Internship Report - EMG-Robot Project at Robokind

The following report describes the activities undertaken during the internship in the EMG-Robot project conducted at Robokind. The main objective of the project is to record, preprocess, and real-time process data from interconnected Myoware EMG sensors via the I2C protocol. Also to train a neural network to estimate arm positions from EMG signals and utilize these estimates to control a robotic arm.

First and second week

In the first and second week of my internship, I was introduced to the participants of the EMG-Robot project at Robokind. I received a presentation on the robots and the structure of the Raspberry Pi, which is the platform used in the project. Additionally, I was granted access to the following repositories on GitHub:

1. [EMG-Robot]: This repository contains the source code of the system, including algorithms for recording, preprocessing, and real-time processing of EMG sensor data, as well as the implementation of the neural network for estimating arm positions. The code was implemented by Mohammed Fakih

2. [EMG-Robot-Dataset]: This repository contains relevant datasets for the project, allowing for the study and analysis of EMG signals captured in different scenarios.

Script Overview

Main.py

The `main.py` file handle tasks related to the EMG-Robot project. It is a command-line interface (CLI) that allows users to perform different actions by passing appropriate arguments. Here is what script is doing:

Function Definitions: Several functions (`do_recording`, `do_preprocessing`, `do_training`, `do_simulate`, `do_fake_control`, `do_control_direct`) are defined to perform specific tasks related to the EMG-Robot project.

Argument Parsing:- The script uses the `ArgumentParser` class from the `argparse` module to parse command-line arguments passed to the script when it is executed.

Argument Specification- Command-line arguments are defined using the `add_argument` method of `ArgumentParser`. These arguments include options to perform actions such as recording, preprocessing, training, simulating, and controlling the robot.

Parsing Command-Line Arguments: The `parse_args()` method is called on the `ArgumentParser` object to parse the command-line arguments provided by the user.

Execution of Actions: Depending on the arguments provided, the script executes the corresponding function to perform the desired action. For example:

- If `-r` or `--record` argument is provided, the `do_recording` function is called to start recording samples from the EMG sensors.
- If `-p` or `--preprocess` argument is provided, the `do_preprocessing` function is called to preprocess a folder of previously recorded EMG data.
- If `-t` or `--train` argument is provided, the `do_training` function is called to train an AI model on the preprocessed data.
- If `-s` or `--simulate` argument is provided, the `do_simulate` function is called to simulate the classification of live EMG data without a robot attached.
- If `-f` or `--fake` argument is provided, the `do_fake_control` function is called to use a fake controller for testing purposes.
- If `-c` or `--control` argument is provided, the `do_control_direct` function is called to use a trained AI model to classify live EMG data and control a robot.

Data Directory Specification: The `data_dir` argument specifies the base folder from which to load and store data related to the EMG-Robot project.

Here is the `Main.py` with more detailed information:

```
import os
from argparse import ArgumentParser
```

- `***import os***`: Imports the `os` module, which provides functions for interacting with the operating system.
- `**from argparse import ArgumentParser**`: Imports the `ArgumentParser` class from the `argparse` module, which is used for parsing command-line arguments.

```
def do_recording(args):
    # from emg_robot.recording import
    # TODO
    raise NotImplementedError()
```

- `**def do_recording(args):**`: Defines a function called `do_recording` that accepts an argument `args`.
- `**raise NotImplementedError()**`: Raises a `NotImplementedError` exception, indicating that this function needs to be implemented. It serves as a placeholder to indicate that this code snippet is incomplete.

```
def do_preprocessing(args):
    from emg_robot.preprocess import process_recordings

    process_recordings(os.path.join(args.data_dir, 'recordings/'),
                       os.path.join(args.data_dir, 'preprocessed/'))
```

- `**def do_preprocessing(args)**:`: Defines a function called `do_preprocessing` that accepts an argument `args`.
- `**from emg_robot.preprocess import process_recordings**`: Imports the `process_recordings` function from the `emg_robot.preprocess` module.
- `**process_recordings(os.path.join(args.data_dir, 'recordings/'), os.path.join(args.data_dir, 'preprocessed/'))**`: Calls the `process_recordings` function with two arguments, which are paths to the input and output data directories.

```
def do_training(args):
    from emg_robot.learn import load_data, save_model, train

    data = load_data(os.path.join(args.data_dir, 'preprocessed/'))
    model = train(data)
    model_file = save_model(model, args.data_dir)

    print(f'Saved model to {model_file}')
```

- `**def do_training(args):**`: Defines a function called `do_training` that accepts an argument `args`.
- `**from emg_robot.learn import load_data, save_model, train**`: Imports the `load_data`, `save_model`, and `train` functions from the `emg_robot.learn` module.
- `**data = load_data(os.path.join(args.data_dir, 'preprocessed/'))**`: Loads preprocessed data using the `load_data` function.
- `**model = train(data)**`: Trains a model with the loaded data using the `train` function.
- `**model_file = save_model(model, args.data_dir)**`: Saves the trained model to a file using the `save_model` function.

- `**print(f'Saved model to {model_file}')**`: Prints the message indicating where the model was saved.

```
def do_simulate(args):
    # TODO
    raise NotImplementedError()
```

- `**def do_simulate(args):**`: Defines a function called `do_simulate` that accepts an argument `args`.
- `**raise NotImplementedError()**`: Raises a `NotImplementedError` exception, indicating that this function needs to be implemented. It serves as a placeholder to indicate that this code snippet is incomplete.

```
def do_fake_control(args):
    from emg_robot.control import start_gui
    from emg_robot.control.controller_direct_fake import
DirectControllerFake
    from emg_robot.defaults import I2C_ADDRESSES, ROBOT_IP,
EMG_CHANNEL_NAMES

    ctrl = DirectControllerFake(I2C_ADDRESSES, ROBOT_IP)
    start_gui(ctrl, EMG_CHANNEL_NAMES)
```

- `**def do_fake_control(args):**`: Defines a function called `do_fake_control` that accepts an argument `args`.
- `**from emg_robot.control import start_gui**`: Imports the `start_gui` function from the `emg_robot.control` module.
- `**from emg_robot.control.controller_direct_fake import DirectControllerFake**`: Imports the `DirectControllerFake` class from the `emg_robot.control.controller_direct_fake` module.
- `**from emg_robot.defaults import I2C_ADDRESSES, ROBOT_IP, EMG_CHANNEL_NAMES**`: Imports constants from the `emg_robot.defaults` module.
- `**ctrl = DirectControllerFake(I2C_ADDRESSES, ROBOT_IP)**`: Creates an instance of `DirectControllerFake`.
- `**start_gui(ctrl, EMG_CHANNEL_NAMES)**`: Starts the graphical user interface with the fake controller.

```
def do_control_direct(args):
    from emg_robot.control import start_gui
    from emg_robot.control.controller_direct import DirectController
    from emg_robot.defaults import I2C_ADDRESSES, ROBOT_IP,
EMG_CHANNEL_NAMES

    ctrl = DirectController(I2C_ADDRESSES, ROBOT_IP)
    start_gui(ctrl, EMG_CHANNEL_NAMES)
```

- `**def do_control_direct(args)**:`: Defines a function called `do_control_direct` that accepts an argument `args`.

- `**from emg_robot.control import start_gui**`: Imports the `start_gui` function from the `emg_robot.control` module.
- `**from emg_robot.control.controller_direct import DirectControlle**r`: Imports the `DirectController` class from the `emg_robot.control.controller_direct` module.
- `**from emg_robot.defaults import I2C_ADDRESSES, ROBOT_IP, EMG_CHANNEL_NAMES**`: Imports constants from the `emg_robot.defaults` module.
- `ctrl = DirectController(I2C_ADDRESSES, ROBOT_IP)`: Creates an instance of `DirectController`.
- `**start_gui(ctrl, EMG_CHANNEL_NAMES)**`: Starts the graphical user interface with the direct controller.

```
if __name__ == '__main__':
    parser = ArgumentParser()

    parser.add_argument("-r", "--record", dest="do_recording",
                        help="Start recording samples from the EMG sensors
until XXX")
    parser.add_argument("-p", "--preprocess", dest="do_preprocessing",
action='store_true',
                        help="Preprocess a folder of previously recorded
EMG data "
                             "(load data from data_dir/recordings/ and save
data to data_dir/preprocessed/)")
    parser.add_argument("-t", "--train", dest="do_training",
action='store_true',
                        help="Train an AI model on the preprocessed data "
                             "(load data from data_dir/preprocessed and
save model to data_dir/)")
    parser.add_argument("-s", "--simulate", dest="do_simulate",
action='store_true',
                        help="Use a trained AI model to classify live EMG
data without a robot attached "
                             "(load model from data_dir/)")
    parser.add_argument("-f", "--fake", dest="do_fake_control",
action='store_true',
                        help="For testing: use a fake controller (no EMG
readings, direct control GUI)")
    parser.add_argument("-c", "--control", dest="do_control",
action='store_true',
                        help="Use a trained AI model to classify live EMG
data and control a robot "
                             "(load model from data_dir/)")
    parser.add_argument("data_dir",
                        help="The base folder to load from and store data
```

```
in")

    args = parser.parse_args()

    if args.do_recording:
        do_recording(args)

    elif args.do_preprocessing:
        do_preprocessing(args)

    elif args.do_training:
        do_training(args)

    elif args.do_simulate:
        do_simulate(args)

    elif args.do_fake_control:
        do_fake_control(args)

    elif args.do_control:
        do_control_direct(args)
```

- `**if __name__ == '__main__':**`: Checks if the script is being run as the main program.
- `**parser = ArgumentParser()**`: Creates an `ArgumentParser` object to parse command-line arguments.
- `**parser.add_argument(...)**`: Adds arguments and options to the argument parser.
- `**args = parser.parse_args()**`: Parses the command-line arguments.
- The remaining code checks which arguments were passed and calls the corresponding functions based on those arguments. Each function is called according to the option passed on the command line. For example, if the `-p` or `--preprocess` argument is passed, the `do_preprocessing` function is called. The `data_dir` directory is also passed as an argument to the functions.

---

emg_robot

---

control

---

controller_direct_fake.py

The `controller_direct_fake.py` file provides a fake implementation of the `DirectController` class, primarily intended for testing purposes. Let's break down the key components and differences compared to the original `controller_direct.py`:

1. Purpose:

- `controller_direct_fake.py`: This file provides a fake implementation of the `DirectController` class, primarily used for testing and development without relying on actual hardware or dependencies that may not be available in a testing environment.
- `controller_direct.py`: This file contains the original implementation of the `DirectController` class, which interfaces with real hardware components and may require specific dependencies such as the `smbus` module.

2. Implementation:

- `controller_direct_fake.py`: The `DirectControllerFake` class extends the `DirectController` class and overrides certain methods such as `calc_features` and `run_once` to provide fake implementations. It generates fake EMG signal values and calculates fake pitch and roll angles based on predefined weights and thresholds.
- `controller_direct.py`: The `DirectController` class implements the actual control logic for interfacing with EMG sensors and the robot arm, including methods for reading EMG signals, processing features, and controlling the robot arm based on the extracted features.

3. Dependency Handling:

- `controller_direct_fake.py`: This file avoids dependencies that may not be available or feasible in a testing environment. For example, it does not rely on the `smbus` module for interacting with I2C devices, which allows for easier testing without actual hardware.
- `controller_direct.py`: The original implementation may rely on external dependencies such as the `smbus` module for communication with I2C devices, which may require actual hardware and specific configurations to function correctly.

4. Testing and Debugging:

- `controller_direct_fake.py`: Provides a simplified and controlled environment for testing and debugging the control logic without the need for real hardware. It allows developers to simulate EMG signals and observe the resulting robot arm movements in a controlled manner.

- `controller_direct.py`: Implements the actual control logic for interfacing with real hardware components such as EMG sensors and the robot arm. It is used in a production environment to control the robot arm based on real-time EMG signals.

`controller_direct_fake.py` serves as a mock implementation of the `DirectController` class for testing and development purposes, while `controller_direct.py` contains the original implementation used in a production environment for interfacing with real hardware components.

---

controller_direct.py

The `controller_direct.py` file contains the `DirectController` class, which is responsible for controlling the robot arm based on EMG signals in a direct manner without using a machine learning model. `controller_direct.py` provides a direct interface for controlling the robot arm based on EMG signals without the need for a machine learning model. It preprocesses EMG signals, calculates features, applies weights and thresholds, and controls the robot arm accordingly.

Here's an explanation of the code:

1. Imports:

- The file imports necessary modules such as `sys` and `numpy`.

- It also imports classes and functions from other modules within the project (`EMGReader` and `RobotInterface`) and from the `defaults` module (`I2C_ADDRESSES`, `ROBOT_IP`).

2. Activation Functions:

- Two activation functions are defined:
  - `act_linear`: This function returns the input values as they are.
  - `act_sigmoid`: This function applies the sigmoid function to the input values to squash them between -1 and 1.

3. DirectController Class:

- This class initializes and controls the direct interaction between the EMG signals and the robot arm.
- It accepts various parameters related to EMG signal processing, robot control, and controller settings.
- The `calc_features()` method preprocesses the EMG signals, calculates features, applies weights and thresholds, and calculates pitch and roll values for controlling the robot arm.
- The `run_once()` method reads EMG signals, processes them, calculates pitch and roll values, and moves the robot arm accordingly.
- The `run()` method runs the controller continuously.
- The `emg_activity()` method returns the latest EMG activity.
- The `stop()` method stops the controller.

4. Main Block:*

- The `__name__ == '__main__'` block initializes a `DirectController` object and runs the controller.
- It handles exceptions such as KeyboardInterrupt and exits the program gracefully.

The key difference between **controller_direct.py** and c**ontroller_model.py** lies in their approach to interpreting EMG signals and generating joint movements for the robot arm. The former employs a direct mapping approach with predefined weights and thresholds, while the latter utilizes a machine learning model for more complex signal interpretation and control.

---

controller_model.py

controller_model.py defines the ModelController class responsible for reading EMG signals, preprocessing them, extracting features, predicting arm angles using an AI model, and controlling the robot arm based on the predictions. It provides methods to run and stop the controller, along with a main block to instantiate and start the controller.

Let's break down the `controller_model.py` file:

1. Import Statements:

- Import necessary modules and classes including `sys`, `numpy` (`np`), `pywt`, `EMGReader`, `RobotInterface`, and constants `I2C_ADDRESSES`, and `ROBOT_IP`.

2. AI_MODEL_PATH Constant:

- Path to the AI model file. This needs to be provided. Currently set as a placeholder with a TODO comment.

3. calc_features Function:

- Function to calculate features from the input EMG signal.
- Calls functions from the `all_features` list defined in `preprocess.py`.

4. ModelController Class

- Initializes the Model Controller with necessary parameters and objects.
- Reads EMG signals, preprocesses them, extracts features, predicts arm angles using the AI model, and moves the robot accordingly.
- Includes methods `run_once()` to execute a single iteration of the controller and `run()` to continuously run the controller.

5. run_once Method:
- Reads EMG signals from the EMG reader.
- Preprocesses the signals using Butterworth filter and wavelet decomposition.
- Calculates features from the decomposed signals.
- Predicts arm angles using the AI model.
- Moves the robot arm based on the predicted angles.
- Clears a portion of the EMG buffer to continue collecting data.

6. run Method:
- Starts the controller in a loop, continuously running `run_once()` until stopped.

7. stop Method:
- Stops the controller loop by setting `running` flag to False.

8. Main Block:
- Instantiates the `ModelController` with the specified parameters and starts running it.
- Handles KeyboardInterrupt to gracefully terminate the controller when interrupted by the user.
- Handles other exceptions and exits with an error message if any occurs.

---

direct_gui.py

The `direct_gui.py` file contains a GUI (Graphical User Interface) implemented using tkinter, a Python library for creating GUIs.The `direct_gui.py` file defines a GUI for controlling a direct controller, allowing users to adjust weights and thresholds for muscle activity in real-time using sliders and spinboxes. The GUI provides visual feedback on muscle activity and allows users to start and stop the controller loop.

1. Import Statements: Imports necessary modules and libraries including `Thread` for multithreading, `tkinter` for GUI components, and other modules from the project such as `DirectController` and `EMG_CHANNEL_NAMES` from `controller_direct.py`.

2. `create_weight_row` Function: - A utility function to create a row in the GUI containing sliders, spinboxes, and progress bars to control weights and thresholds for muscle activity.

3. `DirectControllerGUI` Class: Inherits from `tk.Tk`, the main application window class in tkinter. - Initializes the GUI window and sets up its layout, including tabs for pitch and roll controls, sliders, spinboxes, and buttons.
- Creates GUI elements to control pitch and roll factors, muscle weights, and thresholds.
- Implements methods for starting and stopping the controller loop, handling GUI events, and updating GUI elements.

4. `on_start_stop` Method: - Callback method for the start/stop button in the GUI.
- Starts or stops the controller loop and updates the GUI accordingly.

5. `on_closing` Method: - Callback method for closing the GUI window.
- Stops the controller loop before closing the window.

6. `start_controller` Method:

- Starts the controller loop in a separate thread.

7. `stop_controller` Method:
- Stops the controller loop and joins the thread.

8. `control_loop` Method:
- Main loop of the controller running in a separate thread.
- Continuously updates controller parameters based on GUI inputs and updates GUI elements to reflect the controller's activity.

9. `start_gui` Function:
- Initializes and starts the GUI with the provided controller and channel names.

10. `__name__ == '__main__'` Block:
- Creates an instance of `DirectController` and starts the GUI with the controller and channel names.

---

emg_reader.py

The `emg_reader.py` file contains two classes: **`EMGBuffer`** and **`EMGReader`**, which are used for buffering and reading EMG (Electromyography) data, respectively.

EMGBuffer` Class:

1. `*__init__(self, shape)*`: Initializes an instance of `EMGBuffer` with a buffer of zeros of the specified shape, a position pointer (`pos`), and variables to track the average time interval between data points (`dt_avg`), the second moment (`m2`), and the total number of data points (`n`).

2. `*append(self, values, dt)*`: Appends new values to the buffer along with the time interval `dt` between consecutive data points. It updates the buffer, position pointer, and computes the updated average time interval and second moment.

3. `*values(self)*`: Returns the values currently stored in the buffer.

4. `*dt_avg(self)*`: Returns the average time interval between data points.

5. `*dt_var(self)*`: Returns the variance of the time intervals between data points.

6. `*sampling_rate(self)*`: Calculates and returns the sampling rate based on the number of data points and the average time interval.

7. `*is_window_full(self, window_length_s)*`: Checks if the buffer contains enough data points to fill a window of the given length in seconds.

8. `*clear(self, keep_ratio=0.0)*`: Clears the buffer while optionally retaining a specified ratio of the most recent data points.

9. `*reset(self)*`: Resets the buffer, clearing all data and resetting the time-related variables.

10. `*__sizeof__(self)*`: Returns the size of the buffer.

11. `*__getitem__(self, key)*`: Gets items from the buffer using indexing.

`EMGReader` Class:

1. `*__init__(self, buffer_size, channels)*`: Initializes an instance of `EMGReader` with the specified buffer size and channels to read data from. It also initializes an `EMGBuffer` object to store the read data.

2. `*read(self)*`: Reads data from the specified channels using the `smbus` library, appends it to the buffer, and updates the last read time.

3. `*sampling_rate(self)*`: Gets the sampling rate from the associated buffer.

4. `*clear(self, keep_ratio=0.0)*`: Clears the associated buffer while optionally retaining a specified ratio of the most recent data points.
5. `*has_full_window(self, window_length_s)*`: Checks if the associated buffer contains enough data points to fill a window of the given length in seconds.
6. `*get_samples(self)*`: Gets the samples from the associated buffer.
7. `*__getitem__(self, key)*`: Gets items from the associated buffer using indexing.

---

robot.py
This code defines a `RobotInterface` class that interacts with a robot via the `frankx` library. This `RobotInterface` class provides an interface for controlling a robot's arm motion and ensuring that the motion stays within specified limits.
1. `def __init__(self, ip, dynamic_limit_rel=0.1, joint_change_limit_rad=0.05) -> None:`: This is the constructor method for the `RobotInterface` class. It initializes the class with the robot's IP address (`ip`), the relative dynamic limits of the robot's motion (`dynamic_limit_rel`), and the joint change limit in radians (`joint_change_limit_rad`).
2. `global frankx`: This line declares the `frankx` library as a global variable, indicating that it will be used within the class.
3. `import frankx`: This line imports the `frankx` library, which is used for robot control.
4. `self.ip = ip`: This line assigns the provided IP address to the `ip` attribute of the `RobotInterface` instance.
5. `self.robot = frankx.Robot(ip)`: This line creates a `Robot` object from the `frankx` library using the provided IP address.
6. `self.robot.set_dynamic_rel(dynamic_limit_rel)`: This line sets the dynamic limits (velocity, acceleration, jerk) of the robot's motion relative to its maximum values.
7. `self.joint_change_limit_rad = joint_change_limit_rad`: This line assigns the provided joint change limit in radians to the `joint_change_limit_rad` attribute of the `RobotInterface` instance.
8. `def limit_joint_motion(self, curr, new):`: This method limits the motion of a joint based on its current position (`curr`) and the desired new position (`new`). It ensures that the joint does not move beyond a certain limit (`joint_change_limit_rad`).
9. `def move(self, pitch, roll, relative=False):`: This method moves the robot's arm to a specified pitch and roll position. If `relative` is set to `True`, the pitch and roll values are interpreted as relative offsets from the current position. It first checks if the robot is currently moving, and then adjusts the joint angles (`pitch` and `roll`) accordingly within the specified limits. Finally, it attempts to move the robot to the new joint angles using the `frankx` library.

__init__.py
This `__init__.py` file inside the `EMG-Robot/control/` directory serves as an initializer for the package. This `__init__.py` file is responsible for organizing imports within the `control` package. It imports various classes and functions from different modules within the package, making them accessible to other modules that import from the `control` package.
1. `from .controller_direct import DirectController`: This line imports the `DirectController` class from the `controller_direct` module located in the same directory as this `__init__.py` file. The `.` indicates the current package.

2. `from .controller_model import ModelController`: This line imports the `ModelController` class from the `controller_model` module located in the same directory as this `__init__.py` file.
3. `from .emg_reader import EMGBuffer, EMGReader`: This line imports the `EMGBuffer` and `EMGReader` classes from the `emg_reader` module located in the same directory as this `__init__.py` file.
4. `from .robot import RobotInterface`: This line imports the `RobotInterface` class from the `robot` module located in the same directory as this `__init__.py` file.
5. `from .direct_gui import DirectControllerGUI, start_gui`: This line imports the `DirectControllerGUI` class and the `start_gui` function from the `direct_gui` module located in the same directory as this `__init__.py` file.

---

learn

---

trainer.py
The `trainer.py` file contains functions and classes related to training a machine learning model for processing EMG (Electromyography) data.
1. Imports:
- The file begins with necessary imports from standard libraries such as `os`, `re`, `time`, and `datetime`, as well as external libraries including `numpy`, `pandas`, and `torch` (PyTorch).
2. **Constants and Hyperparameters:**
- Constants and hyperparameters such as `BATCH_SIZE`, `SEQ_LENGTH`, `NUM_LAYERS`, `HIDDEN_SIZE`, `OUTPUT_SIZE`, `NUM_FEATURES`, `EMG_CHANNELS`, and `WT_DECOMPOSITIONS` are defined. These parameters control the architecture and behavior of the neural network model.
3. AIModel Class:
- `AIModel` is a subclass of `torch.nn.Module` representing the neural network model. It contains layers for processing input data and generating output predictions.
- The `forward` method defines the forward pass of the model, which takes input data and produces output predictions.
4. load_data Function:
- The `load_data` function loads EMG data and ground truth orientation data from files stored in a specified directory.
- It organizes the data into a `TensorDataset` object, which is compatible with PyTorch's data loading utilities.
5. train Function:
- The `train` function trains the AI model using the provided data.
- It initializes the model, defines the loss function and optimizer, and iterates through epochs to update model parameters based on training data.
- During each epoch, the function prints the loss and progress of the training process.
6. save_model and load_model Function:
- `save_model` saves the trained model to a specified directory using the current timestamp as part of the filename.
- `load_model` loads a trained model from a specified file path and returns the model.

These functions and the `AIModel` class collectively provide the functionality to train, save, and load a neural network model for processing EMG data in the context of controlling a robot arm. Here is `trainer.py` file line by line:

```
import os
import re
import time
from datetime import datetime
from collections import OrderedDict
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset
from ..preprocess.features import all_features
```

- This section imports necessary modules and libraries for data processing, neural network modeling, and dataset handling. Notable imports include:
  - `torch`: PyTorch library for deep learning.
  - `torch.nn`: PyTorch's neural network module.
  - `TensorDataset` and `DataLoader`: PyTorch utilities for handling datasets.
  - `numpy` and `pandas`: Libraries for numerical computing and data manipulation.
  - `datetime`: Library for handling dates and times.
  - `OrderedDict`: Dictionary subclass that remembers the order in which its contents are added.
  - `all_features`: A variable from the `preprocess.features` module containing a list of all features used in the model.

```
BATCH_SIZE = 4  # Number of data batches when training (should be a power of 2)
SEQ_LENGTH = 5  # The number of EMG windows to consider (i.e. how far back the RNN will remember)
NUM_LAYERS = 1  # How many stacks the model should have
HIDDEN_SIZE = 128  # TODO tune (should be a power of 2?)
OUTPUT_SIZE = 2  # pitch & roll of the forearm

NUM_FEATURES = len(all_features)
EMG_CHANNELS = 5  # Number of EMG channels
WT_DECOMPOSITIONS = 3  # level 2 wavelet
NUM_INPUT_FEATURES = EMG_CHANNELS * NUM_FEATURES * WT_DECOMPOSITIONS
```

- Here, various hyperparameters and constants for the model are defined. These include batch size, sequence length, number of layers, hidden size, output size, number of features, number of EMG channels, wavelet decompositions, and number of input features.

```
class AIModel(torch.nn.Module):
    def __init__(self) -> None:
```

```
        super().__init__()

        # See https://pytorch.org/docs/stable/generated/torch.nn.RNN.html
        # TODO input size must take ignored features into account
        self.model = nn.LSTM(input_size=NUM_INPUT_FEATURES,
hidden_size=HIDDEN_SIZE, num_layers=NUM_LAYERS, bidirectional=True,
batch_first=True)
        self.fc = nn.Linear(2 * HIDDEN_SIZE, OUTPUT_SIZE)
        self.device = 'cpu'
```

- The `AIModel` class is defined as a subclass of `torch.nn.Module`, which represents the neural network model.
- The `__init__` method initializes the model architecture, including an LSTM layer (`nn.LSTM`) and a fully connected layer (`nn.Linear`). The LSTM layer is bidirectional and processes input data in batches.
- The `device` attribute is set to `'cpu'` by default.

```
    def to(self, device):
        self.device = device
        super().to(device)

    def forward(self, input):
        #batch_size = input.size(0)

        # Bidirectional introduces a factor of 2 in some places
        # Input shape is (BATCH_SIZE, SEQ_LENGTH, NUM_INPUT_FEATURES)
        # Output shape is (BATCH_SIZE, SEQU_LENGTH, 2 * HIDDEN_SIZE)
        # Hidden shape is (2 * NUM_LAYERS, BATCH_SIZE, HIDDEN_SIZE)
        #hidden = torch.zeros(2 * NUM_LAYERS, batch_size,
HIDDEN_SIZE).to(self.device)
        out, state = self.model(input)
        estimates = self.fc(out)

        return estimates, state
```

- The `to` method sets the device on which the model will be trained and moves the model to that device.
- The `forward` method defines the forward pass of the model. It takes input data and returns output predictions. The input is passed through the LSTM layer (`self.model`) and then through the fully connected layer (`self.fc`). The output predictions and the hidden state are returned.

```
def load_data(dir, files=None, ignored_features=None):
    # Group files by common prefix
    # match[1]: original data filename
    # match[2]: wavelet coefficient type (cA2, cD1, cD2)
```

```
    regex = re.compile(r'(.*)_(c[AD][12])_features\.csv')
    sets = OrderedDict()
    keys = set()
    files = sorted(os.listdir(dir))
    for f in files:
        match = regex.search(f)
        if match:
            keys.add(match[1])
            sets.setdefault(match[1], []).append(f)

    # Load files and concatenate those that belong together
    datasets = []
    for group in sets.values():
        # IMPORTANT: coefficients will be in the order cA2, cD1, cD2!
        group = sorted(group)
        frames = []
        for f in group:
            match = regex.search(f)
            data = pd.read_csv(os.path.join(dir, f))
            data = data.rename(columns=lambda l: f'{l}_{match[2]}')
            frames.append(data)
        # Stack horizontally so all features for each frame are next to
each other
        datasets.append(pd.concat(frames, axis=1))

    # Load the groundtruth for each recording
    groundtruths = []
    for k in sorted(keys):
        try:
            gt = pd.read_csv(os.path.join(dir, k +
'_orientation_kalman_windowed.csv'))
        except FileNotFoundError:
            gt = pd.read_csv(os.path.join(dir, k +
'_orientation_simple_windowed.csv'))
        groundtruths.append(gt)

    # Stack vertically to create one big dataset
    all_input = pd.concat(datasets, axis=0, keys=keys)
    all_groundtruth = pd.concat(groundtruths, axis=0,
keys=keys).drop(columns='window')

    if ignored_features:
        ignored_features = ['f_' + f if not f.startswith('f_') else f for f
```

```
in ignored_features]
        channels = [f'emg_{i}' for i in range(EMG_CHANNELS)]
        ignored_cols = [f'{c}_{f}' for f in ignored_features for c in
channels]
        all_input[ignored_cols] = 0.0

    training_data = torch.from_numpy(all_input.values.astype(np.float32))
    groundtruth_data =
torch.from_numpy(all_groundtruth.values.astype(np.float32))

    return TensorDataset(training_data, groundtruth_data)
```

- The `load_data` function loads EMG data and ground truth orientation data from files stored in a specified directory.
- It groups the files by a common prefix and loads them into datasets. It also loads corresponding ground truth data.
- The loaded data is concatenated and organized into a `TensorDataset` object, which is returned.

```
def train(data):
    if torch.cuda.is_available():
        device = torch.device('cuda')
        print('Training RNN on CUDA device')
    else:
        device = torch.device('cpu')
        print('Training RNN on CPU')

    # For reference:
https://machinelearningmastery.com/multivariate-time-series-forecasting-lst
ms-keras/
    model = AIModel()
    model.to(device)

    epochs = 100
    learning_rate = 0.001

    loss_function = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    dataloader = DataLoader(data, batch_size=BATCH_SIZE * SEQ_LENGTH,
shuffle=True, drop_last=True)
    num_samples = len(dataloader.dataset)

    for epoch in range(epochs):
```

```
        print(f"Epoch {epoch + 1}\n-------------------------------")
        for batch_id, (samples, groundtruth) in enumerate(dataloader):
            optimizer.zero_grad()
            samples = samples.to(device)
            groundtruth = groundtruth.to(device)
            output, _ = model(samples.view(BATCH_SIZE, SEQ_LENGTH, -1))
            loss = loss_function(output, groundtruth.view(BATCH_SIZE,
SEQ_LENGTH, -1))
            loss.backward()
            optimizer.step()

            if batch_id % 100 == 0:
                loss, current = loss.item(), (batch_id + 1) * len(samples)
                print(f"loss: {loss:>7f}
[{current:>5d}/{num_samples:>5d}]")

    return model
```

- The `train` function trains the AI model using the provided data.
- It initializes the model, loss function, optimizer, and data loader.
- It iterates through epochs and mini-batches of data, computes the loss, and performs backpropagation to update model parameters.
- During training, it prints the loss and training progress.

```
def save_model(model, dir):
    t = datetime.now().strftime('%Y%m%d_%H%M%S')
    model_name = f'model_{t}.torch'
    out_file = os.path.join(dir, model_name)
    torch.save(model.state_dict(), out_file)
    return out_file


def load_model(path):
    model = AIModel()
    model.load_state_dict(torch.load(path))
    model.eval()
    return model
```

- `save_model` saves the trained model to a specified directory with a filename containing the current timestamp.
- `load_model` loads a trained model from a specified file path and returns the model.

This file encapsulates the functionality for training a neural network model on EMG data and provides utilities for saving and loading trained models.

__init__.py
The `__init__.py` file in the `emg_robot.learn` package serves as an initializer for the package, defining the list of modules to be imported when the package is imported as a whole. Let's break down the content of this file:

```
from .trainer import load_data, train, save_model, load_model
```

1. `from .trainer import load_data, train, save_model, load_model`: This line imports specific functions (`load_data`, `train`, `save_model`, `load_model`) from the `trainer` module within the same package (`emg_robot.learn`). These functions are essential for training machine learning models related to EMG (Electromyography) data.
The purpose of the **`__init__.py`** file is to provide a convenient way to access important functions related to training machine learning models within the `emg_robot.learn` package. By importing these functions in the `__init__.py` file, users can directly access them upon importing the package itself, making the API more user-friendly and intuitive.

---

preprocess

---

augmentor.py
This file encapsulates the functionality for preprocessing EMG and IMU data, including filtering, feature extraction, and orientation estimation.
The `augmentor.py` file step by step:

```
import os
import time

import numpy as np
import pandas as pd
import pywt
from imusensor.filters import kalman
from scipy.signal import butter, lfilter

from .features import all_features
from .utils import *
```

- This section imports necessary modules and libraries for data preprocessing and feature extraction. Notable imports include:
  - `numpy` and `pandas`: Libraries for numerical computing and data manipulation.
  - `pywt`: Python library for wavelet transform.
  - `kalman` from `imusensor.filters`: Kalman filter for sensor data.
  - `butter` and `lfilter` from `scipy.signal`: Functions for Butterworth filter.
  - `all_features` from `features`: A variable containing a list of all features used in the preprocessing.
  - Various utility functions from the `utils` module.

```
EMG_RES = 2**12  # 12bit ADC
# TODO adjust
WINDOW_LENGTH = 0.1  # How long each window should roughly be in seconds
WINDOW_OVERLAP = 0.5  # ratio of how much windows should overlap
IMU_METHOD = 'kalman'  # simple | kalman

COL_DT = 'dt'
COL_EMG_CHANNELS = ['emg_0', 'emg_1', 'emg_2', 'emg_3', 'emg_4']
COL_IMU_ACCELS = ['acc_x', 'acc_y', 'acc_z']
COL_IMU_GYROS = ['gyro_x', 'gyro_y', 'gyro_z']
```

- This section defines constants and parameters used for preprocessing. These include the EMG resolution, window length, window overlap ratio, method for estimating IMU angles, and column names for various data types.

```
def get_col_label(channel, feature):
    return f"{channel}_{feature.split('_', 1)[1]}"


def get_dt(df):
    if isinstance(COL_DT, float):
        COL_DT
    return df[COL_DT].mean()
```

- `get_col_label` is a utility function that generates a column label based on the channel and feature.
- `get_dt` is a function that calculates the mean delta time from a DataFrame.

```
def normalize(df):
    min = df.min(axis=0)
    return (df - min) / (df.max(axis=0) - min)
```

- `normalize` is a function that normalizes the values of a DataFrame between 0 and 1.

```
def filter_butterworth(data, lowcut, highcut, fs, order=5):
    # Butterworth filter implementation
    # ...
    return y
```

- `filter_butterworth` applies a Butterworth bandpass filter to data.

```
def calc_features(df):
    # Calculate features for each channel
    # ...
    return ret
```

- `calc_features` calculates features for each channel of EMG data.

```
def calc_emg_wavelet_features(coeffs):
```

```
    # Calculate wavelet features
    # ...
    return f
```

- `calc_emg_wavelet_features` computes wavelet features from wavelet coefficients.

```
def calc_emg_wavelets(df):
    # Calculate wavelet decompositions
    # ...
    return [pd.DataFrame(arr, columns=['window'] + cols) for arr in (cA2,
cD2, cD1)]
```

- `calc_emg_wavelets` performs discrete wavelet decompositions on EMG data.

```
def calc_emg_features(df, outdir, basename):
    # Calculate EMG features
    # ...
```

- `calc_emg_features` processes EMG data, calculates features, applies filters, and saves the results to files.

```
def calc_imu_orientation_simple(df):
    # Calculate simple IMU orientation
    # ...
    return ret
```

- `calc_imu_orientation_simple` computes simple orientation estimates from IMU accelerometer data.

```
def calc_imu_orientation_kalman(df, kf=kalman.Kalman()):
    # Calculate IMU orientation using Kalman filter
    # ...
    return ret
```

- `calc_imu_orientation_kalman` estimates IMU orientation using a Kalman filter.

```
def calc_imu_orientation(df):
    # Calculate IMU orientation based on selected method
    # ...
    return ret
```

- `calc_imu_orientation` calculates IMU orientation based on the selected method (Kalman filter or simple calculation).

```
def calc_imu_orientation_windowed(df, ori):
    # Calculate windowed IMU orientation
    # ...
    return owin
```

- `calc_imu_orientation_windowed` computes windowed IMU orientation from continuous estimates.

```
def calc_imu_features(df, outdir, basename):
    # Calculate IMU features
    # ...
```

- `calc_imu_features` processes IMU data, calculates orientation, and saves the results to files.

```
def process_recordings(recordings_dir, out_dir=None, do_emg=True,
do_imu=True):
    # Process recordings from a directory
    # ...
```

- `process_recordings` is the main function that processes recordings from a directory, calculates features, and saves the results to files.

---

features.py

The `features.py` file contains a collection of functions for computing various features from EMG (Electromyography) data. Let's break down each function:

```
import numpy as np
```

This line imports the NumPy library, which is used for numerical computing.

```
# TODO T-values
myop_T = 0.5
wamp_T = 0.5
ialv_T = 1.0   # to avoid log of negative values
```

These variables define threshold values used in some feature calculations.
Now, let's go through each feature function:
- `diff(win)`: Computes the difference between consecutive samples in a window.
- `f_iemg(win)`: Computes the integrated EMG, which is the absolute sum of EMG values in a window.
- `f_mav(win)`: Computes the mean absolute value of EMG in a window.
- `f_ssi(win)`: Computes the simple square integrated EMG.
- `f_rms(win)`: Computes the root mean square of EMG values in a window.
- `f_var(win)`: Computes the variance of EMG values in a window.
- `f_myop(win)`: Computes the myopulse percentage rate.
- `f_wl(win)`: Computes the waveform length.
- `f_damv(win)`: Computes the difference absolute mean value.
- `f_m2(win)`: Computes the second-order moment.
- `f_dvarv(win)`: Computes the difference variance version.
- `f_dasdv(win)`: Computes the difference absolute standard deviation value.
- `f_max(win)`: Computes the maximum EMG value in a window.
- `f_min(win)`: Computes the minimum EMG value in a window.
- `f_wamp(win)`: Computes the Willison amplitude.
- `f_iasd(win)`: Computes the integrated absolute of the second derivative.
- `f_iatd(win)`: Computes the integrated absolute of the third derivative.
- `f_ieav(win)`: Computes the integrated exponential of absolute values.

- `f_ialv(win)`: Computes the integrated absolute log values.
- `f_ie(win)`: Computes the integrated exponential.

Finally, the `all_features` list contains a selection of these feature functions that are considered useful for EMG analysis.

These functions provide a range of features that can be computed from EMG data, which are commonly used in signal processing and analysis for various applications such as gesture recognition and muscle activity monitoring.

---

utils.py

The `utils.py` file contains utility functions used in preprocessing EMG (Electromyography) data. The utils functions are used to prepare EMG data by dividing it into overlapping windows, calculating parameters for windowing, and generating window indices. They facilitate the preprocessing of EMG signals for further analysis or modeling.

```
import numpy as np
```

This line imports the NumPy library, which is used for numerical computing.

```
def get_samples_per_window(window_length, dt):
    return int(window_length // dt)
```

This function calculates the number of samples per window based on the desired window length and the time interval between samples (`dt`).

```
def get_window_params(num_samples, samples_per_window, overlap):
    if isinstance(overlap, float):
        num_overlap = int(overlap * samples_per_window)
    else:
        num_overlap = overlap

    window_offset = int(samples_per_window - num_overlap)
    num_windows = int(num_samples // window_offset - samples_per_window /
window_offset + 1)
    return num_windows, window_offset
```

This function calculates parameters for dividing a signal into overlapping windows. It takes the total number of samples (`num_samples`), the number of samples per window (`samples_per_window`), and the overlap ratio (`overlap`). It returns the number of windows and the window offset.

```
def get_window_intervals(num_samples, samples_per_window, overlap):
    if isinstance(overlap, int):
        r_overlap = overlap / samples_per_window
    else:
        r_overlap = overlap
    num_windows, _ = get_window_params(num_samples, samples_per_window,
overlap)
    idx = np.arange(num_windows) * (1. - r_overlap)
```

```
    idx = np.tile(idx, (2, 1)).T
    idx[:, 1] += 1
    idx *= samples_per_window
    return idx.astype(np.int32)
```

This function computes the intervals for overlapping windows. It takes the total number of samples (`num_samples`), the number of samples per window (`samples_per_window`), and the overlap ratio (`overlap`). It returns an array of window intervals.

```
def get_window_indices(num_windows, target_len):
    return np.repeat(np.arange(num_windows), target_len).astype(np.int32)
```

This function generates indices for windowing data. It takes the number of windows (`num_windows`) and the target length of each window (`target_len`). It returns an array of window indices.

---

visualize.py

The `visualize.py` file contains a script to visualize features of preprocessed EMG (Electromyography) data using Plotly Express.This script visualizes features of preprocessed EMG data by displaying an image representation of the data using Plotly Express. It automatically detects and processes files in a specific directory based on their filenames.
Let's break down the code:

```
import plotly.express as px
import pandas as pd

import os
import re
```

This section imports the necessary libraries: `plotly.express` for interactive plotting, `pandas` for data manipulation, `os` for operating system operations, and `re` for regular expressions.

```
def show_features(data):
    if isinstance(data, str):
        data = pd.read_csv(str)

    fig = px.imshow(data.to_numpy())
    fig.show()
```

This function `show_features` takes a dataframe (`data`) or a file path (`str`) as input. If the input is a string (file path), it reads the CSV file into a pandas dataframe. Then, it uses Plotly Express to create an image representation of the dataframe using `imshow()`. Finally, it displays the figure.

```
if __name__ == '__main__':
    regex = re.compile(r'(.*)_(c[AD][12])_features\.csv')
    files = sorted(os.listdir("d:/Code/workspace/EMG
Robot/datasets/test/preprocessed/"))
```

```
    for f in files:
        match = regex.search(f)
        if match:
            show_features(f)
```

The `if __name__ == '__main__':` block is the entry point of the script. It defines a regular expression `regex` to match filenames with a specific pattern. It then lists files in a directory and iterates over them. For each file, it checks if the filename matches the pattern defined by the regular expression. If it matches, it calls the `show_features` function to visualize the features of the corresponding CSV file.

_augmentor_openpose.py

The `_augmentor_openpose.py` script is designed to perform data augmentation using OpenPose, a library for real-time multi-person keypoint detection and multi-threading written in C++. The `_augmentor_openpose.py` script provides placeholders for functions to find synchronization markers in EMG and OpenPose data, calculate angles between keypoints using OpenPose, and calculate ground truth angles from videos using OpenPose. These functions need to be implemented to perform actual data augmentation with OpenPose.

Let's break down the code:

```
import numpy as np
import cv2
from openpose import pyopenpose as op
```

This section imports necessary libraries: `numpy` for numerical computations, `cv2` for computer vision tasks, and `pyopenpose` module for interfacing with the OpenPose library.

```
openpose_proto = 
"openpose/mpi/pose_deploy_linevec_faster_4_stages.prototxt"
openpsoe_weights = "openpose/mpi/pose_iter_160000.caffemodel"
```

These variables store the paths to the OpenPose model files: the `.prototxt` file containing the network architecture and the `.caffemodel` file containing the pre-trained weights.

```
shoulder_idx = 2
elbow_idx = 3
wrist_idx = 4
```

These variables define the indices of keypoints corresponding to the shoulder, elbow, and wrist in the OpenPose output.

```
def find_sync_markers_emg(emg, expected_len_samples, threshold):
    pass # TODO
```

This function is intended to find synchronization markers in EMG data, but it is currently a placeholder (`pass`) and needs to be implemented.

```
def find_sync_markers_openpose(op_seq):
```

```
    pass # TODO
```

Similar to the previous function, this one is intended to find synchronization markers in OpenPose data, but it is also a placeholder (`pass`) and needs to be implemented.

```
def calc_angle_of_points(p0, p1, p2):
    # Function to calculate the angle between three points
    pass # TODO
```

This function calculates the angle between three points (p0, p1, p2) using trigonometry. However, it is currently a placeholder (`pass`) and needs to be implemented.

```
def calc_groundtruth(df, vid, params=None):
    # Function to calculate ground truth angles from video using OpenPose
    pass # TODO
```

This function takes a pandas DataFrame (`df`) and a video file path (`vid`) as input. It reads the video frame by frame, detects keypoints using OpenPose, calculates angles between shoulder, elbow, and wrist keypoints, and stores them in the DataFrame. However, it is currently a placeholder (`pass`) and needs to be implemented.

__init__.py

The `__init__.py` file in the `preprocess` module of the EMG-Robot project serves as an initializer for the module, defining what should be imported when the module is imported elsewhere in the project. `__init__.py` file in the `preprocess` module of the EMG-Robot project imports specific functions and classes from the `augmentor.py` file and the `all_features` list from the `features.py` file, making them accessible when the `preprocess` module is imported in the project.

Let's break down the content of this file:

```
from .augmentor import \
        process_recordings, filter_butterworth, \
        calc_imu_features, calc_imu_orientation, \
        calc_emg_features, calc_emg_wavelets, calc_emg_wavelet_features, \
        get_window_params, get_window_intervals, get_window_indices,
get_samples_per_window
```

This line imports specific functions and classes from the `augmentor.py` file within the same directory. These functions and classes include:

- **`process_recordings`**: A function to process recordings and perform data augmentation.
- **`filter_butterworth`:** A function to apply a Butterworth filter to EMG data.
- **`calc_imu_features`:** A function to calculate features from IMU (Inertial Measurement Unit) data.
- **`calc_imu_orientation`:** A function to calculate orientation angles from IMU data.
- **`calc_emg_features`:** A function to calculate features from EMG (Electromyography) data.
- **`calc_emg_wavelets`:** A function to perform wavelet decomposition on EMG data.
- **`calc_emg_wavelet_features`:** A function to calculate features from wavelet coefficients of EMG data.

- `get_window_params`: A function to calculate parameters for data windowing.
- `get_window_intervals`: A function to calculate intervals for data windowing.
- `get_window_indices`: A function to calculate indices for data windowing.
- `get_samples_per_window`: A function to calculate the number of samples per window for data windowing.

```
from .features import all_features
```

This line imports the `all_features` list from the `features.py` file within the same directory. The `all_features` list contains a collection of functions used to calculate different features from EMG data.

---

record

---

emg_recorder.py
The `emg_recorder.py` script is designed to record data from EMG (Electromyography) sensors and an IMU (Inertial Measurement Unit) sensor. The `emg_recorder.py` script continuously records data from EMG sensors and an IMU sensor, storing the data in a CSV file. It handles signals to gracefully exit the recording process and calculates the time intervals between sensor readings to maintain a specified sampling rate. Let's break down the functionality and structure of the script:
Importing Libraries
- `smbus`: Used for I2C communication with the sensors.
- `signal`: Used to handle signals such as SIGINT and SIGTERM.
- `math`: Provides mathematical functions.
- `time`: Used for time-related functions.
- `sys`: Provides access to some variables used or maintained by the Python interpreter and to functions that interact with the interpreter.
Initializing Variables
- Constants are defined for I2C addresses of different sensors (`A0` to `A7` for EMG sensors and `IMU` for the IMU sensor).
- `bus` is initialized as an instance of `smbus.SMBus(1)` for I2C communication.
- `addresses` are obtained from command-line arguments (excluding the last argument, which is used as the filename).
- `FILENAME` is set as the filename for the CSV file where data will be stored.
Defining Utility Functions
- `get_value`: A function to parse the raw data from sensors and convert it to the appropriate value using scaling factor.
- `get_imu_acc_raw` and `get_imu_gyro_raw`: Functions to read raw acceleration and gyroscope data from the IMU sensor.
- `get_imu_acc` and `get_imu_gyro`: Functions to read and calculate actual acceleration and gyroscope values from the IMU sensor.
Setting Up Signal Handling
- A `signal_handler` function is defined to handle SIGINT and SIGTERM signals.
- Signals are set up to call the `signal_handler` function in case of a kill signal.

Main Loop
- Constants for sampling rate (`ABTAST`) and distance between readings (`DISTANCE`) are defined.
- The IMU sensor is configured by writing to its register.
- A CSV header is written with column names for EMG data, accelerometer data, gyroscope data, and time interval (`dt`).
- A while loop is started to continuously read sensor data.
- Inside the loop:
  - Time difference since the last sensor read is calculated.
  - If the time difference is less than the specified distance, the loop continues.
  - Raw EMG sensor data is read and stored in the `values` list.
  - Raw acceleration and gyroscope data from the IMU sensor are read and stored in the `values` list.
  - The time difference (`delta`) is added to the `values` list.
  - The `values` list is written to the CSV file.
  - Time difference (`delta`) is added to `distances`, and `i` is incremented.

Now the script `emg_recorder.py` line by line:

```
#!/usr/bin/env python3
# Beispielhafte Benutzung:
# python main.py A0 A1 A2 A3 A4 Bewegung_1
# python main.py A2 A4 A0 A3 A1 Bewegung_1_1
```

- This line indicates that the script is written in Python 3.
- The lines starting with `#` are comments providing examples of how to use the script with command-line arguments.

```
import smbus
import signal
import math
import time
import sys
import csv
```

- These lines import necessary libraries:
  - `smbus`: for I2C communication with the sensors.
  - `signal`: for handling signals.
  - `math`: for mathematical functions.
  - `time`: for time-related functions.
  - `sys`: for accessing command-line arguments and system-specific parameters.
  - `csv`: for reading and writing CSV files.

```
A0 = 0x48
A1 = 0x49
A2 = 0x4A
A3 = 0x4B
A4 = 0x4C
```

```
A5 = 0x4D
A6 = 0x4E
A7 = 0x4F
IMU = 0x68
bus = smbus.SMBus(1)
addresses = []
for a in sys.argv[1:-1]:
  addresses.append(locals()[a])

print(addresses)
```

- These lines define I2C addresses for EMG sensors (`A0` to `A7`) and the IMU sensor (`IMU`).
- An instance of `smbus.SMBus(1)` is created for I2C communication.
- The script retrieves the I2C addresses from command-line arguments and stores them in the `addresses` list.

```
FILENAME = f"./data-{sys.argv[-1]}.csv"
print(FILENAME)
file = open(FILENAME, "w")
writer = csv.writer(file)
```

- This block sets the filename for the CSV file based on the last command-line argument and opens the file in write mode.
- It initializes a CSV writer to write data to the file.

```
def get_value(data, idx, scaling_f):
  v = (data[idx] << 8) | data[idx+1]
  if v >= 0x8000:
    v = -(65535 - v) + 1
  return v / scaling_f
```

- This function `get_value` is defined to parse the raw data from sensors and convert it to the appropriate value using a scaling factor.

```
def get_imu_acc_raw(address):
  data = bus.read_i2c_block_data(address, 0x3b, 6)
  return [get_value(data, i*2, 16384.0) for i in range(3)]

def get_imu_acc(address):
  data = bus.read_i2c_block_data(address, 0x3b, 6)
  ax, ay, az = [get_value(data, i*2, 16384.0) for i in range(3)]
  rx = math.atan2(ay, math.sqrt(ax**2 + az**2))
  ry = math.atan2(ax, math.sqrt(ay**2 + az**2))
  return (rx, ry)
```

- These functions `get_imu_acc_raw` and `get_imu_acc` read raw acceleration data from the IMU sensor and calculate the actual acceleration values.

```
def get_imu_gyro_raw(address):
  data = bus.read_i2c_block_data(address, 0x43, 6)
  return [get_value(data, i*2, 131.0) for i in range(3)]

def get_imu_gyro(address):
  data = bus.read_i2c_block_data(address, 0x43, 4)
  gx, gy = [get_value(data, i*2, 131.0) for i in range(2)]
  return (gx, gy)
```

- These functions `get_imu_gyro_raw` and `get_imu_gyro` read raw gyroscope data from the IMU sensor and calculate the actual gyroscope values.

```
def signal_handler(sig, frame):
    print('Mittlere Smampling rate: ', (distances / i))
    sys.exit(0)
```

- This function `signal_handler` is defined to handle SIGINT and SIGTERM signals.

```
for kill_signal in [signal.SIGINT, signal.SIGTERM]:
  signal.signal(kill_signal, signal_handler)
```

- This block sets up signal handling for SIGINT and SIGTERM signals, calling the `signal_handler` function when these signals are received.

```
ABTAST = 1.5
DISTANCE = 1/(ABTAST*1000)

last_sensor_read = int(time.time())
distances = 0
i = 0
```

- Constants `ABTAST` (sampling rate) and `DISTANCE` (distance between readings) are defined.
- Variables `last_sensor_read`, `distances`, and `i` are initialized for tracking time and iterations.

```
bus.write_byte_data(IMU, 0x6b, 0)
```

- This line configures the IMU sensor by writing to its register.

```
header = ["emg_0", "emg_1", "emg_2", "emg_3", "emg_4", "acc_x", "acc_y",
"acc_z", "gyro_x", "gyro_y", "gyro_z", "dt"]
writer.writerow(header)
```

- This line writes a CSV header with column names for EMG data, accelerometer data, gyroscope data, and time interval (`dt`).

```
while True:
  delta = time.time() - last_sensor_read
  if delta < DISTANCE:
    continue
```

```
  last_sensor_read = time.time()
  values = []

  for a in addresses:
    data = bus.read_i2c_block_data(a, 0, 2)
    values.append((data[0] << 8) | data[1])

  values += get_imu_acc_raw(IMU)
  values += get_imu_gyro_raw(IMU)
  values.append(delta)
  writer.writerow(values)

  distances += delta
  i += 1
```

- This is the main loop of the script:
  - It calculates the time difference since the last sensor read.
  - If the time difference is less than the specified distance, the loop continues.
  - EMG sensor data is read and stored in the `values` list.
  - Raw acceleration and gyroscope data from the IMU sensor are read and stored in the `values` list.
  - The time difference (`delta`) is added to the `values` list.
  - The `values` list is written to the CSV file.
  - Time difference (`delta`) is added to `distances`, and `i` is incremented.

This loop continues indefinitely until the script is terminated by a SIGINT or SIGTERM signal.

---

i2c_test.py

Sure, let's break down the script `i2c_test.py` line by line:

```
#!/usr/bin/env python3
```

- This line indicates that the script is written in Python 3.

```
import smbus
import time
import sys
```

- These lines import necessary libraries:
  - `smbus`: for I2C communication with the sensors.
  - `time`: for time-related functions.
  - `sys`: for accessing command-line arguments and system-specific parameters.

```
A0 = 0x48
A1 = 0x49
A2 = 0x4A
A3 = 0x4B
```

```
A4 = 0x4C
A5 = 0x4D
A6 = 0x4E
A7 = 0x4F
```

- These lines define I2C addresses for different devices (`A0` to `A7`).

```
bus = smbus.SMBus(1)
```

- This line creates an instance of `smbus.SMBus(1)` for I2C communication. The argument `1` represents the bus number.

```
address = locals()[sys.argv[1]]
```

- This line retrieves the I2C address from the command-line argument and stores it in the variable `address`.

```
while True:
    data = bus.read_i2c_block_data(address, 0, 2)
    value = (data[0] << 8) | data[1]
    print(value)
    time.sleep(1)
```

- This is the main loop of the script:
  - It continuously reads data from the specified I2C address.
  - `bus.read_i2c_block_data(address, 0, 2)` reads a block of data from the I2C device with the given address.
  - The received data is combined into a single value using bitwise shift and OR operations.
  - The value is printed to the console.
  - The loop pauses for 1 second using `time.sleep(1)`.

---

visualizer.py

Visualizer.py script is a Python program that serves the purpose of visualizing electromyography (EMG) data stored in a CSV file. It uses the Plotly Express library to generate interactive line plots of the EMG data, allowing users to visualize the patterns and characteristics of the recorded signals. Let's break down the script `visualizer.py` line by line:

```
import pandas as pd
import plotly.express as px
from augmentor import calc_emg_features, normalize
```

These lines import necessary libraries:
  - `pandas`: for data manipulation and analysis.
  - `plotly.express`: for interactive visualization.
  - `calc_emg_features`: a function from `augmentor` module for calculating EMG features.
  - `normalize`: a function from `augmentor` module for normalizing data.

```
win_len = 30
```

- This line defines the window length for calculating EMG features.

```
def show(df):
    fig = px.line(df)
    fig.show()
```

- This function `show(df)` takes a DataFrame `df` as input and generates a line plot using Plotly Express.

```
if __name__ == '__main__':
```

- This line ensures that the following code block is executed only when the script is run directly, not when it's imported as a module.

```
    df = pd.read_csv('recordings/data-test3.csv')
```

- This line reads a CSV file named `data-test3.csv` located in the `recordings` directory into a DataFrame `df`.

```
   df = normalize(df[:])
```

- This line normalizes the data in the DataFrame `df` using the `normalize` function.

```
    show(df)
```

- This line calls the `show` function to display the line plot of the entire DataFrame `df`.

```
    emg_cols = [c for c in df.columns if c.startswith('emg_')]
```

- This line creates a list `emg_cols` containing column names that start with `'emg_'`.

```
    calc_emg_features(df, win_len, emg_cols)
```

- This line calculates EMG features for the specified window length and EMG columns using the `calc_emg_features` function.

```
    for col in emg_cols:
        related = [c for c in df.columns if c.startswith(col)]
        show(df[related])
```

- This loop iterates over each EMG column in `emg_cols`, generates a list `related` containing columns related to the current EMG column, and displays the line plot of those related columns using the `show` function.

---

init.py

No information

_outline.py

This code outlines the structure for processing EMG data, classifying it, and controlling a robot based on the classification results. However, most of the functionality is marked with `TODO` comments, indicating that it needs to be implemented.

Importing Libraries:

```
import sys
import time
import numpy as np
```

Initialization Functions:

```
def init_emg(emg_config):
    if (emg_config.source == 'emg'):
        # TODO: Load and initialize neural network or whatever classifier
we'll be using
        # TODO: Initialize EMG sensors. This is probably also where we
would run a calibration or even additional training for our classifier
        pass
    elif (emg_config.source == 'dataset'):
        # TODO: Load dataset
        pass

def init_robot(robot_config):
    # TODO: Establish connection to robot (ROS)
    pass
```

Data Reading Function:

```
def read_data(data, emg_config):
    # TODO: Read next sample(s) into data and remove old ones
    if (emg_config.source == 'emg'):
        # TODO: Return samples collected since last call (might have to
collect in a separate thread?)
        pass
    elif (emg_config.source == 'dataset'):
        # TODO: Load data from dataset
        pass
```

Filtering Function:

```
def emg_filter(data, emg_config):
    # TODO: Apply band filter, noise reduction, whitening, normalization,
smoothing (moving average)...
    pass
```

Classification Function:

```
def emg_classify(data, emg_config):
    joint_estimates = {
        'joints': None,            # array of joint estimates for EITHER
the actual human's arm OR the resulting robot's arm
        'timestamp': time.time(),  # needed to calculate joint velocities
    }

    # TODO: Get joint estimates from classifier (e.g., neural network)
    return joint_estimates
```

Velocity Limiting Function:

```
def limit_joint_velocities(prev_joint_estimates, joint_estimates):
    # TODO: Avoid sudden accelerations due to noise or wrong
classifications
    pass
```

Sending Data to Robot Function:

```
def send_to_robot(joint_estimates, robot_config):
    # TODO: Send new joint space pose to robot (ROS)
```

Main Execution:

```
if __name__ == '__main__':
    # Configuration and metadata of the EMG source
    emg_config = {
        'source': 'dataset',
        'derivatives': 16,
        'samples_per_second': 100,
        'moving_average_window': 50,
    }
    # Everything we need to connect to the robot and control it
    robot_config = {
        'ip': '192.168.2.1',
        'port': 3000,
    }

    init_emg(emg_config)
    init_robot(robot_config)

    data = np.zeros((emg_config['moving_average_window'],
emg_config['derivatives']))
    prev_joint_estimates = None

    while True:
        try:
            read_data(data, emg_config)
            emg_filter(data, emg_config)
            joint_estimates = emg_classify(data, emg_config)
            limit_joint_velocities(prev_joint_estimates, joint_estimates)
            send_to_robot(joint_estimates, robot_config)
        except KeyboardInterrupt:
            print('Terminated by user')
            break
        except Exception as e:
            sys.exit(e)
```

```
    sys.exit(0)
```

Here is the `outline.py` file with detailed information.

`def init_emg(emg_config):``: Defines a function `init_emg` that takes an `emg_config` argument. This function is responsible for initializing the EMG (Electromyography) sensors or loading the EMG dataset, based on the configuration provided.

`def init_robot(robot_config):``: Defines a function `init_robot` that takes a `robot_config` argument. This function is responsible for establishing a connection to the robot (likely using ROS, Robot Operating System).

`def read_data(data, emg_config``):`: Defines a function `read_data` that takes `data` and `emg_config` as arguments. This function is responsible for reading the next sample(s) of EMG data into the `data` array and removing old samples.

`def emg_filter(data, emg_config):``: Defines a function `emg_filter` that takes `data` and `emg_config` as arguments. This function is responsible for applying filters and preprocessing techniques to the EMG data, such as band filtering, noise reduction, normalization, etc.

`def emg_classify(data, emg_config)`*:*`: Defines a function `emg_classify` that takes `data` and `emg_config` as arguments. This function is responsible for classifying the EMG data to estimate joint movements, possibly using a classifier like a neural network.

`def limit_joint_velocities(prev_joint_estimates, joint_estimates):``: Defines a function `limit_joint_velocities` that takes `prev_joint_estimates` and `joint_estimates` as arguments. This function is responsible for limiting sudden accelerations in joint movements to avoid noise or incorrect classifications.

`def send_to_robot(joint_estimates, robot_config):``: Defines a function `send_to_robot` that takes `joint_estimates` and `robot_config` as arguments. This function is responsible for sending the estimated joint space pose to the robot, likely using ROS.

`if __name__ == '__main__':``: Checks if the script is being run directly by the Python interpreter.

`emg_config = { ... }``: Defines the configuration and metadata of the EMG source, including the source type, number of derivatives, sampling rate, and moving average window.

`robot_config = { ... }`: Defines the configuration needed to connect to and control the robot, including the IP address and port.

`init_emg(emg_config)`: Initializes the EMG system based on the provided configuration.

`init_robot(robot_config)`: Initializes the robot connection based on the provided configuration.

`data = np.zeros((emg_config['moving_average_window'], emg_config['derivatives']))``: Initializes an array `data` filled with zeros to store EMG data samples.

`prev_joint_estimates = None``: Initializes the variable `prev_joint_estimates` to `None`.

`while True: ...`: Starts an infinite loop to continuously read, filter, classify, limit, and send data to the robot.

`try: ... except KeyboardInterrupt: ... except Exception as e: ...`: Handles keyboard interrupts and other exceptions gracefully, printing a message and exiting the program.

`sys.exit(0)`: Explicitly exits the program with a success status code.

---

defaults.py

This module defines constants and configurations related to I2C addresses of EMG sensors, their corresponding channel names, and the IP address of a robot device. These configurations provide information for the system to interact with the sensors and the robot.

Let's break it down:

Constants:

```
A0 = 0x48
A1 = 0x49
A2 = 0x4A
A3 = 0x4B
A4 = 0x4C
A5 = 0x4D
A6 = 0x4E
A7 = 0x4F
```

These constants represent hexadecimal addresses for different devices. In this context, they represent the I2C addresses of sensors or peripherals connected to the system.

Configuration:

```
# Addresses of the EMG sensors to read on every cycle. Order matters!
I2C_ADDRESSES = [A4, A5, A2, A6, A0]
```

This configuration variable `I2C_ADDRESSES` is a list that contains the I2C addresses of EMG sensors. These addresses are ordered based on the priority or sequence in which the data from these sensors should be read.

Configuration:

```
EMG_CHANNEL_NAMES = ["biceps", "triceps", "pronator teres",
"brachioradialis", "supinator"]
```

This configuration variable `EMG_CHANNEL_NAMES` is a list that contains the names of EMG channels corresponding to each sensor. Each name represents a muscle or muscle group being monitored by the respective sensor.

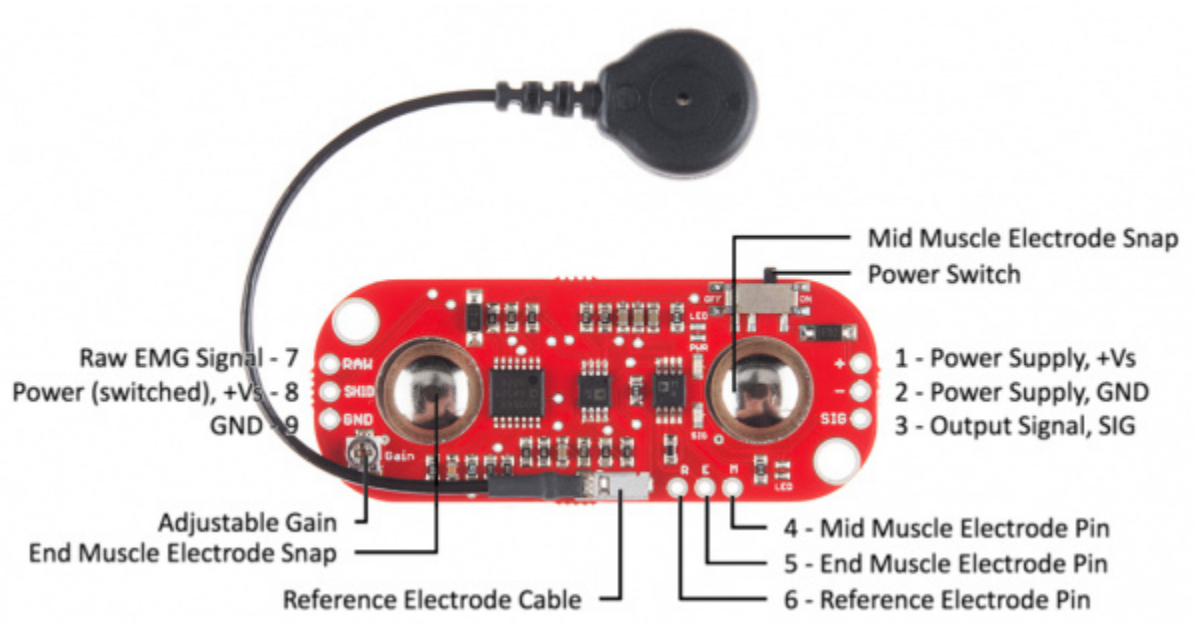
Configuration:

```
ROBOT_IP = "192.168.2.12"
```

This configuration variable `ROBOT_IP` represents the IP address of a robot device. It specifies the network address used to communicate with the robot.

Hardware

MyoWare EMG Muscle Sensor

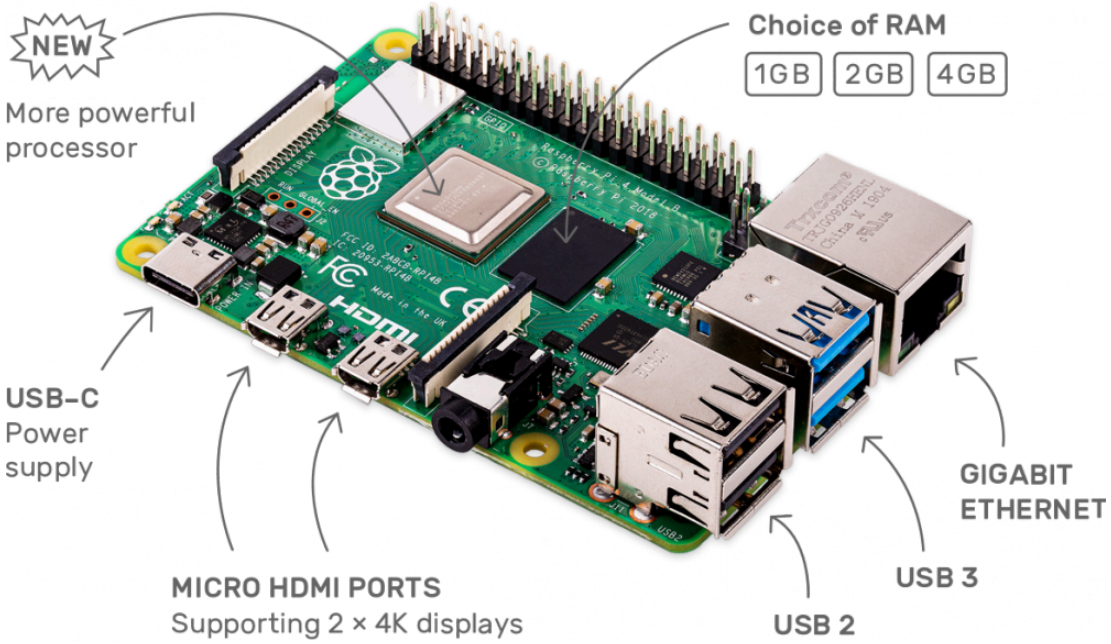


Here's a explanation of each point on the MyoWare EMG Muscle Sensor:

1. **Power Supply, +Vs (Pin 1)**: This is the positive power supply pin. It provides the voltage needed for the sensor to operate.
2. Power Supply, GND (Pin 2): This is the ground pin. It completes the circuit and provides the reference voltage for the sensor.
3. Output Signal, SIG (Pin 3): This is the output signal pin. It carries the processed EMG signal that can be used for various applications.
4. Mid Muscle Electrode Pin (Pin 4): This pin connects to the mid-section of the sensor pad, providing a reference point for measuring muscle activity.
5. **End Muscle Electrode Pin (Pin 5)**: This pin connects to the end section of the sensor pad, completing the circuit for accurate EMG measurement.
6. Reference Electrode Pin (Pin 6): This pin connects to the reference electrode, which helps establish a baseline for measuring muscle activity.
7. Raw EMG Signal (Pin 7): This pin provides the raw, unprocessed EMG signal from the muscle.
8. Power (switched), +Vs (Pin 8): This pin supplies power to the sensor. The power can be switched on or off using the power switch.
9. GND (Ground) (Pin 8): This is the ground connection for the sensor.
10. Reference Electrode Cable: The cable used for connecting the reference electrode.

11. Adjustable Gain: This feature allows you to adjust the gain or amplification of the EMG signal to suit your specific needs.
12. Mid Muscle Electrode Snap: This is where you attach one of the sensor pads to the muscle you want to measure.

Raspberry Pi 3 Model B



4 x USB 2 Ports: Four USB 2.0 ports for connecting peripherals like keyboards and mice.
40 Pin Extended GPIO: GPIO header for connecting sensors and devices.
Broadcom BCM2837 Processor: 64-bit quad-core CPU running at 1.2GHz.
xGB RAM**: Sufficient memory for multitasking and running applications.
MicroSD Card Slot: Slot for storing the operating system and data.
10/100 LAN Port: Ethernet port for wired network connection.
Micro USB Power Input: Powered via micro USB for easy setup.

Robot

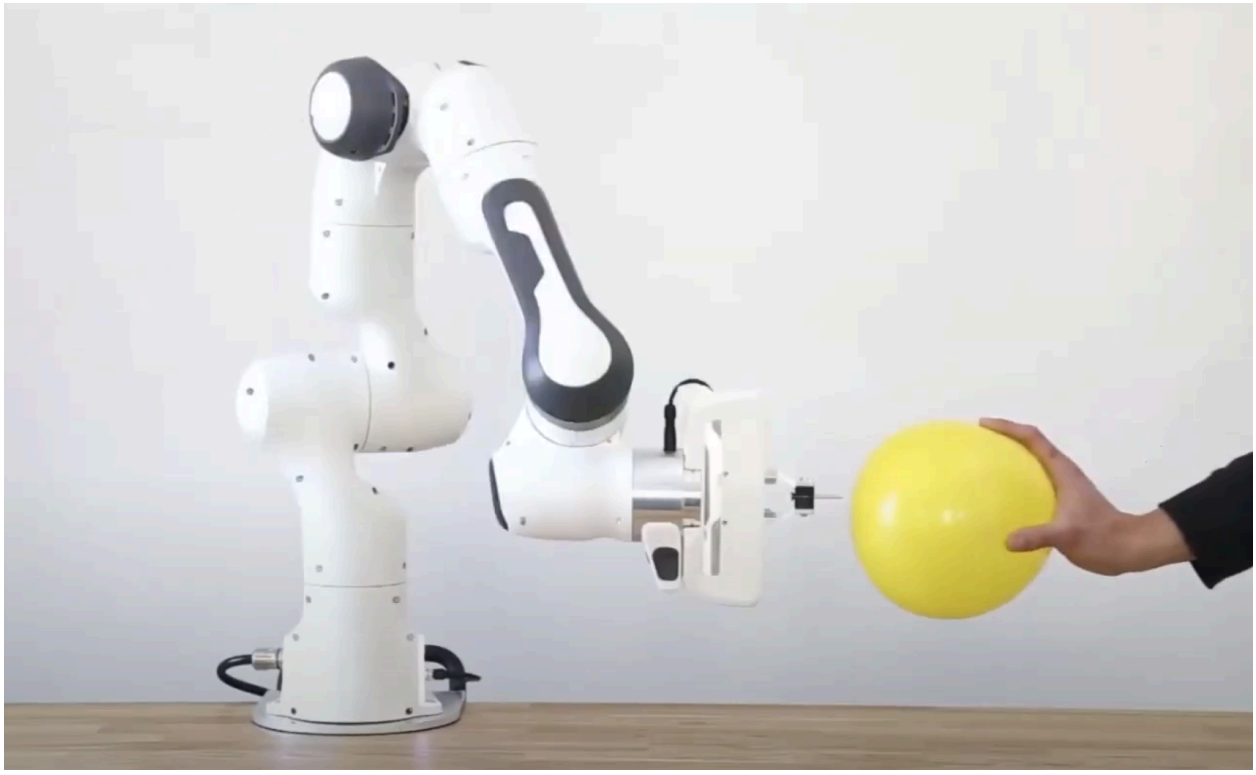


Foto: Robokind - Robonatives Initiative erhält Auszeichnung „Digitaler Ort Niedersachsen"

TODO: Get information about the robot

Dataset Analysis

These datasets (final_df) correspond to recordings made by Mohammed Fakih as part of his bachelor thesis in July 2022. The data was collected from 6 females and 6 males aged between 16 and 46 years, with no known muscular particularities
The dataset (final_df) consists of 4,169,942 entries and 12 columns, including:
- Five columns (emg_0, emg_1, emg_2, emg_3, emg_4) representing electromyography (EMG) sensor readings as integers.
- Seven columns (acc_x, acc_y, acc_z, gyro_x, gyro_y, gyro_z, dt) representing accelerometer and gyroscope readings as floating-point numbers.
The analysis considered data from all the listed CSV files:

rec-22-07-11-1.csv, rec-22-07-13-1.csv, rec-22-07-18-1.csv, rec-22-07-18-2.csv, rec-22-07-18-3.csv, rec-22-07-24-1.csv, rec-22-07-24-2.csv, rec-22-07-26-1.csv, rec-22-07-26-2.csv, rec-22-07-29-1.csv, rec-22-07-29-2.csv, rec-22-07-29-3.csv.

My analysis involved:
- Checking for missing values in each column (there are none).

- Conducting descriptive statistics to understand the distribution of numerical data.
- Creating scatter plots to visualize relationships between variables.
- Constructing box plots to identify outliers and distribution characteristics.
- Generating histograms to examine the distribution of individual variables.

These steps aim to gain insights into data patterns, correlations, and potential anomalies, thereby facilitating further analysis and modeling.

Descriptive statistics

```
DataFrame Information:
RangeIndex: 4169942 entries, 0 to 4169941
Data columns (total 12 columns):
 #   Column  Dtype
---  ------  -----
 0   emg_0   int64
 1   emg_1   int64
 2   emg_2   int64
 3   emg_3   int64
 4   emg_4   int64
 5   acc_x   float64
 6   acc_y   float64
 7   acc_z   float64
 8   gyro_x  float64
 9   gyro_y  float64
 10  gyro_z  float64
 11  dt      float64
dtypes: float64(7), int64(5)
```

Checking out the count, mean, standard deviation, minimum, 25th percentile (25%), median (50th percentile), 75th percentile (75%), and maximum values:

	emg_0	emg_1	emg_2	emg_3	emg_4	acc_x
count	4.169942e+06	4.169942e+06	4.169942e+06	4.169942e+06	4.169942e+06	4.169942e+06
mean	2.135609e+03	1.924850e+03	1.692768e+03	2.546116e+03	1.697954e+03	-2.556392e-01
std	1.319153e+03	1.166734e+03	1.200965e+03	1.013784e+03	1.266860e+03	3.829752e-01
min	1.360000e+02	3.610000e+02	9.500000e+01	3.490000e+02	8.100000e+01	-1.410522e+00
25%	9.550000e+02	8.430000e+02	6.930000e+02	1.673000e+03	6.670000e+02	-5.938721e-01
50%	1.943000e+03	1.590000e+03	1.547000e+03	2.364000e+03	1.303000e+03	-2.430420e-01
75%	3.808000e+03	2.982000e+03	2.465000e+03	3.803000e+03	2.725000e+03	6.713867e-02
max	3.824000e+03	3.823000e+03	3.815000e+03	3.819000e+03	3.815000e+03	1.359619e+00

	acc_y	acc_z	gyro_x	gyro_y	gyro_z	dt
count	4.169942e+06	4.169942e+06	4.169942e+06	4.169942e+06	4.169942e+06	4.169942e+06
mean	2.930594e-01	-2.187701e-01	-1.633320e+00	-3.311296e+00	1.968114e-02	2.521889e-03
std	5.081161e-01	6.757951e-01	1.005115e+02	4.925429e+01	3.483780e+01	2.051492e-03
min	-1.999878e+00	-1.999878e+00	-2.501221e+02	-2.501221e+02	-2.501221e+02	1.212835e-03
25%	-2.233887e-02	-8.087158e-01	-1.144275e+01	-9.106870e+00	-3.770992e+00	2.512932e-03
50%	3.417969e-01	-3.089600e-01	-1.183206e+00	-3.847328e+00	3.511450e-01	2.515793e-03
75%	6.928711e-01	4.150391e-01	8.404580e+00	1.648855e+00	5.145038e+00	2.518654e-03
max	1.999939e+00	1.999939e+00	2.501298e+02	2.501298e+02	2.501298e+02	1.418656e+00

Missing values per column

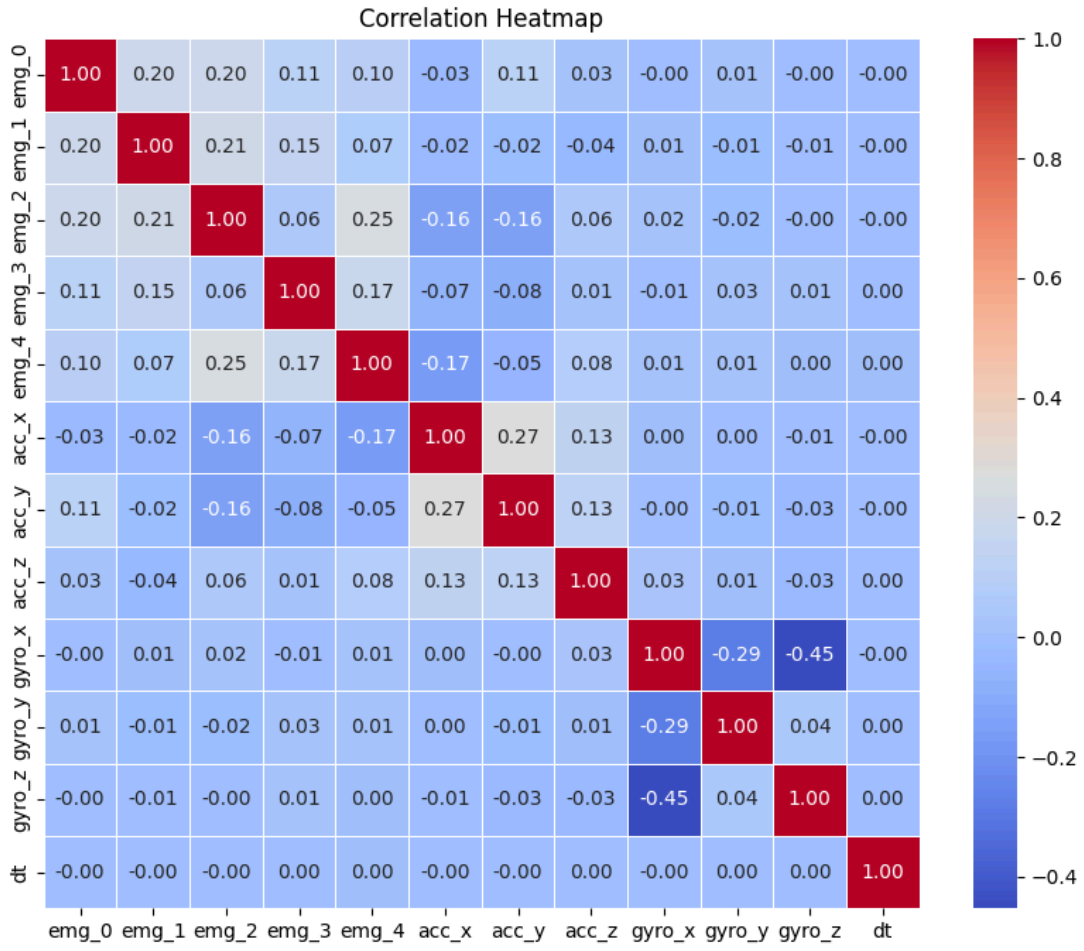
```
emg_0     0
emg_1     0
emg_2     0
emg_3     0
emg_4     0
acc_x     0
acc_y     0
acc_z     0
gyro_x    0
gyro_y    0
gyro_z    0
dt        0
dtype: int64
```

Correlations between variables

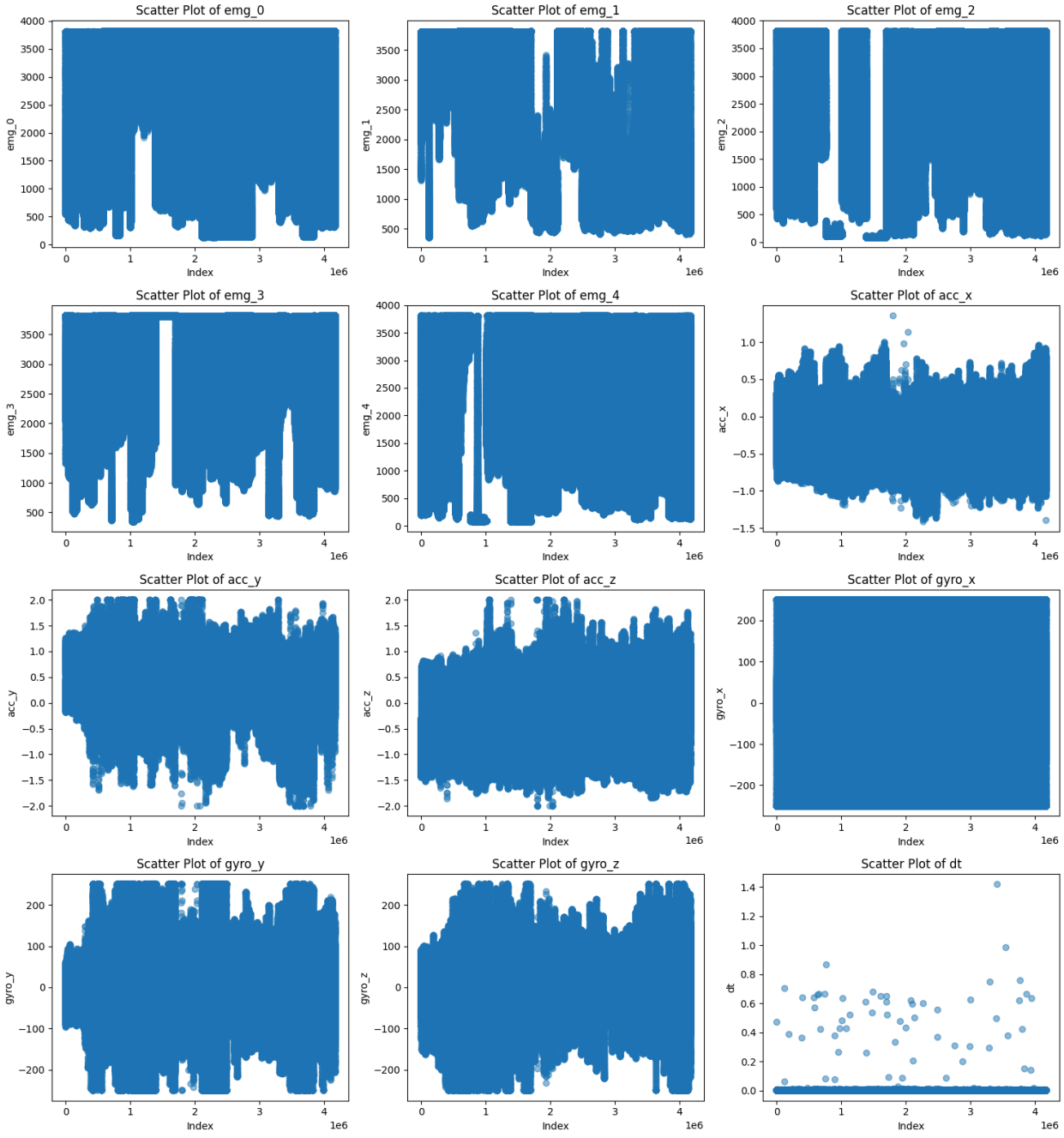
	emg_0	emg_1	emg_2	emg_3	emg_4	acc_x	acc_y	acc_z	gyro_x	gyro_y	gyro_z	dt
emg_0	**1.000000**	0.195007	0.197769	0.112502	0.097511	-0.029036	0.110004	0.026210	-0.004983	0.007119	-0.003528	-0.000080
emg_1	0.195007	**1.000000**	0.207796	0.147298	0.070808	-0.024481	-0.018113	-0.043499	0.005907	-0.010433	-0.007607	-0.000179
emg_2	0.197769	0.207796	**1.000000**	0.055816	0.246422	-0.157261	-0.156899	0.063337	0.021893	-0.019842	-0.003767	-0.000666
emg_3	0.112502	0.147298	0.055816	**1.000000**	0.174842	-0.072460	-0.078756	0.014810	-0.007231	0.026588	0.014784	0.000764
emg_4	0.097511	0.070808	0.246422	0.174842	**1.000000**	-0.165142	-0.052031	0.079421	0.011687	0.012530	0.003060	0.001188
acc_x	-0.029036	-0.024481	-0.157261	-0.072460	-0.165142	**1.000000**	0.271988	0.134142	0.003493	0.000735	-0.009514	-0.001264
acc_y	0.110004	-0.018113	-0.156899	-0.078756	-0.052031	0.271988	**1.000000**	0.125636	-0.000658	-0.011715	-0.028390	-0.000259
acc_z	0.026210	-0.043499	0.063337	0.014810	0.079421	0.134142	0.125636	**1.000000**	0.029867	0.005672	-0.025505	0.000447
gyro_x	-0.004983	0.005907	0.021893	-0.007231	0.011687	0.003493	-0.000658	0.029867	**1.000000**	-0.285746	-0.453075	-0.000278
gyro_y	0.007119	-0.010433	-0.019842	0.026588	0.012530	0.000735	-0.011715	0.005672	-0.285746	**1.000000**	0.042689	0.000496
gyro_z	-0.003528	-0.007607	-0.003767	0.014784	0.003060	-0.009514	-0.028390	-0.025505	-0.453075	0.042689	**1.000000**	0.000124

dt	-0.000080	-0.000179	-0.000666	0.000764	0.001188	-0.001264	-0.000259	0.000447	-0.000278	0.000496	0.000124	**1.000000**

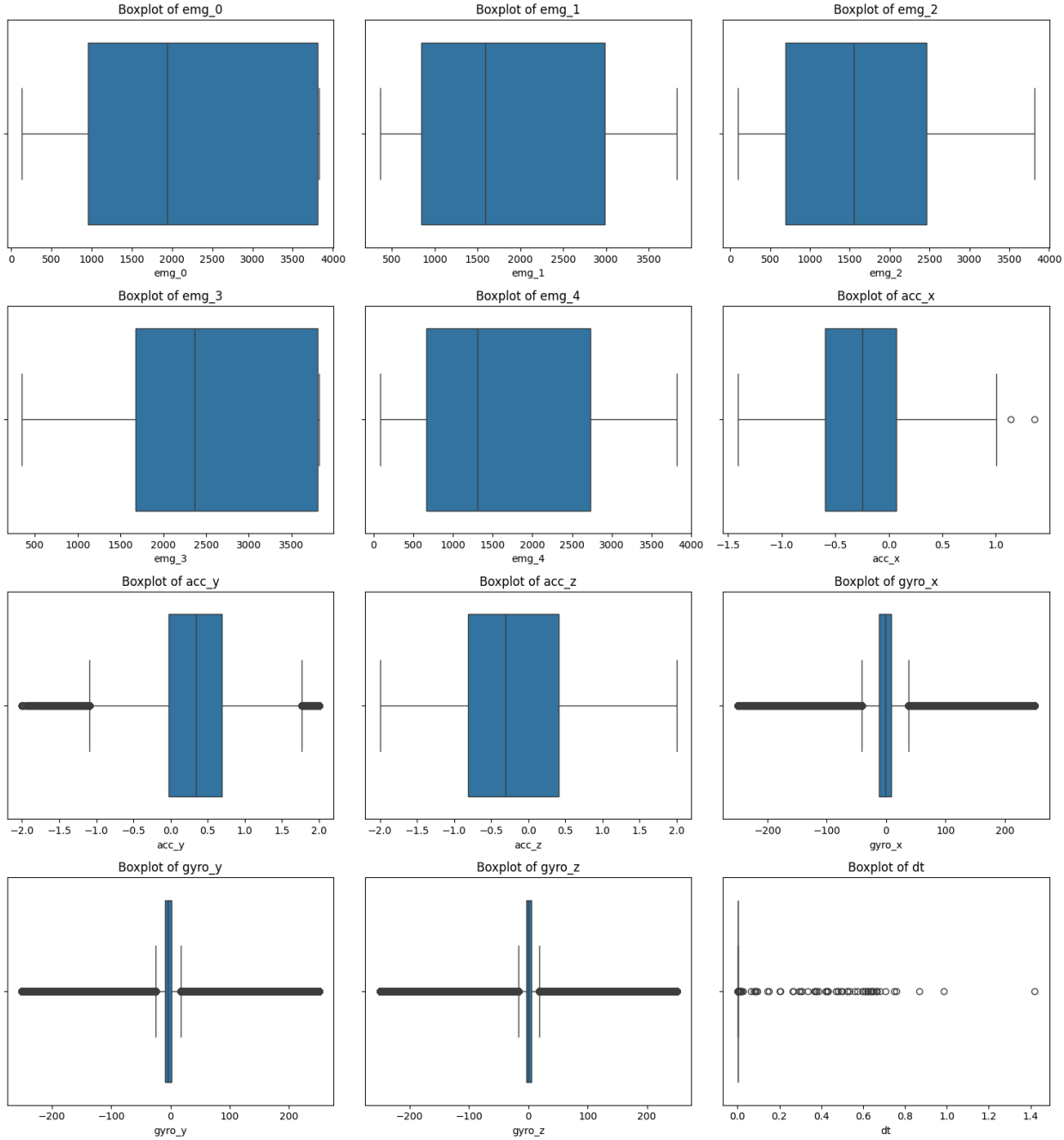
Heatmap correlation



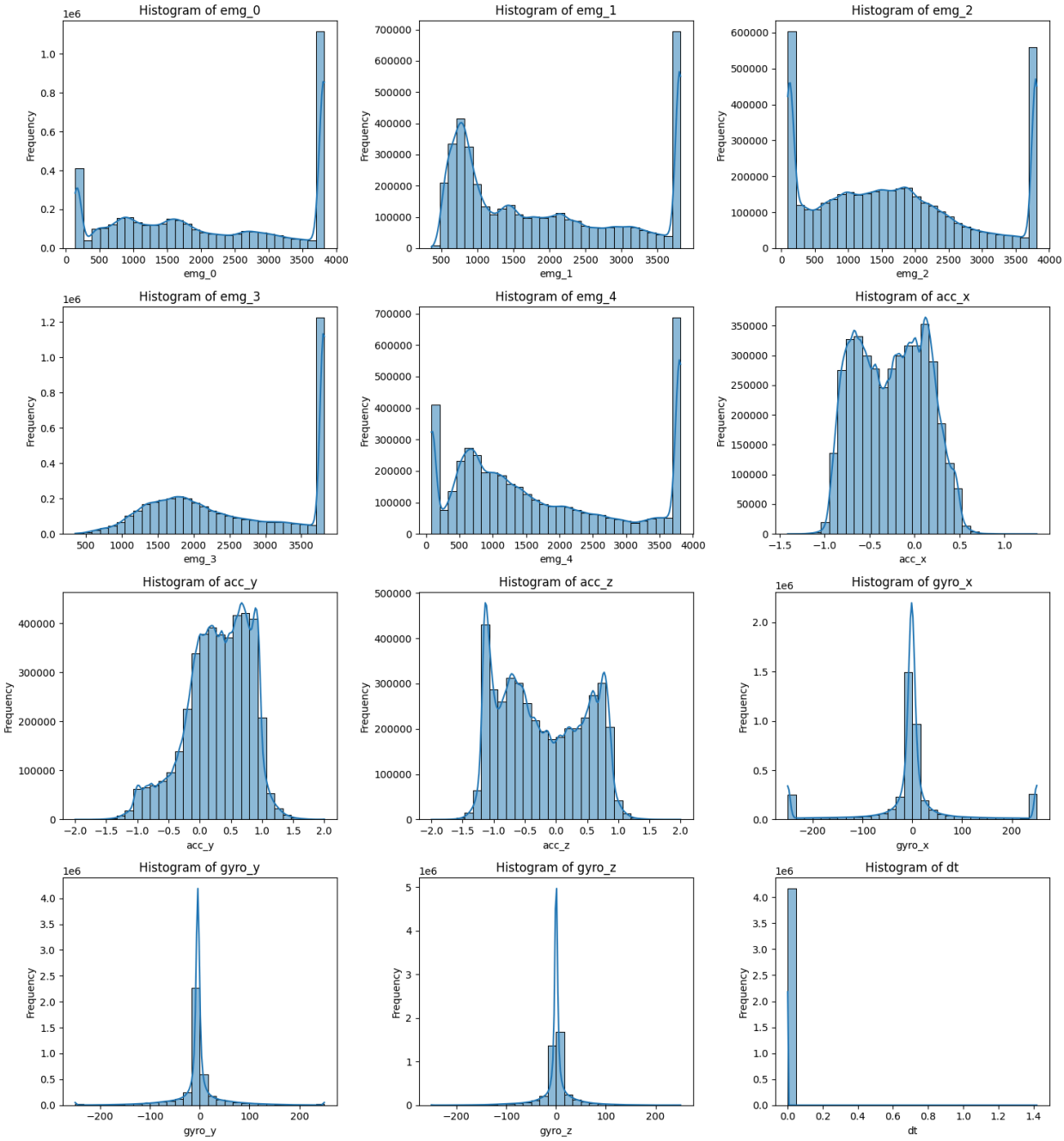
Scatter plots



Box plots


Boxplot of emg_0
emg_0
Boxplot of emg_1
emg_1
Boxplot of emg_2
emg_2
Boxplot of emg_3
emg_3
Boxplot of emg_4
emg_4
Boxplot of acc_x
acc_x
Boxplot of acc_y
acc_y
Boxplot of acc_z
acc_z
Boxplot of gyro_x
gyro_x
Boxplot of gyro_y
gyro_y
Boxplot of gyro_z
gyro_z
Boxplot of dt
dt

Histogram


Histogram of emg_0
Histogram of emg_1
Histogram of emg_2
Histogram of emg_3
Histogram of emg_4
Histogram of acc_x
Histogram of acc_y
Histogram of acc_z
Histogram of gyro_x
Histogram of gyro_y
Histogram of gyro_z
Histogram of dt
Frequency
emg_0
emg_1
emg_2
emg_3
emg_4
acc_x
acc_y
acc_z
gyro_x
gyro_y
gyro_z
dt
1e6

Remaining TODOs in the code

TODO in EMG-Robot\src\main.py

Some functions are missing on the main.py, we need to:
Complete the do_recording(args) function to start recording samples from the EMG sensors.
Implement the do_simulate(args) function to simulate classification of live EMG data without a robot.

TODO in EMG-Robot\src\emg_robot\control\controller_direct_fake.py

Base Class Creation: Implement a base class to encapsulate shared functionalities.
Inheritance Setup: Ensure both `DirectController` and `DirectControllerFake` inherit from the base class.
Real Controller Implementation: Create a class for the real direct controller to manage hardware interactions.
Fake Controller Refactoring: Adjust `DirectControllerFake` to provide simulated data for testing.
Common Method Definitions: Define common methods like `run_once()`, `run()`, and `emg_activity()` in the base class.
3wee

TODO in EMG-Robot\src\emg_robot_outline.py

Load and Initialize Classifier: Implement loading and initialization of the classifier or neural network used for EMG classification.
Initialize EMG Sensors (if applicable): If the EMG data comes from sensors, initialize them here. This may include calibration or additional training.
Load Dataset (if applicable): If using a dataset for EMG data, implement loading the dataset.
Establish Connection to Robot (ROS): Set up a connection to the robot using the Robot Operating System (ROS).
Read Next EMG Sample(s): Implement functionality to read the next sample(s) of EMG data and update the data structure.
Apply EMG Filtering: Implement signal processing techniques like band filtering, noise reduction, etc., to preprocess EMG data.
EMG Classification: Classify preprocessed EMG data into joint estimates using the loaded classifier.
Limit Joint Velocities: Implement a mechanism to limit joint velocities to ensure smooth control and prevent sudden movements.
Send Joint Space Pose to Robot (ROS): Translate joint estimates into robot commands and send them to the robot using ROS.

bus.read_i2c_block_data(a, 0, 2)

Other Observations: (TODO)

It is necessary to confirm whether the recordings are correctly recognizing the EMG signals. Attention should be paid to ensure that the EMG data is not being generated randomly and to understand the proper functioning of the EMG measurement equipment.

- **CODING and Testing**

  While testing the code in the EMG_recorder.py file, I encountered errors, and the recordings appeared to be random. After numerous nonsensical recordings, I decided to individually test each EMG connector to measure muscle movement. Several conclusions arose from this process:

  Port A1 - 0x49, was not being recognized.
  Port A3 - 0x4B, was not being recognized.
  Port A7 - 0x4F, was not being recognized.

  Attempts were made to modify configurations within the EMG_recorder.py code in copied files, as well as following tutorials to enable recognition of these devices using the Raspberry Pi's boot/config. Unfortunately, none of these approaches worked. I also tried addressing multiple I2C buses in programs as suggested in this link, but it didn't solve the problem either.

  To address this issue, I developed a new code named '*emg_recorder_mjz_withoutA1-A3-A.py*', which I have provided below:

```
#!/usr/bin/env python3
import smbus
import signal
import math
import time
import sys
import csv

A0 = 0x48
A1 = 0x49
A2 = 0x4A
```

```
A3 = 0x4B
A4 = 0x4C
A5 = 0x4D
A6 = 0x4E
A7 = 0x4F
IMU = 0x68

bus = smbus.SMBus(1)

# Function to read a single byte from a device
def read_byte(address, register):
    try:
        return bus.read_byte_data(address, register)
    except IOError as e:
        print(f"I/O error while trying to read from device at address
{hex(address)}:", e)
        return None

# Function to read accelerometer data from IMU
def read_accelerometer_raw():
    try:
        data = bus.read_i2c_block_data(IMU, 0x3B, 6)
        acc_x = (data[0] << 8 | data[1]) / 16384.0
        acc_y = (data[2] << 8 | data[3]) / 16384.0
        acc_z = (data[4] << 8 | data[5]) / 16384.0
        return acc_x, acc_y, acc_z
    except IOError as e:
        print(f"I/O error while trying to read accelerometer data:", e)
        return None, None, None

# Function to read gyroscope data from IMU
def read_gyroscope_raw():
    try:
        data = bus.read_i2c_block_data(IMU, 0x43, 6)
        gyro_x = (data[0] << 8 | data[1]) / 131.0
        gyro_y = (data[2] << 8 | data[3]) / 131.0
        gyro_z = (data[4] << 8 | data[5]) / 131.0
        return gyro_x, gyro_y, gyro_z
    except IOError as e:
        print(f"I/O error while trying to read gyroscope data:", e)
        return None, None, None

# Function to handle Ctrl+C signal
```

```
def signal_handler(sig, frame):
    print('Exiting...')
    sys.exit(0)

# Register signal handler for Ctrl+C
signal.signal(signal.SIGINT, signal_handler)

# Open CSV file for writing
filename = f"data-{sys.argv[-1]}.csv"
with open(filename, "w", newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["4th_EMG", "A2", "2nd_EMG", "A5", "1st_EMG", "Acc_X",
"3rd_EMG", "Acc_Z", "Gyro_X", "Gyro_Y", "Gyro_Z", "Timestamp"])
    #writer.writerow(["A0", "A1", "A2", "A3", "A4", "A5", "A6", "A7",
"Acc_X", "Acc_Y", "Acc_Z", "Gyro_X", "Gyro_Y", "Gyro_Z", "Timestamp"])

    # Main loop
    while True:
        timestamp = time.time()
        values = []

        for address in [A0, A1, A2, A3, A4, A5, A6, A7, IMU]:
            value = read_byte(address, 0x00)
            values.append(value)

        acc_x, acc_y, acc_z = read_accelerometer_raw()
        gyro_x, gyro_y, gyro_z = read_gyroscope_raw()

        row = values + [acc_x, acc_y, acc_z, gyro_x, gyro_y, gyro_z,
timestamp]
        writer.writerow(row)

        time.sleep(1)  # Delay for 1 second
```

The new code, named emg_recorder_mjz.py, addresses these issues. Also, I changed the name of the excel table for the position that each EMG was connected to.

Now the new code recognizes all plugged devices and indicates connection errors for faulty devices. If an error is encountered, the terminal will indicate the port that is not being recognized. as in the example below Results were:

```
I/O error when trying to read from device at address 0x4d: [Errno 121]
Remote I/O error
I/O error when trying to read from the device at address 0x4f: [Errno 121]
Remote I/O error
I/O error when trying to read from device at address 0x49: [Errno 121]
Remote I/O error
```

With that being said I did some more recordings: One EMG device exhibited recording errors and unstable connections, a potential defect related to poor cable and connector contacts.

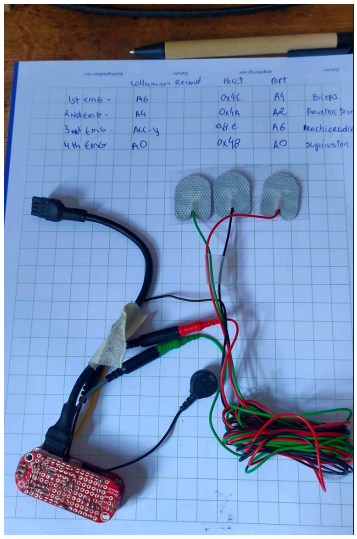


Figure: Probably defective device: (A5 0x4D) - Triceps - not recording properly, sometimes records data, sometimes not. LED light is unstable.

Additionally, issues were detected with the gyroscope and accelerometer, which inconsistently provided readings in some recordings. To address these issues, they were isolated until a better solution could be found. Consequently, four EMG sensors remained available:

A4 (0x4C),
A2 (0x44A),
A6 (0x4E),
A0 (0x48).

DOING RECORDS:

	A	B	C	D	E	F	G	H	I	J	
1	4th_EMG	A2	2nd_EMG	A5	1st_EMG	Acc_X	3rd_EMG	Acc_Z	Gyro_X	Gyro_Y	Gyr
2	14		14		14		14				
3	14		14		14		14				
4	14		5		14		14				
5	14		1		14		14				
6	14		1		14		12				
7	14		1		14		14				
8	14		1		14		14				
9	14		1		14		14				
10	14		1		14		14				
11	14		1		14		14				
12	14		5		14		14				
13	14		1		14		14				
14	14		5		14		11				
15	14		1		14		11				
16	14		6		14		14				
17	14		4		14		14				
18	14		3		14		14				
19	14		10		14		14				
20	14		2		14		14				
21	14		11		14		12				
22	14		3		14		14				
23	14		10		14		9				
24	14		1		14		14				
25	14		7		14		14				
26	14		2		14		10				
27	14		4		14		14				
28	14		14		14		14				
29	14		14		14		14				
30	14		14		14		9				
31	14		13		11		8				
32	14		10		3		9				
33	14		1		11		14				
34	14		1		14		14				
35	14		1		14		14				
36	14		11		14		14				
37	14		1		14		14				

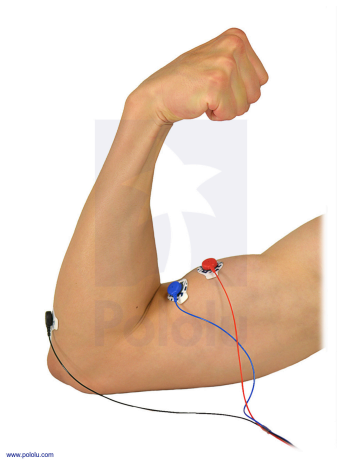
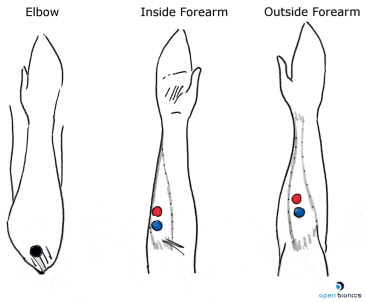
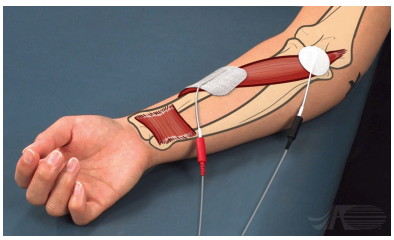
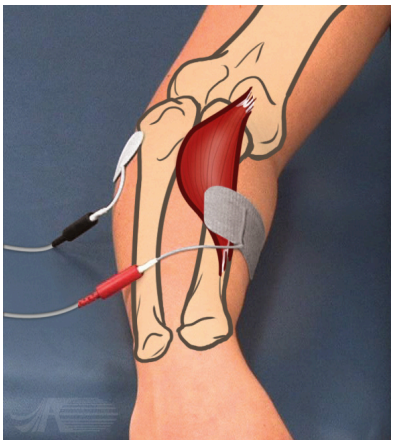
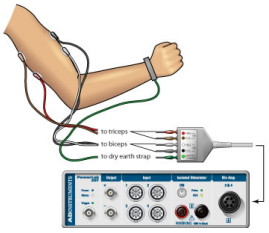
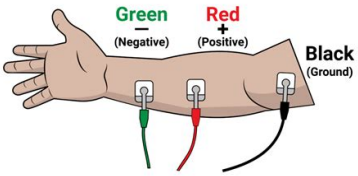
Now at least I can identify that the recordings are happening and there is movement happening with the EMG sensors. The columns on the Excel table correspond to the following:

1st_EMG - Biceps
2nd_EMG -Pronator Teres
3rd_EMG - Brochiaradialis
4th_EMG - Supinator
One device was unplugged - (0x4D - A5) - Triceps.

Figure: How I positioned the electrodes (Biceps, Pronator Teres, Brachiaradialis, Supinator). All neutral electrodes (black) were placed on the elbow area.

Muscles one by one

Biceps

		
Biceps	Pronator Teres	Brachioradialis
		
Supinator	Triceps*	Explanation