Robotic Arm Project Document

1. Introduction

- **Project Overview**:
    - The purpose of this project is to design and build a functional robotic arm capable of performing basic pick-and-place tasks. The arm is controlled through potentiometers, allowing precise manual adjustments to servo positions. It serves as a platform for experimentation and development, featuring adaptability for integration with imitation learning models via a custom API. The project aims to demonstrate a modular approach to robotics, emphasizing ease of control, accurate motion, and scalability for more advanced automation tasks.
- **Scope**:
    - **Sorting Tasks**: The robotic arm can identify, pick, and place objects into designated categories or areas based on predefined criteria, such as size or color (when equipped with additional sensors or vision systems).
    - **Basic Manipulation**: Capable of handling objects with precision for simple tasks, such as stacking, assembling, or repositioning items.
    - **Prototyping and Development**: Serves as a platform for testing new algorithms, control methods, and hardware components in a modular and scalable manner.
    - **Integration into Larger Robotic Systems**: The arm is designed to interface seamlessly with other robotic systems, enabling collaborative automation in environments like warehouses, manufacturing lines, or research setups.
    - **Imitation Learning Models**: Equipped with an API to integrate with AI-driven systems for learning and replicating complex tasks through observation and data analysis.

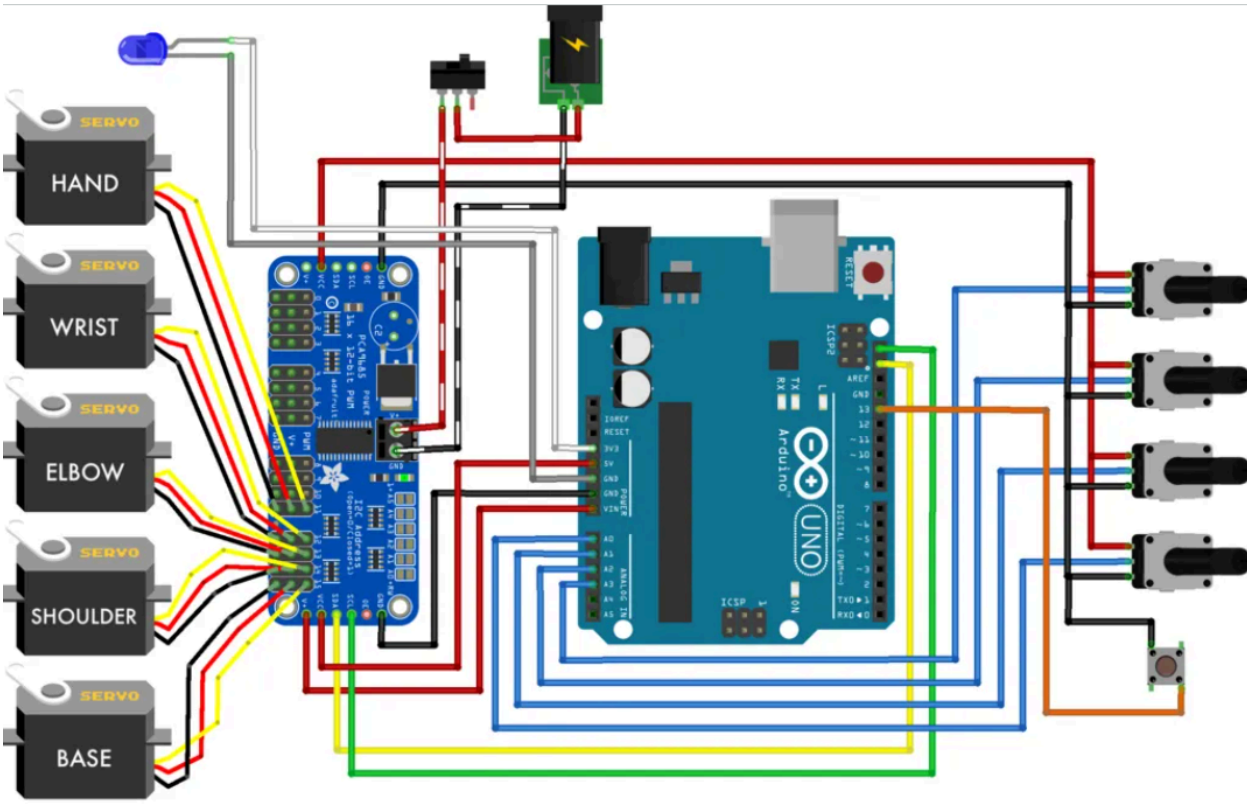
2. System Design

- **Hardware Components**:
    - List and describe all hardware used, including their specifications:
        - **Servos**: MS24 6-8.4V 20KG servos for precision control x4.
        - **Sensors**: B10K Potentiometers for positional input, Pi 1080p camera for pixel matching, Luna LiDAR for distance measurements.
        - **Microcontrollers**: Raspberry Pi and Arduino (used for prototyping and control).
        - **ADC**: ADS1115 for accurate analog-to-digital conversion.
        - **Power Supply**: 5V SHNITPWR AC/DC powersupply
        - **3D-Printed Parts**: Components designed in Fusion 360/AutoCAD and printed on the X1C printer.
- **Mechanical Design**:

- CAD Models linked here: https://www.printables.com/model/818975-compact-robot-arm-arduino-3d-printed
  Credit: Joseph Prusia

- **Electrical Design**:

  Arduino Variant: (for use with Pi connect signal pin of potentiometers to ADC input pins)

- **Software Stack**:
  - List the programming languages, libraries, and frameworks used:
    - Python for Raspberry Pi prototyping.
    - Arduino IDE and C for final integration and embedded system type architecture.

3. Functionality

- **Control System**:

  Purpose of Potentiometers:

  - Potentiometers serve as input devices to provide analog signals representing the desired position of each servo in the robotic arm.
  - By adjusting the potentiometer, the user sets a voltage level that corresponds to a specific angle or position for the servo.

Analog Signal Conversion:

- The potentiometer outputs an analog voltage (e.g., 0 to 3.3V) proportional to its rotational position.
- Since the Raspberry Pi cannot directly read analog signals, the **ADS1115 ADC (Analog-to-Digital Converter)** is used to convert the analog voltage into a digital value.

Reading Values:

- The Raspberry Pi communicates with the ADS1115 over the **I2C protocol** to fetch the converted digital values.
- Each potentiometer's output is sampled at regular intervals, providing a digital value representing its position.

Mapping Values to Servo Angles:

- The ADC provides a raw digital value (e.g., 0–32767 for 16-bit resolution).
- This value is mapped to the servo's range of motion (e.g., 0–180 degrees) using a mathematical formula:

$$\text{Servo Angle} = \frac{\text{ADC Value}}{\text{Max ADC Value}} \times \text{Max Servo Angle}$$

- For example, an ADC value of 16383 (halfway) maps to 90 degrees for a servo with a range of 0–180 degrees. However this will need to be scaled based on your remote setup, for a 1:1 response it should be noted most potentiometers have 270 degrees of motion.

Servo Control:

- The calculated angle is sent to the servo driver (e.g., PCA9685 PWM driver) using PWM (Pulse Width Modulation) signals.
- The servo then moves to the corresponding angle.

Movements:

Base Rotation (1 DOF):

- **Type**: Rotational.
- **Description**: Allows the arm to rotate horizontally around its base, enabling it to reach different areas within its operational range.
- **Axis**: Vertical (z-axis).

Shoulder Joint (1 DOF):

- **Type**: Rotational.
- **Description**: Provides up-and-down movement for the main arm segment, raising or lowering the arm to different heights.
- **Axis**: Horizontal (y-axis).

Elbow Joint (1 DOF):

- **Type**: Rotational.
- **Description**: Enables further up-and-down motion for the forearm segment, increasing the arm's reach and flexibility.
- **Axis**: Horizontal (y-axis).

Wrist Rotation (1 DOF):

- **Type**: Rotational.
- **Description**: Allows the wrist to twist, orienting the end effector (e.g., gripper or tool) for precise positioning.
- **Axis**: Longitudinal (along the forearm's length).

Wrist Flexion (Optional, 1 DOF):

- **Type**: Rotational.
- **Description**: Allows the wrist to move up and down for fine adjustments in tool or gripper positioning.
- **Axis**: Horizontal (y-axis).

Gripper or End-Effector (1 DOF):

- **Type**: Linear (or angular, depending on design).
- **Description**: Enables opening and closing (linear motion) to grasp or release objects. This could also be a rotational movement if a claw or rotating tool is used.

API Integration:

- API is currently being revised, come back later!

4. Integration Procedures

- **System Calibration - Potentiometer**:

  Measure Voltage Range:

- Connect the potentiometer to a voltage source and measure the output voltage across its full rotation using a multimeter.
- Record the minimum (V_min) and maximum (V_max) voltage values.

Determine Usable Range:

- Ensure the potentiometer covers the servo's intended motion range (e.g., 0–180°).
- If the potentiometer's range is larger or smaller than required, adjust the mapping in software.

Software Mapping:

- Use a formula to map the voltage range to a servo

$$\text{Servo Angle} = \frac{\text{Measured Voltage} - V_{\text{min}}}{V_{\text{max}} - V_{\text{min}}} \times 180$$

- Implement this calculation in your control code

Test and Verify:

- Rotate the potentiometer and verify that the mapped servo angle corresponds to the expected physical position of the servo.

Adjust Offsets (if needed):

- If there is a consistent discrepancy, adjust offsets in the code to fine-tune the mapping.

- **System Calibration - Servo**

  Define Servo Range:

  - Identify the minimum and maximum PWM pulse width (e.g., 500µs to 2500µs) corresponding to the servo's 0° and 180° positions.

  Software Configuration:

  - Map the input angle (e.g., 0–180°) to the servo's pulse width range:

$$\text{PWM Signal} = \text{Min Pulse Width} + \frac{\text{Angle}}{180} \times (\text{Max Pulse Width} - \text{Min Pulse Width})$$

  - Update this formula in the servo control software.

  Physical Testing:

    - Send commands to move the servo to 0°, 90°, and 180° positions.
    - Verify the servo reaches the expected positions accurately.

      Adjust Endpoints:

    - If the servo exceeds its physical limits or does not utilize the full range, modify the PWM values to match its mechanical constraints.

      Repeatability Test:

    - Move the servo back and forth across its range multiple times to ensure consistent accuracy.
- **System Calibration - Potentiometer**:

  Verify Voltage Input:

  - Use a multimeter to confirm the voltage input to the ADC matches the expected range (e.g., 0–3.3V).

    Check Reference Voltage:

  - Ensure the ADC's reference voltage (V_ref) is correctly set (e.g., 3.3V for Raspberry Pi systems).
  - If the ADC supports external reference voltage, confirm the hardware configuration matches your system's requirements.

    Measure ADC Output:

  - Rotate the potentiometer and record the digital values output by the ADC across its range.
  - Ensure the ADC provides values from 0 to its maximum resolution (e.g., 0–32767 for 16-bit ADC).

    Linear Mapping:

  - Map the ADC output to the potentiometer's voltage: Voltage=ADC ValueMax ADC Value×Vref\text{Voltage} = \frac{\text{ADC Value}}{\text{Max ADC Value}} \times V_{\text{ref}}Voltage=Max ADC ValueADC Value×Vref
  - Verify this calculation in the software.

    Test Consistency:

  - Rotate the potentiometer to specific positions (e.g., minimum, midpoint, maximum) and verify that the ADC provides consistent readings.

    Correct Offsets:

- If the ADC's output does not match expected values, calibrate the offsets in the code to ensure accuracy.

System-Level Calibration

Once individual components are calibrated, perform a system-level calibration:

Combine Potentiometer, ADC, and Servo:

- Test the full control loop by rotating the potentiometer and observing servo movement.
- Verify that the servo moves to the expected position based on the potentiometer's setting.

Fine-Tune Mapping:

- If the servo does not align perfectly, adjust the mapping functions or offsets in the software can also adjust the pulse width on servos.

Repeatability Test:

- Perform repetitive movements to ensure consistent behavior across multiple iterations.

Integration Testing:

- Test the system under different conditions (e.g., varying input speeds, load changes) to ensure robustness.

6. Testing and Validation

Step-by-step guide for testing the arm:

1. Initial Hardware Verification

Objective: Ensure all components are correctly assembled and connected.

- **Visual Inspection**:
  - Verify that all wiring and connections (e.g., servos, potentiometers, ADC) are secure and match the wiring diagram.
  - Check for loose or improperly soldered joints.
- **Power Check**:
  - Power on the system and confirm voltage levels at key points (e.g., servos, ADC) using a multimeter.
  - Ensure no components overheat or exhibit unusual behavior.

---

2. Individual Component Testing

Objective: Validate the functionality of each component separately.

1. **Potentiometers**:
    - Rotate each potentiometer and monitor the voltage output.
    - Use software to log ADC values and confirm they correspond to the expected voltage range.
    - Verify smooth and consistent changes in readings across the potentiometer's full rotation.
2. **Servos**:
    - Send test PWM signals to each servo to move them to their minimum, midpoint, and maximum positions.
    - Confirm that servos reach the correct physical positions without jitter or overshoot.
    - Observe any unusual noises or resistance, which may indicate mechanical issues.
3. **ADC**:
    - Read input values from the ADC for known voltages (e.g., 0V, 1.65V, 3.3V).
    - Compare digital output to expected values based on the ADC's resolution (e.g., 16-bit for ADS1115).

---

3. Control Loop Testing

Objective: Test the system's control loop, linking potentiometers to servo movement.

1. **Mapping Validation**:
    - Rotate each potentiometer and verify that the corresponding servo moves to the expected angle.
    - Check the software's mapping functions for accuracy.
2. **Full-Range Motion**:
    - Move each potentiometer through its full range and ensure the corresponding servo moves smoothly and accurately.
    - Observe for any lag, erratic movements, or deviations in servo response.
3. **Edge Case Testing**:
    - Test boundary positions (e.g., extreme potentiometer angles) to ensure the servo does not exceed its physical limits.

---

4. System Integration Testing

Objective: Verify that all components work together seamlessly.

1. **Simultaneous Movement**:
    - Test all potentiometers and servos simultaneously to ensure no interference or unexpected behavior.
    - Observe how the system performs under varying loads or conditions.
2. **Predefined Tasks**:
    - Execute a series of predefined movements (e.g., a pick-and-place sequence).
    - Record the results and check for accuracy and repeatability.

---

5. Repeatability Testing

Objective: Ensure the system performs consistently over multiple iterations.

1. **Task Repetition**:
    - Perform the same task (e.g., moving an object) multiple times.
    - Verify that the arm repeats the task accurately and without deviation.
2. **Long-Duration Test**:
    - Operate the system continuously for an extended period to identify any potential reliability issues, such as overheating or drift.

---

6. Stress and Edge Case Testing

Objective: Test the system under extreme or unexpected conditions.

1. **Stress Test**:
    - Simulate high-frequency input changes (e.g., rapid potentiometer rotations) to test responsiveness and stability.
    - Apply increased loads to the servos (e.g., lifting heavier objects) to evaluate performance under stress.
2. **Edge Case Scenarios**:
    - Provide invalid or extreme inputs (e.g., rapid changes between minimum and maximum positions) and observe how the system handles them.
    - Introduce noise or interruptions in the signal (e.g., disconnecting a potentiometer) to evaluate error handling.

---

7. Validation Against Specifications

Objective: Confirm that the system meets design and performance specifications.

1. **Accuracy**:
    - Measure the deviation between commanded and achieved positions for each servo.
    - Validate that the system operates within acceptable tolerances.
2. **Speed**:
    - Measure the time taken to execute specific movements or tasks.
    - Verify that the system meets speed requirements for real-world applications.
3. **Safety**:
    - Test fail-safes, such as stopping all servos when an emergency condition (e.g., power loss or signal interruption) occurs.

---

8. Feedback and Iteration

Objective: Identify areas for improvement and refine the system.

1. **Analyze Results**:
    - Review data from all tests and note any inconsistencies or performance issues.
    - Gather feedback from users or testers on the system's usability and functionality.
2. **Adjust and Retest**:
    - Make necessary adjustments (e.g., recalibrating components, refining code) and repeat relevant tests.

6. Troubleshooting and Debugging

- **Common Issues**:

1. ADC Inconsistencies

Symptoms:

- Unexpected fluctuations in the ADC readings (e.g., noisy or unstable output).
- Incorrect mapping of potentiometer values to servo angles.

Possible Causes:

- Electrical noise on the input signal.
- Incorrect wiring or grounding issues.

- Faulty or improperly configured ADC (e.g., wrong reference voltage).

Resolutions:

- **Use Filtering**:
  - Implement a software filter (e.g., moving average) to smooth noisy data.
- **Improve Wiring**:
  - Use shielded cables for connections between the potentiometer and ADC.
  - Ensure proper grounding of all components to avoid interference.
- **Verify ADC Configuration**:
  - Check the reference voltage and resolution settings in the software.
  - Perform calibration to ensure accurate readings.
- **Add Decoupling Capacitors**:
  - Place capacitors (e.g., 0.1 µF) near the ADC input to filter out high-frequency noise.

---

2. Servo Jitter

Symptoms:

- Servos make small, rapid, and unintended movements when holding a position.
- Inconsistent or shaky motion during transitions.

Possible Causes:

- Inaccurate or noisy PWM signals.
- Insufficient power supply to servos.
- Interference from other components (e.g., high-current devices).

Resolutions:

- **Ensure Stable PWM Signals**:
  - Use a dedicated servo driver (e.g., PCA9685) to generate precise PWM signals.
  - Avoid generating PWM signals directly from the Raspberry Pi's GPIO to reduce timing inaccuracies.
- **Check Power Supply**:
  - Use a stable and adequately rated power supply for the servos.
  - Separate the power source for servos from the control electronics to prevent voltage drops.
- **Dampen Commands**:
  - Implement a dead zone or hysteresis in the control code to avoid minor fluctuations in position.

---

3. Inaccurate Potentiometer Mapping

Symptoms:

- The servo position does not align with the expected angle based on potentiometer input.
- Significant deviation in motion at extreme potentiometer positions.

Possible Causes:

- Potentiometer output not covering the full expected voltage range.
- Calibration issues in the software mapping.

Resolutions:

- **Recalibrate**:
  - Measure the potentiometer's actual minimum and maximum output voltages and adjust the mapping in software.
- **Adjust Software**:
  - Add offsets or scale factors to align the potentiometer range with the servo range.
- **Use Higher-Quality Potentiometers**:
  - Replace worn or low-resolution potentiometers with higher-quality alternatives.

---

4. Power Supply Issues

Symptoms:

- System reboots or components fail to respond under load.
- Servos slow down or behave erratically.

Possible Causes:

- Insufficient current supplied to the servos or control electronics.
- Voltage drops due to shared power lines.

Resolutions:

- **Upgrade Power Supply**:
  - Use a power supply with sufficient current capacity to handle the load (e.g., 6–8.4V servos often require a high-amperage source).
- **Separate Power Rails**:

    - Power servos and the control system (e.g., Raspberry Pi or ADC) from independent power sources to avoid interference.
- **Add Capacitors**:
    - Use capacitors near the power inputs to smooth voltage fluctuations.

---

5. Mechanical Issues

Symptoms:

- Arm movement feels restricted or inconsistent.
- Servos produce unusual noise or strain.

Possible Causes:

- Misaligned or poorly assembled components.
- Overloading the servos with excessive weight or torque.

Resolutions:

- **Inspect Assembly**:
    - Check for loose screws, misaligned joints, or friction points in the arm.
- **Limit Load**:
    - Ensure the arm is not lifting more weight than the servos can handle.
- **Lubricate Joints**:
    - Apply appropriate lubrication to reduce friction in mechanical parts.

---

6. System Freezes or Crashes

Symptoms:

- The robotic arm stops responding during operation.
- The Raspberry Pi or microcontroller becomes unresponsive.

Possible Causes:

- Excessive CPU or memory usage.
- Faulty code or infinite loops in the software.

Resolutions:

- **Optimize Code**:
    - Profile and optimize the control software to reduce resource usage.
    - Add error handling to prevent infinite loops or unhandled exceptions.

- **Monitor System Resources**:
    - Use tools to track CPU and memory usage during operation.
- **Implement Watchdog Timers**:
    - Set up a watchdog timer to reset the system in case of a software freeze.

---

7. Communication Errors

Symptoms:

- Data transmission between components fails intermittently.
- I2C or SPI devices stop responding.

Possible Causes:

- Incorrect wiring or loose connections.
- Signal interference or timing mismatches.

Resolutions:

- **Secure Connections**:
    - Recheck and secure all I2C or SPI wiring.
- **Add Pull-Up Resistors**:
    - Ensure I2C lines (SDA, SCL) have appropriate pull-up resistors (e.g., 4.7kΩ).
- **Check Bus Timing**:
    - Confirm that all devices operate at compatible clock speeds.
    -

- **Error Logs**:
    - TODO Create repo and store error logs, Link here when completed

7. Mapping the Robotic Arm Environment to BCO

Steps in BCO Integration

Step 1: Data Collection

- **Learner Data**:
    - The robotic arm explores its environment using random or heuristic-driven actions to collect states and actions.
    - Actions are servo commands, and states are the resulting joint positions.
    - This data is stored in a dataset for training the inverse dynamic model.
- **Expert Data**:
    - Using manual control (potentiometers), an expert demonstrates tasks. Only the states (joint positions at each step) are recorded.
    - These demonstrations define the goal behavior.

Step 2: Train the Inverse Dynamic Model

- **Inputs**:
    - Learner's current state (e.g., joint positions).
    - Learner's next state (resulting from an action).
- **Output**:
    - The predicted action that caused the state transition.
- **Loss Function**:
    - Mean Squared Error (MSE) or other loss functions to compare the predicted action with the actual action taken by the learner.

Step 3: Test the Inverse Dynamic Model

- **Inputs**:
    - Expert's current state and next state (from recorded expert demonstrations).
- **Output**:
    - Predicted action corresponding to the expert's state transition.

Step 4: Train the Behavior Model (Policy)

- **Inputs**:
    - Expert's current state.
- **Output**:
    - Predicted expert action, derived from the inverse dynamic model.
- **Loss Function**:
    - Measures the error between the predicted action and the target action.

Step 5: Iterative Training and Interaction

- The robotic arm interacts with its environment using the trained behavior model (policy) to select actions based on the current state.
- New states and actions are collected during these interactions.

- Both the inverse dynamic model and behavior model are updated iteratively with this new data.

Required Adjustments

1. **Customizing for the Robotic Arm**:
    - Modify the **OpenAI Gym environment** to reflect the robotic arm's state and action space.
    - Integrate task-specific reward functions, such as completing a pick-and-place task efficiently.
2. **Hardware-Specific Inputs**:
    - Replace simulated environment data (used in Pendulum or Bipedal Walker) with real-time data from the robotic arm (e.g., potentiometer readings and servo positions).
3. **Continuous Feedback**:
    - Incorporate real-time feedback from sensors to refine the policy, especially for tasks requiring precision.
4. **API Development**:
    - Develop an API to stream state-action data to the learning model and execute the policy's predicted actions on the robotic arm.

Future Directions

- Extend the environment to include more complex tasks, such as multi-object sorting or obstacle avoidance.
- Add vision-based inputs (e.g., from cameras) to provide richer state information for advanced learning.
- Implement advanced reward functions to encourage smoother, more efficient motions.