

4.1.2 Setting and Resetting Red Pitaya's Outputs

A series of functions has been diligently crafted to exercise command and control over the Red Pitaya's output channels. These functions have been tailored to administer precise manipulation of ASG0, ASG1, IQ0, DAC2, and PID0. Furthermore, a dedicated function has been incorporated into the class to facilitate a comprehensive reset operation, enabling the restoration of these output channels to their default states as required by the experimental protocols under consideration.

```
def reset(self) -> None:
    # Turn off arbitrary signal generator channel 0
    self._setAsg0(output_direct='off', amp=0, offset=0)
    # Turn off arbitrary signal generator channel 1
    self._setAsg1(output_direct='off', amp=0, offset=0)
    # Turn off I+Q quadrature demodulation/modulation modules
    self.redpitaya.iq0.output = 'off'
    self._setIQ0(output_direct='off')
    # Turn off PID module 0
    self._setPid0(0, 0, 0, 0, 'off')
    # Turn off dac2
    self.setdac2(0)

def _setAsg0(self, waveform: str = 'halframp', output_direct: str = 'out1',
             amp: float = 0.5, offset: float = 0.5, freq: float = 1e2) -> None:
    self.redpitaya.asg0.setup(waveform=waveform, output_direct=output_direct,
                              trigger_source='immediately', offset=offset,
                              amplitude=amp, frequency=freq)

def _setAsg1(self, waveform: str = 'halframp', output_direct: str = 'out2',
             amp: float = 0.8, offset: float = 0.0, freq: float = 25e6) -> None:
    self.redpitaya.asg1.setup(waveform=waveform, output_direct=output_direct,
                              trigger_source='immediately', offset=offset,
                              amplitude=amp, frequency=freq)

def _setIQ0(self, frequency: float = 25e6, bandwidth: list = [2e6, 2e6],
            gain: float = 0.5, phase: int = 0, acbandwidth: float = 5e6,
            amplitude: float = 1., input: str = 'in1', output_direct: str = 'out2',
            output_signal: str = 'quadrature', quadrature_factor: int = 1) -> None:
    self.redpitaya.iq0.setup(frequency=frequency, bandwidth=bandwidth, gain=gain,
                             phase=phase, acbandwidth=acbandwidth, amplitude=amplitude,
                             input=input, output_direct=output_direct,
                             output_signal=output_signal,
                             quadrature_factor=quadrature_factor)

def setdac2(self, voltage: float = 0.) -> None:
    self.redpitaya.ams.dac2 = voltage   # pin 17 output 0

def _setPid0(self, ival: float = 0, integrator: float = 1e3,
            proportional: float = 0, differantiator: float = 0, input='iq0',
            output_direct: str = 'out1') -> None:
    # Clear integrator
    self.redpitaya.pid0.ival = ival
    # Proportinal
    self.redpitaya.pid0.p = proportional
    # Integrator
    self.redpitaya.pid0.i = integrator
    # differentiator
    self.redpitaya.pid0.d = differantiator
    # input or output
    self.redpitaya.pid0.input = input
    self.redpitaya.pid0.output_direct = output_direct
```

4.1.3 Reading the Scope

The Red Pitaya is equipped with an internal oscilloscope, which serves as a valuable tool for conducting the analyses as described earlier. To harness this functionality effectively, we have developed two distinct functions: the first function is designed to acquire data directly from the oscilloscope, while the second function also leverages the `scope` functions but additionally restructures the acquired data, thereby aligning the ramp signal into a linear trajectory.

```
def scope(self, input1='out1', input2='in2', hysteresis=0.01, trigger_source='immediately'):
    self.redpitaya.scope.decimation = 256
    self.redpitaya.scope.input1 = input1
    # Scope's second input
    self.redpitaya.scope.input2 = input2
    self.redpitaya.scope.threshold = self.redpitaya.asg0.offset +
                                     self.redpitaya.asg0.amplitude / 2
    # Trigger Hysteresis
    self.redpitaya.scope.hysteresis = hysteresis
    # Trigger Time Delay
    self.redpitaya.scope.trigger_delay = 0
    # Take a Scope Trace
    self.redpitaya.scope.trigger_source = trigger_source
    return self.redpitaya.scope.single()

def _orderedScopeTrace(self, ordered=False):
    purple_signal, blue_signal = self.scope()
    # find wave peak
    purple_signal_peak_index = np.where(purple_signal == purple_signal.max())[0][0]
    first_position = purple_signal_peak_index + 1
    purple_signal = np.concatenate(
                    (purple_signal[first_position:], purple_signal[:first_position])
                    )
    blue_signal = np.concatenate(
                    (blue_signal[first_position:], blue_signal[:first_position])
                    )
    return purple_signal, blue_signal
```

4.1.4 Temperature Scanning

The following function is the implementation of Algorithm 1.

```
def scan_temperature(self, epsilon=1000) -> bool:
    for i in np.arange(0, 0.3, 0.00025):
        # set temperature
        self.setdac2(i)
        # take scope
        _, blue_signal = self._orderedScopeTrace()
        half_scope_trace = int(blue_signal.shape[0]/2)

        blue_signal_peak_index = np.where(blue_signal == blue_signal.max())[0][0]
        if half_scope_trace - epsilon < blue_signal_peak_index <
                half_scope_trace + epsilon and blue_signal.max() > .95:
            print(i)
            return True
    return False
```

4.1.5 Piezo Handling

Two functions are implemented to ramp the Piezo, using previously defined functions.

```
def scanPiezo(self, asg: bool = True, output_direct: str = 'out1', amp: float = 0.5,
             offset: float = 0.5, freq: float = 1e2) -> None:
    if asg:
        self._setAsg0(waveform='halframp', output_direct=output_direct,
                      amp=amp, offset=offset, freq=freq)
    else:
        self._setAsg1(waveform='halframp', output_direct=output_direct,
                      amp=amp, offset=offset, freq=freq)

def ramp_piezo(self, phase=15):
    self.reset()
    self.scanPiezo(freq=1 / (8E-9 * (2 ** 14) * 256))
    self._setIQ0(phase=phase)
```

4.1.6 PHD Lock

To perform PHD lock the following function was implemented inside the class:

```
def lockCavity(self, phase=20):
    print("Scan Piezo")
    self.scanPiezo(freq=1 / (8E-9 * (2 ** 14) * 256))
    print("Run on Modulation")
    self._setIQ0(phase=phase)
    print("Take a scope trace")
    scope_trace = self._scopeTrace()
    print("Done taking scope trace")
    np.save('scope_trace.npy', scope_trace)
    print("Saved scope trace")
    ch1, ch2 = scope_trace
    # Guess Initial Parameters
    print("Curve fit")
    offs = np.mean(ch2)
    gamma = (np.max(ch1) - np.min(ch1)) / 10
    x0 = (np.max(ch1) - np.min(ch1)) / 2
    amp = (np.max(ch2) - np.mean(ch2)) * (x0 ** 3)
    # Curve Fit
    poptLine, pcovLine = scipy.optimize.curve_fit(lPrime, ch1, ch2,
                                                  p0=[amp, offs, gamma, x0])
    fit = lPrime(ch1, poptLine[0], poptLine[1], poptLine[2], poptLine[3])
    # Plot Measured Data and Curve Fit
    plt.figure(1)
    plt.title("PDH Error Signal")
    plt.xlabel("PZT Drive Voltage (V)")
    plt.ylabel("Error Signal (V)")
    plt.grid(True)
    plt.plot(ch1, ch2)
    plt.plot(ch1, fit)

    print("Go back to resonance")
    # Go to resonance (CONSTANT PIEZO)
    self._setAsg0(waveform='dc', output_direct='out1', offset=poptLine[3])

    print("Close the feedback loop")
    # Close the Feedback Loop
    # Set PID gains and corner frequencies
    # Set Point
    self.redpitaya.pid0.setpoint = poptLine[1]
    self._setPid0()
```

4.1.7 Maintain Lock: Software Control

Algorithm 2 is implemented in the following code snippet:

```
def auto_relock(self):
    self.reset()
    self.ramp_piezo()
    if self.scan_temperature(1000):
        time.sleep(10)
        try:
            self.lockCavity()
        catch:
            self.auto_relock()
        starting_time = time.time()
        print(f'Locked at: {starting_time}')
        while True:
            time.sleep(10)
            print(f'Seconds after start: {time.time() - starting_time}')
            purple_signal, blue_signal = self.scope()
            print(f'purple (fast) signal mean: {purple_signal.mean()}')
            print(f'blue signal mean: {blue_signal.mean()}')
            if blue_signal.max() < 0.95:
                end_time = time.time()
                print(f'Lost lock at: {end_time}. Took {end_time-starting_time}')
                break
    self.auto_relock()
```

A try-catch statement was added to handle cases in which *scipy.optimize.curve_fit* fails.

4.2 RedPitayaQLearning Class

To integrate Q-Learning with our Red Pitaya a specific Python class was developed.

4.2.1 Constructor

In the constructor, all the important parameters needed to handle the lock and Q-Learning are set up. Also functionality for reading room temperature using the `u3` library are included. However, it's important to note that the temperature reading doesn't impact the Q-Learning algorithm, but it is only used for external analysis.

```
def __init__(self, hostname: str, user: str = 'root', password: str = 'root',
             config: str = 'fermi', gui: bool = False, load=False, test=False):
    self.rpc = RedPitayaController(hostname, user, password, config, gui)
    # states
    self.voltage_range = np.arange(-1, 1.1, 0.1)
    self.num_states = len(self.voltage_range)  # 21
    # actions
    self.action_range = np.arange(-.001, .0015, .0005)
    self.num_actions = len(self.action_range)  # 5
    # SARSA parameters
    self.learning_rate = 0.4
    self.discount_factor = 0.99
    self.epsilon = 0.7
    self.num_episodes = 5000
    self.tmp35 = u3.U3()
    self.test = test

    if self.test:
        self.epsilon = 0

    # Q-values
    if load:
        self.Q = np.load('q_qlearning.npy')
    else:
        self.Q = np.zeros((self.num_states, self.num_actions))
```

4.2.2 Utils

Here some utility functions:

```
def _get_state_index(self, temperature):
    return np.argmin(np.abs(self.voltage_range - temperature))

def _get_action_index(self, action):
    return np.argmin(np.abs(self.action_range - action))
```

4.2.3 Learning

Q-Learning is executed using the following functions, which integrate the `auto_relock()` functionality from the `RedPitayaController` into the Q-Learning analysis.

```
def qlearning(self, episode: int = 0):
    while episode < self.num_episodes:
        print(f'EPISODE {episode}')
        if episode == 1000 and not self.test:
            self.epsilon = 0.3
        self.rpc.reset()
        self.rpc.ramp_piezo()
        if self.rpc.scan_temperature(500):
            time.sleep(10)
            try:
                self.rpc.lockCavity()
            except:
                continue
            system_unlock = False
            print(f'\tLocked at: {time.time()}')
            print(f'\tInitial temperature voltage: {self.rpc.redpitaya.ams.dac2}V')
            purple_signal, _ = self.rpc.scope()
            print(f'\tFast signal mean: {purple_signal.max()}')
            state = self._get_state_index(round_to_nearest_0_1(purple_signal.max()))
            while True:
                time.sleep(1)
                # Choose action using epsilon-greedy policy
                if np.random.rand() < self.epsilon and not self.test:
                    action = np.random.choice(self.num_actions)
                else:
                    action = np.argmax(self.Q[state, :])
                print(f'\tAction index: {action}')
                self.rpc.setdac2(self.rpc.redpitaya.ams.dac2 + self.action_range[action])
                print(f'\tTemperature voltage: {self.rpc.redpitaya.ams.dac2}V')
                time.sleep(0.1)
                # Get the next state, reward, and system_unlock
                purple_signal, blue_signal = self.rpc.scope()
                print(f'\tFast signal mean: {purple_signal.max()}')
                if blue_signal.max() < 0.95:
                    print(f'\tLost lock at: {time.time()}')
                    system_unlock = True
                next_state = self._get_state_index(round_to_nearest_0_1(purple_signal.max()))
                reward = 1 if not system_unlock else 0
                print(f'\tState index: {next_state}')
                ain0bits, = self.tmp35.getFeedback(u3.AIN(0))
                ain0Value = self.tmp35.binaryToCalibratedAnalogVoltage(ain0bits,
                                                                      isLowVoltage=False,
                                                                      channelNumber=0)
                ain2bits, = self.tmp35.getFeedback(u3.AIN(2))
                ain2Value = self.tmp35.binaryToCalibratedAnalogVoltage(ain2bits,
                                                                      isLowVoltage=False,
                                                                      channelNumber=0)
                print(f'\tTMP35 AIN0: {ain0Value}')
                print(f'\tTMP35 AIN2: {ain2Value}')
                # Update Q-values
                if not self.test:
                    max_next_q = np.max(self.Q[next_state, :])
                    self.Q[state, action] = (self.Q[state, action] +
                                             self.learning_rate *
                                                    (reward +
                                                    self.discount_factor * max_next_q
                                                    - self.Q[state, action]))
                state = next_state

                if system_unlock:
                    if not self.test:
                        np.save('q_qlearning.npy', self.Q)
                    episode += 1
                    break
```

4.2.4 General Utils

Here some utility functions:

```
def _get_state_index(self, temperature):
    return np.argmin(np.abs(self.voltage_range - temperature))

def _get_action_index(self, action):
    return np.argmin(np.abs(self.action_range - action))
```

4.3 RedPitayaDoubleQLearning Class

To integrate Double Q-Learning with our Red Pitaya a specific Python class was developed. (The code is very similar to the `RedPitayaQLearning Class`).

4.3.1 Constructor

In the constructor, all the important parameters needed to handle the lock and Double Q-Learning are set up.

```python
def __init__(self, hostname: str, user: str = 'root', password: str = 'root',
             config: str = 'fermi', gui: bool = False, learning_rate=0.4,
             discout_factor=0.99, epsilon=0.7, num_episodes=5000,
             load=False, test=False):
    self.rpc = RedPitayaController(hostname, user, password, config, gui)
    # states
    self.voltage_range = np.arange(-1, 1.05, 0.1)
    self.num_states = len(self.voltage_range)  # 21
    # actions
    self.action_range = np.arange(-.001, .0015, .0005)
    self.num_actions = len(self.action_range)  # 5
    # SARSA parameters
    self.learning_rate = learning_rate
    self.discount_factor = discout_factor
    self.epsilon = epsilon
    self.num_episodes = num_episodes
    self.tmp35 = u3.U3()
    self.test = test

    if self.test:
        self.epsilon = 0

    # Initialize Q-values for two Q-functions
    if load:
        self.Q1 = np.load('q1_dqlearning.npy')
        self.Q2 = np.load('q2_dqlearning.npy')
    else:
        self.Q1 = np.zeros((self.num_states, self.num_actions))
        self.Q2 = np.zeros((self.num_states, self.num_actions))
```

4.3.2 Utils

Here some utility functions:

```python
def _get_state_index(self, temperature):
    return np.argmin(np.abs(self.voltage_range - temperature))
```

```
def _get_action_index(self, action):
    return np.argmin(np.abs(self.action_range - action))
```

4.3.3 Learning

Doubling Q-Learning is executed using the following functions, which extends the previously presented Q-Learning function.

```
def doubleqlearning(self, episode: int = 0):
    while episode < self.num_episodes:
        print(f'EPISODE {episode}')
        if episode == 1000 and not self.test:
            self.epsilon = 0.3
        self.rpc.reset()
        self.rpc.ramp_piezo()
        if self.rpc.scan_temperature(500):
            time.sleep(10)
            try:
                self.rpc.lockCavity()
            except:
                continue
            system_unlock = False
            print(f'\tLocked at: {time.time()}')
            print(f'\tInitial temperature voltage: {self.rpc.redpitaya.ams.dac2}V')
            purple_signal, _ = self.rpc.scope()
            print(f'\tFast signal mean: {purple_signal.max()}')
            state = self._get_state_index(round_to_nearest_0_1(purple_signal.max()))
            reward = 0
            while True:
                time.sleep(1)
                # Choose action using epsilon-greedy policy
                if np.random.rand() < self.epsilon and not self.test:
                    action = np.random.choice(self.num_actions)
                    if np.random.rand() < 0.5:
                        q1_selected = True
                    else:
                        q1_selected = False
                else:
                    if np.random.rand() < 0.5:
                        action = np.argmax(self.Q1[state, :])
                        q1_selected = True
                    else:
                        action = np.argmax(self.Q2[state, :])
                        q1_selected = False
                print(f'\tAction index: {action}')
                self.rpc.setdac2(self.rpc.redpitaya.ams.dac2 + self.action_range[action])
                print(f'\tTemperature voltage: {self.rpc.redpitaya.ams.dac2}V')
                time.sleep(0.1)
                # Get the next state, reward, and system_unlock
                purple_signal, blue_signal = self.rpc.scope()
                print(f'\tFast signal mean: {purple_signal.max()}')
                if blue_signal.max() < 0.95:
                    print(f'\tLost lock at: {time.time()}')
                    system_unlock = True
                next_state = self._get_state_index(round_to_nearest_0_1(purple_signal.max()))
                reward += 1 if not system_unlock else 0
                print(f'\tState index: {next_state}')
                # Temperature
                ain0bits, = self.tmp35.getFeedback(u3.AIN(0))
                ain0Value = self.tmp35.binaryToCalibratedAnalogVoltage(ain0bits,
                                                                        isLowVoltage=False,
                                                                        channelNumber=0)
                ain2bits, = self.tmp35.getFeedback(u3.AIN(2))
                ain2Value = self.tmp35.binaryToCalibratedAnalogVoltage(ain2bits,
                                                                        isLowVoltage=False,
                                                                        channelNumber=0)
                print(f'\tTMP35 AIN0: {ain0Value}')
                print(f'\tTMP35 AIN2: {ain2Value}')

                # Update Q-values
                if not self.test:
                    if q1_selected:
                        max_next_q = self.Q1[next_state, np.argmax(self.Q2[next_state, :])]
                        self.Q1[state, action] = (self.Q1[state, action] +
                                                  self.learning_rate *
                                                  (reward
                                                   + self.discount_factor * max_next_q
                                                   - self.Q1[state, action]))
                    else:
                        max_next_q = self.Q2[next_state, np.argmax(self.Q1[next_state, :])]
                        self.Q2[state, action] = (self.Q2[state, action] +
                                                  self.learning_rate *
                                                  (reward
                                                   + self.discount_factor * max_next_q
                                                   — self.Q2[state, action]))

                state = next_state

                if system_unlock:
                    if not self.test:
                        np.save('q1_dqlearning.npy', self.Q1)
                        np.save('q2_dqlearning.npy', self.Q2)
                    episode += 1
                    break
```

5 Conclusion

Over the course of two months, I acquired a comprehensive understanding of the methodologies involved in controlling the Red Pitaya. My knowledge extended to the seamless integration of this system with other components, enabling the acquisition of signals essential for executing precision operations such as laser stabilization within the TEM00 Gaussian mode. One notable achievement was the successful implementation of a Q-Learning agent, which markedly enhanced the system's lock maintenance capabilities, resulting in a significantly reduced occurrence of unlocking events. Nevertheless, it is important to note that the evaluation of the Q-learning algorithm remains incomplete. To rectify this, I plan to conclude the testing phase in the forthcoming weeks to ensure a fair comparison between the Q-learning agent and the conventional *software-controlled* solution. It is worth mentioning that the advantages of introducing the Q-learning agent are already discernible.

It is imperative to acknowledge that the utilization of Q-learning necessitates the discretization of both actions and states, which can potentially impose limitations. A more precise degree of control could be achieved by considering these parameters in their continuous form. To pursue this avenue, alternative reinforcement learning techniques can be explored to approximate the Q-matrix and facilitate a continuous approach. One promising solution in this regard is the Deep Deterministic Policy Gradient (DDPG) method, as described in works by Lillicrap et al. (2019)[6] and Silver et al. (2014)[7]. DDPG leverages the strengths of deep neural networks and policy gradients to effectively address problems with continuous action spaces. The DDPG framework consists of two integral components:

References

[1] Richard Bellman. Dynamic programming and stochastic control processes. *Information and control*, 1(3):228–239, 1958.

[2] Eric D Black. An introduction to pound–drever–hall laser frequency stabilization. *American journal of physics*, 69(1):79–87, 2001.

[3] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

[4] Charles J. Kowalski. On the effects of non-normality on the distribution of the sample product-moment correlation coefficient. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 21(1):1–12, 1972.

[5] Dongjun Li, Vincent S. H. Lee, Yanbei Chen, and Kathryn M. Zurek. Interferometer response to geontropic fluctuations. *Phys. Rev. D*, 107:024002, Jan 2023.

[6] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[7] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. Pmlr, 2014.

[8] Student. Probable error of a correlation coefficient. *Biometrika*, 6(2/3):302–310, 1908.

[9] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[10] Gabor J. Szekely and Maria L. Rizzo. Partial distance correlation with methods for dissimilarities, 2014.

[11] Gá bor J. Székely, Maria L. Rizzo, and Nail K. Bakirov. Measuring and testing dependence by correlation of distances. *The Annals of Statistics*, 35(6), dec 2007.

[12] Gábor J. Székely and Maria L. Rizzo. Energy statistics: A class of statistics based on distances. *Journal of Statistical Planning and Inference*, 143(8):1249–1272, 2013.

[13] Erik P. Verlinde and Kathryn M. Zurek. Observational signatures of quantum gravity in interferometers. *Physics Letters B*, 822:136663, 2021.

[14] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

Algorithm 4 Q-Learning

```
procedure QLearning(episode)
    while episode < MAX_EPISODE do
        if episode == 1000 and not test then
            $\epsilon = .3$
        Reset
        RampPiezo
        if ScanTemperature(500) then
            Sleep(10)
            LockCavity
            $systemUnlock \leftarrow False$
            $purple\_signal, \_ \leftarrow$ Scope
            $state \leftarrow$ StateIndex($purple\_signal$)
            while  do
                Sleep(1)
                if Rand < $\epsilon$ and not test then
                    $action \leftarrow$ RandomChoise($Actions$)
                else
                    $action \leftarrow$ ArgMax($Q[state, :]$)
                SetDAC2($currentDAC2 + action$)
                Sleep(0.1)
                $purple\_signal, blue\_signal \leftarrow$ Scope
                if Max($blue\_signal$) < 0.95 then:
                    $systemUnlock \leftarrow True$
                $nextState \leftarrow$ StateIndex($purple\_signal$)
                if $systemUnlock$ then
                    $reward \leftarrow 1$
                else
                    $reward \leftarrow 0$
                if not test then
                    $maxNextQ \leftarrow$ Max($Q[nextState, :]$)
                    update Q
                $state \leftarrow nextState$
                if $systemUnlock$ then
                    break
```

Algorithm 5 Q-Learning

```
1: procedure QLearning(episode)
2:     while episode < MAX_EPISODE do
3:         if episode == 1000 and not test then
4:             $\epsilon = .3$
5:         Reset
6:         RampPiezo
7:         if ScanTemperature(500) then
8:             Sleep(10)
9:             LockCavity
10:            $systemUnlock \leftarrow False$
11:            $purple\_signal, \_ \leftarrow$ Scope
12:            $state \leftarrow$ StateIndex($purple\_signal$)
13:            $reward \leftarrow 0$
14:            while  do
15:                Sleep(1)
16:                if Rand $< \epsilon$ and not test then
17:                    $action \leftarrow$ RandomChoise($Actions$)
18:                    if Rand $< 0.5$ then
19:                        $q1\_selected \leftarrow True$
20:                    else
21:                        $q1\_selected \leftarrow False$
22:                else
23:                    if Rand $< 0.5$ then
24:                        $action \leftarrow$ ArgMax($Q1[state, :]$)
25:                        $q1\_selected \leftarrow True$
26:                    else
27:                        $action \leftarrow$ ArgMax($Q2[state, :]$)
28:                        $q1\_selected \leftarrow False$
29:                SetDAC2($currentDAC2 + action$)
30:                Sleep(0.1)
31:                $purple\_signal, blue\_signal \leftarrow$ Scope
32:                if Max($blue\_signal$) $< 0.95$ then:
33:                    $systemUnlock \leftarrow True$
34:                $nextState \leftarrow$ StateIndex($purple\_signal$)
35:                if $systemUnlock$ then
36:                    $reward \leftarrow reward + 1$
37:                if not test then
38:                    if $q1\_selected$ then
39:                        $maxNextQ1 \leftarrow Q1[nextState,$ Argmax$(Q2[nextState, :])]$
40:                        update Q1
41:                    else
42:                        $maxNextQ2 \leftarrow Q2[nextState,$ Argmax$(Q1[nextState, :])]$
43:                        update Q2
44:                $state \leftarrow nextState$
45:                if $systemUnlock$ then
46:                    break
```