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