**PlatformCalibration.py**

**Overview**

This script is a **real-time calibration and testing application** for the Mdx motion platform controlled via Festo **pneumatic artificial muscles**. The application integrates **hardware interfaces, kinematics, and a graphical user interface (GUI) using PyQt5**. The platform collects sensor data, maps distances to pressures, and allows for manual and automated control of actuator movements.

The main functionalities include:

1. **Serial Communication** with encoders, (IMU, scale, and a servo model not needed)
2. **Kinematics & Dynamics** calculations to determine actuator behavior.
3. **Graphical User Interface (GUI)** for controlling and monitoring the system.
4. **Calibration Procedures** for mapping actuator movements to pressure values.
5. **Data Logging & Processing** to store captured sensor data for further analysis.

**Key Functional Components**

The script is divided into the following modules:

**1. GUI Initialization (MainWindow Class)**

* Loads a UI from a Qt Designer file (calibration\_gui.ui).
* Initializes **PyQt timers** to handle periodic updates.
* Sets up **signal connections** for UI buttons and checkboxes.

**2. Serial Communication (configure\_serial())**

* Uses the SerialContainer class from common.serialSensors to manage:
  + **Encoders** (to measure actuator positions)

*Following not needed for creating D to P files*

* + ***- IMU*** *(for roll, pitch, and yaw)*
  + ***- Scale*** *(to read applied loads)*
  + ***-ServoModel*** *(to communicate with a servo-based model)*
* Assigns default communication ports and baud rates.
* Provides **start/stop functions** to open/close serial connections dynamically.

**3. Kinematics & Dynamics Setup (configure\_kinematics())**

* Uses the Kinematics and Dynamics classes to:
  + Define the platform’s **geometry and constraints**.
  + Compute **inverse kinematics** for mapping actuator movements.
  + Process **distance-to-pressure mappings** (D\_to\_P class).

**4. Data Capture & Processing**

* **Buffers** (reset\_buffers()) store captured data:
  + Time, actuator positions, target and actual pressures, IMU readings.
* **Data capture is controlled** via start\_capture() and stop\_capture().
* **Calibration data** is stored in step sequences (calibrate()).

**5. Platform Control & Movement**

* **Manual and automated control** of actuator movement:
  + move(): Moves actuators based on user-defined input.
  + move\_actuator(): Moves a specific actuator.
  + step\_platform(): Moves the platform in controlled steps during calibration.
* Sends calculated **pressure values** to Festo controllers (muscle\_output).

**6. Calibration & Lookup**

* **Calibration (calibrate())**:
  + Steps through pressure values and records actuator responses.
  + Saves calibration data for later use.
* **Lookup Table Generation (run\_lookup())**:
  + Determines **closest matching** pressure-distance curves for a given load.

**7. File Handling & Data Logging**

* **Saves calibration results** (save\_step\_data()).
* **Merges multiple calibration datasets** (merge\_d\_to\_p()).
* **Processes and generates lookup tables** (create\_d\_to\_p()).
* **Raw data logging** (save\_raw\_data()).

**8. Emergency Stop & Safety**

* Implements **emergency stop functionality** (estop()):
  + Halts platform movement.
  + Disables ongoing calibration or capture operations.

**9. Logging & Command Line Arguments**

* Uses Python's logging module to store logs in PlatformCalibration.log.
* Parses **command-line arguments** for:
  + Log level (-l option)
  + Custom IP for Festo controller (-f option).

**Execution Flow**

1. **Application starts** and initializes the PyQt GUI.
2. **Serial devices are detected and configured**.
3. **User selects a calibration routine or manual movement**.
4. **Encoders, IMU, and Festo controller provide real-time feedback**.
5. **Calibration data is captured and stored** for further analysis.
6. **Lookup tables are generated** to map actuator movement to pressures.
7. **User can manually move the platform** using UI controls.
8. **Emergency stop halts movement** if necessary.

**Suggestions to modify code for a single muscle**

To modify the code to support **only one muscle and encoder (instead of six)**, you will need to make changes in multiple areas where the **assumption of six muscles/encoders is hardcoded**. Below are the key areas to modify:

**1. Adjust Constants for Single Muscle**

**File-Wide Constants**

* Change the **hardcoded lists** and arrays that assume six muscles to **handle only one muscle**.
* Define a **single-motor mode flag** for conditional logic.

NUM\_MUSCLES = 1 # Change from 6 to 1

**2. Modify GUI Elements**

**Encoders and Pressure Displays**

* The UI currently displays six encoders and pressure bars.
* Modify configure\_festo\_info() to only use **one muscle pressure bar**:

def configure\_festo\_info(self):

self.pressure\_bars = [self.ui.muscle\_0] # Keep only one

self.actual\_bars = [self.ui.actual\_0]

self.txt\_muscles = [self.ui.txt\_muscle\_0]

self.txt\_muscles[0].setText('?')

self.ui.chk\_festo\_actuals.stateChanged.connect(self.festo\_check)

* Remove or hide extra UI elements in calibration\_gui.ui.

**3. Update Serial Communication Handling**

**Change Encoder Setup (configure\_serial())**

* The encoder\_directions list assumes six encoders.
* Change it to a single value:

encoder\_directions = [-1] # Only one muscle now

* Ensure SerialContainer instances are adjusted:

self.encoder = SerialContainer(Encoder(), self.ui.cmb\_encoder\_port, "*encoder", self.ui.lbl\_encoders, 115200)*

**4. Adjust Data Buffers**

**Modify Arrays in reset\_buffers()**

Change **multi-muscle data buffers** to **single-muscle storage**:

def reset\_buffers(self):

self.distances = [] # Encoder readings (was 6, now 1)

self.target\_pressures = [] # Pressures sent to Festo (was 6, now 1)

self.pressure\_deltas = [] # Difference between commanded and actual pressure

self.imu\_data = [] # Roll, pitch, yaw

self.time = [] # Time stamps

log.info("Buffers reset")

**5. Modify Calibration Steps**

**Update calibrate()**

* The code loops over 6 muscles when logging actuator movement.
* Modify calibrate() to handle a **single pressure value** instead of a list:

pressures = [int(pressure)] # Only one muscle instead of six

self.muscle\_output.send\_pressures(pressures)

**6. Change Data Capture (data\_update())**

* The script currently stores **six encoder readings** per cycle.
* Modify it to **store a single encoder value**:

def data\_update(self):

encoder\_data, timestamp = self.encoder\_update()

if encoder\_data and timestamp != 0:

self.distances = [encoder\_data[0]] # Store only the first value

if self.is\_capturing\_data:

self.time.append(timestamp)

self.target\_pressures.append(self.muscle\_output.festo.out\_pressures[0])

delta = self.muscle\_output.in\_pressures[0] - self.muscle\_output.festo.out\_pressures[0]

self.pressure\_deltas.append(delta)

**7. Update Motion Control**

**Modify move()**

* Instead of controlling six actuators, now control **only one**:

def move(self):

if self.is\_calibrating or self.estopped:

print("Manual mode disabled while another activity is active")

else:

percent = self.ui.sld\_percent.value()

request = [percent \* 0.01] # Only one muscle

self.muscle\_output.move\_percent(request)

**Modify step\_platform()**

* The function currently **sends six pressure values**.
* Change it to send **one**:

pressures = [int(pressure)] # Send only one pressure value

**8. Modify Distance-to-Pressure Mapping (D\_to\_P)**

**Fix Lookup Table Processing**

* The D\_to\_P class processes six distances.
* Modify it to handle **only one value**.

Before:

self.DtoP.set\_index(up\_pressure, encoder\_data, 'up')

After:

self.DtoP.set\_index(up\_pressure, [encoder\_data[0]], 'up') # Use only one encoder

**9. Adjust Data Saving (save\_step\_data())**

* The calibration data files (DtoP\_\*.csv) currently expect six muscle values.
* Update save\_step\_data() to only save **one column of distances**.

self.outfile.write("cycle,dir,step,pressure,d0,t0\n")

for step in self.step\_data:

data = step[:4] + [step[4][0]] + [step[5][0]] # Extract only the first value

line = ','.join(str(n) for n in data)

self.outfile.write(line + "\n")

**10. Remove Unused Elements**

**Modify UI Layout**

* Remove unnecessary buttons and displays.
* Modify calibration\_gui.ui to only **show one pressure bar and encoder value**.

**Summary of Changes**

| **Section** | **Change** |
| --- | --- |
| **Constants** | Change NUM\_MUSCLES = 1 |
| **GUI Elements** | Remove extra encoder and pressure displays |
| **Serial Setup** | Change encoder\_directions to [-1] |
| **Data Buffers** | Change distances = [] to store only one value |
| **Calibration** | Modify calibrate() to process a single actuator |
| **Motion Control** | Change move() and step\_platform() to handle one muscle |
| **Lookup Tables** | Modify D\_to\_P class to process single values |
| **Data Saving** | Update save\_step\_data() to log a single column of data |