Open-Source Leg

Release 0.1.0

Senthur Raj Ayyappan

Sep 14, 2023

GETTING STARTED

An open-source software library for numerical computation, data acquisition, and control of lower-limb robotic prostheses.

CHAPTER

ONE

INSTALLATION

1.1 Prerequisites

Before installing **opensourceleg** library, you should ensure that you have the following prerequisites installed:

- `Python 3.9` or later
- `pip` package manager

1.2 Installation

To install **opensourceleg** library, you can use `pip` to download and install the package from **PyPI**. Here are the steps:

1. Open a terminal or command prompt.
2. Run the following command to install opensourceleg

```
pip install opensourceleg
```

3. This will download and install the latest version of the opensourceleg library from PyPI.
4. Once the installation is complete, you can verify that the opensourceleg library is installed by running the following command

```
python -c "import opensourceleg; print(opensourceleg.version)"
```

This should print the version number of the **opensourceleg** library.

1.3 Usage

To use the **opensourceleg** library in your Python code, you can import it like this

```
import opensourceleg
```

That's it! You should now be able to use the **opensourceleg** library in your Python projects.

CHAPTER

TWO

TUTORIALS

In this section you will find tutorials on how to use the **opensourceleg** library.

2.1 Adding Joints

In this tutorial, we'll show you how to add a joint to your open-source leg using the **opensourceleg** library.

Step 1: Import the OpenSourceLeg Class

To get started, we need to import the `OpenSourceLeg` class, which provides an interface for controlling the open-source leg.

```
from opensourceleg.osl import OpenSourceLeg
```

Step 2: Create an instance of the OpenSourceLeg Class

Next, we need to create an OpenSourceLeg object. This object represents the open-source leg, provides methods for controlling its joint, and provides methods for a variety of other tasks.

```
osl = OpenSourceLeg(frequency=200, file_name="getting_started.log")
```

This will create an OpenSourceLeg object with a *frequency of 200 Hz* and a log file named `getting_started.log`.

Step 3: Add a Joint Object

To add a joint to the open-source leg, we can use the `add_joint` method of the OpenSourceLeg object.

```
osl.add_joint(name="knee", gear_ratio=41.99, has_loadcell=False)
```

This will add a joint object named `knee` with a gear ratio of `41.99` to the *osl* object. You can also specify the **port** the joint is connected to using the `port` parameter of the `add_joint` method. If you don't specify a port, the joint will be connected to the first available port.

Note: The `has_loadcell` parameter indicates whether or not the actuator has a load cell connected to it via an FFC cable. This feature is only supported by the Dephy actuators. If you are using a TMotor actuator, this parameter should always be set to `False`. We'll discuss the loadcell in more detail in a later tutorial.

You can also add the ankle joint to the open-source leg by calling the `add_joint` method again. If the `port` parameter is not specified, the joint will use the next available port.

```
osl.add_joint(name="ankle", gear_ratio=41.99, has_loadcell=False)
```

Warning: Please ensure that you are powering-on the actuators in the order of initialization, i.e. if you are initializing the knee joint first, then the knee joint should be powered-on first.

Here is the code for this tutorial:

```
from opensourceleg.osl import OpenSourceLeg

osl = OpenSourceLeg(frequency=200, file_name="getting_started.log")
osl.add_joint(name="knee", gear_ratio=41.99, has_loadcell=False)
osl.add_joint(name="ankle", gear_ratio=41.99, has_loadcell=False)
```

2.2 Adding a Loadcell

In this tutorial, we'll show you how to add a load cell to your open-source leg using the *opensourceleg* library.

Step 1: Create an OpenSourceLeg Object

First, we need to create an **OpenSourceLeg** object. This object represents your open-source leg.

```
osl = OpenSourceLeg(frequency=200, file_name="getting_started.log")
```

This will create an **OpenSourceLeg** object with a *frequency of 200 Hz* and a log file named `getting_started.log`.

Step 2: Add a Load Cell Object

To add a load cell to the `osl` object, we can use the *add_loadcell* method.

```
osl.add_loadcell(dephy_mode=False, loadcell_matrix=constants.LOADCELL_MATRIX)
```

This will add a load cell `osl` object with the specified *loadcell_matrix*, which is the calibration matrix for the load cell. This calibration matrix is unique to each load cell and can be found in the load cell's datasheet. The *dephy_mode* argument is set to *False*, which means that the load cell is connected to the Raspberry Pi via the GPIO pins and not to the Dephy actuator using an FFC cable. If you are using a load cell connected to the Dephy actuator, you should set this argument to *True*.

Here is an example of how you would add a load cell to the *osl* object if you were using a load cell connected to the Dephy actuator:

```
osl.add_joint(name="knee", gear_ratio=41.99, has_loadcell=True)
osl.add_loadcell(dephy_mode=True, joint=osl.knee, loadcell_matrix=constants.LOADCELL_
↪MATRIX)
```

This method requires a joint to be added to the *osl* object first. This is because the load cell is connected to the Dephy actuator, which reads the load cell data and streams it to the Raspberry Pi. This joint object is passed to the *add_loadcell* method so that the load cell data can be read from the joint object.

Note: If you are using a different load cell amplifier, your amplifier gain and excitation voltage may be different. You can change these values by passing the `amp_gain` and `exc` arguments to the *add_loadcell* method. The default values for these arguments are `amp_gain=125` and `exc=5`.

Here is the code for this tutorial:

```
import opensourceleg.constants as constants
from opensourceleg.osl import OpenSourceLeg

osl = OpenSourceLeg(frequency=200, file_name="getting_started.log")
osl.add_joint(name="knee", gear_ratio=41.99, has_loadcell=False)
osl.add_joint(name="ankle", gear_ratio=41.99, has_loadcell=False)

osl.add_loadcell(dephy_mode=False, loadcell_matrix=constants.LOADCELL_MATRIX)
```

That's it! You've now added a load cell to your open-source leg using the *opensourceleg* library.

2.3 Reading from Sensors

In this tutorial, we'll show you how to read from the sensors of your open-source leg using the **opensourceleg** library.

Step 1: Importing the *OpenSourceLeg* Class

To use the *OpenSourceLeg* class, we first need to import it from the *opensourceleg.osl* module:

```
from opensourceleg.osl import OpenSourceLeg
```

Step 2: Create an *OpenSourceLeg* Object

Once we have imported the *OpenSourceLeg* class, we can create an instance of the class with the desired frequency and joint configuration:

```
osl = OpenSourceLeg(frequency=200)
osl.add_joint(gear_ratio=9.0)
```

In this code, we create an *OpenSourceLeg* object named *osl* with a frequency of 200 and a joint with a gear ratio of 9.0.

Step 3: Setting Units for the *position* Attribute

We can set the units for the *position* attribute of the *osl* object using the *units* dictionary:

```
osl.units["position"] = "deg"
osl.log.info(osl.units)
```

In this code, we set the units for the *position* attribute to "deg" and log the units to the console.

Step 4: Reading from Sensors

To read from sensors, we can use a *with* block to ensure that the *OpenSourceLeg* object is properly opened and cleaned up after use:

```
with osl:
    osl.knee.set_mode("voltage")

    for t in osl.clock:
        osl.log.info(osl.knee.motor_position)
        osl.update()
```

In this code, we enter a *with* block that sets the mode of the *knee* joint to "voltage". We then loop over the *osl.clock* generator, which generates a sequence of timestamps at the specified frequency, and log the motor position of the *knee* joint to the console at each timestamp. We then call the *osl.update()* method to update the state of the *OpenSourceLeg* object.

> **Warning:** This code assumes that the *OpenSourceLeg* object is properly configured and calibrated, and that the sensors are properly connected and functioning.

Here is the code for this tutorial:

```
from opensourceleg.osl import OpenSourceLeg

osl = OpenSourceLeg(frequency=200)
osl.add_joint(gear_ratio=9.0)

osl.units["position"] = "deg"
osl.log.info(osl.units)

with osl:
    osl.knee.set_mode("voltage")

    for t in osl.clock:
        osl.log.info(osl.knee.motor_position)
        osl.update()
```

2.4 Commanding Voltage

In this tutorial, we'll show you how to use the *OpenSourceLeg* library to control the voltage of a joint.

Step 1: Importing the *OpenSourceLeg* Class

To use the *OpenSourceLeg* class, we first need to import it from the *opensourceleg.osl* module:

```
from opensourceleg.osl import OpenSourceLeg
```

Step 2: Creating an *OpenSourceLeg* Object

Once we have imported the *OpenSourceLeg* class, we can create an instance of the class with the desired frequency and joint configuration:

```
osl = OpenSourceLeg(frequency=200)  # 200 Hz
osl.add_joint(gear_ratio=9.0)
```

In this code, we create an *OpenSourceLeg* object named *osl* with a frequency of 200 Hz and a joint with a gear ratio of 9.0.

Step 3: Setting Units for the *position* Attribute

We can set the units for the *position* attribute of the *osl* object using the *units* dictionary:

```
osl.units["position"] = "deg"
osl.log.info(osl.units)
```

In this code, we set the units for the *position* attribute to "deg" and log the units to the console.

Step 4: Controlling Joint Voltage

To control the voltage of a joint, we can use a *with* block to ensure that the *OpenSourceLeg* object is properly opened and cleaned up after use:

```
with osl:
    osl.knee.set_mode("voltage")

    for t in osl.clock:
        osl.knee.set_voltage(1000)  # mV
        osl.log.info(osl.knee.motor_position)
        osl.update()
```

In this code, we enter a *with* block that sets the mode of the *knee* joint to "voltage". We then loop over the *osl.clock* generator, which generates a sequence of timestamps at the specified frequency, and set the voltage of the *knee* joint to 1000 mV. We then log the motor position of the *knee* joint to the console at each timestamp. We then call the *osl.update()* method to update the state of the *OpenSourceLeg* object.

Warning: This code assumes that the *OpenSourceLeg* object is properly configured and calibrated, and that the joint is properly connected and functioning.

Here is the code for this tutorial:

```
from opensourceleg.osl import OpenSourceLeg

osl = OpenSourceLeg(frequency=200)  # 200 Hz
osl.add_joint(gear_ratio=9.0)

osl.units["position"] = "deg"
osl.log.info(osl.units)

with osl:
    osl.knee.set_mode("voltage")

    for t in osl.clock:
        osl.knee.set_voltage(1000)  # mV
        osl.log.info(osl.knee.motor_position)
        osl.update()
```

2.5 Commanding Current

In this tutorial, we'll show you how to use the *OpenSourceLeg* library to control the current of a joint.

Step 1: Import the OpenSourceLeg Class

To use the *OpenSourceLeg* class, we first need to import it from the *opensourceleg.osl* module:

```
from opensourceleg.osl import OpenSourceLeg
```

Step 2: Create an instance of the OpenSourceLeg Class and add a Joint Object

Once we have imported the *OpenSourceLeg* class, we can create an instance of the class with the desired frequency and joint configuration:

```
osl = OpenSourceLeg(frequency=200)  # 200 Hz
osl.add_joint(gear_ratio=9.0)
```

In this code, we create an *OpenSourceLeg* object named *osl* with a frequency of 200 Hz and a joint with a gear ratio of 9.0.

Step 3: Setting Units for the *position* Attribute

We can set the units for the *position* attribute of the *osl* object using the *units* dictionary:

```
osl.units["position"] = "deg"
osl.log.info(osl.units)
```

In this code, we set the units for the *position* attribute to "deg" and log the units to the console.

Step 4: Controlling Joint Current

To control the current of a joint, we can use a *with* block to ensure that the *OpenSourceLeg* object is properly opened and cleaned up after use:

```
with osl:
    osl.knee.set_mode("current")

    for t in osl.clock:
        osl.knee.set_current(400)  # 400 mA
        osl.log.info(osl.knee.motor_position)
        osl.update()
```

In this code, we enter a *with* block that sets the mode of the *knee* joint to "current". We then loop over the *osl.clock* generator, which generates a sequence of timestamps at the specified frequency, and set the current of the *knee* joint to 400 mA. We then log the motor position of the *knee* joint to the console at each timestamp. We then call the *osl.update()* method to update the state of the *OpenSourceLeg* object.

> **Warning:** This code assumes that the *OpenSourceLeg* object is properly configured and calibrated, and that the joint is properly connected and functioning.

Here is the code for this tutorial:

```
from opensourceleg.osl import OpenSourceLeg

osl = OpenSourceLeg(frequency=200)  # 200 Hz
osl.add_joint(gear_ratio=9.0)

osl.units["position"] = "deg"
osl.log.info(osl.units)

with osl:
    osl.knee.set_mode("current")

    for t in osl.clock:
        osl.knee.set_current(400)  # 400 mA
        osl.log.info(osl.knee.motor_position)
        osl.update()
```

2.6 Commanding Position

In this tutorial, we'll show you how to control your open-source leg in position mode usin the *opensourceleg* library.

Step 1: Importing the *OpenSourceLeg* Class

To use the *OpenSourceLeg* class, we first need to import it from the *opensourceleg.osl* module:

```
from opensourceleg.osl import OpenSourceLeg
```

Step 2: Creating an *OpenSourceLeg* Object

Once we have imported the *OpenSourceLeg* class, we can create an instance of the class with the desired frequency and joint configuration:

```
osl = OpenSourceLeg(frequency=200)  # 200 Hz
osl.add_joint(gear_ratio=9.0)
```

In this code, we create an *OpenSourceLeg* object named *osl* with a frequency of 200 Hz and a joint with a gear ratio of 9.0.

Step 3: Setting Units for the *position* Attribute

We can set the units for the *position* attribute of the *osl* object using the *units* dictionary:

```
osl.units["position"] = "deg"
osl.log.info(osl.units)
```

In this code, we set the units for the *position* attribute to "deg" and log the units to the console.

Step 4: Controlling Joint Position

To control the position of a joint, we can use a *with* block to ensure that the *OpenSourceLeg* object is properly opened and cleaned up after use:

```
set_point = 50  # motor ticks

with osl:
    osl.knee.set_mode("position")
    osl.knee.set_motor_position(osl.knee.motor_position + set_point)

    for t in osl.clock:
        osl.log.info(osl.knee.motor_position)
        osl.update()
```

In this code, we enter a *with* block that sets the mode of the *knee* joint to "position", and set the motor position of the *knee* joint to the current position plus the *set_point* value. We then loop over the *osl.clock* generator, which generates a sequence of timestamps at the specified frequency, and log the motor position of the *knee* joint to the console at each timestamp. We then call the *osl.update()* method to update the state of the *OpenSourceLeg* object.

Warning: This code assumes that the *OpenSourceLeg* object is properly configured and calibrated, and that the joint is properly connected and functioning.

Here is the code for this tutorial:

```
from opensourceleg.osl import OpenSourceLeg

osl = OpenSourceLeg(frequency=200)  # 200 Hz
osl.add_joint(gear_ratio=9.0)

osl.units["position"] = "deg"
osl.log.info(osl.units)

set_point = 50  # motor ticks

with osl:
    osl.knee.set_mode("position")

    osl.knee.set_motor_position(osl.knee.motor_position + set_point)

    for t in osl.clock:
        osl.log.info(osl.knee.motor_position)
        osl.update()
```

2.7 Commanding Impedance

In this tutorial, we'll show you how to use the *OpenSourceLeg* library to control the impedance of a joint.

Step 1: Importing the *OpenSourceLeg* Class

To use the *OpenSourceLeg* class, we first need to import it from the *opensourceleg.osl* module:

```
from opensourceleg.osl import OpenSourceLeg
```

Step 2: Creating an *OpenSourceLeg* Object

Once we have imported the *OpenSourceLeg* class, we can create an instance of the class with the desired frequency and joint configuration:

```
osl = OpenSourceLeg(frequency=200)  # 200 Hz
osl.add_joint(gear_ratio=9.0)
```

In this code, we create an *OpenSourceLeg* object named *osl* with a frequency of 200 Hz and a joint with a gear ratio of 9.0.

Step 3: Setting Units for the *position* Attribute

We can set the units for the *position* attribute of the *osl* object using the *units* dictionary:

```
osl.units["position"] = "deg"
osl.log.info(osl.units)
```

In this code, we set the units for the *position* attribute to "deg" and log the units to the console.

Step 4: Controlling Joint Impedance

To control the impedance of a joint, we can use a *with* block to ensure that the *OpenSourceLeg* object is properly opened and cleaned up after use:

```
test_stiffness_value = 20  # Nm/rad
test_damping_value = 20  # Nm/rad/s
set_point = 50  # motor ticks

with osl:
    osl.knee.set_mode("impedance")
    osl.knee.set_joint_impedance(K=test_stiffness_value, B=test_damping_value)
    osl.knee.set_motor_position(osl.knee.motor_position + set_point)

    for t in osl.clock:
        osl.log.info(osl.knee.motor_position)
        osl.update()
```

In this code, we enter a *with* block that sets the mode of the *knee* joint to "impedance" and set the motor position of the *knee* joint to the current position plus a set point of 50 degrees. We then set the impedance of the *knee* joint using the *set_joint_impedance* method, with a stiffness of 20 Nm/rad and a damping of 20 Nm/rad/s. We then loop over the *osl.clock* generator, which generates a sequence of timestamps at the specified frequency, and log the motor position of the *knee* joint to the console at each timestamp. We then call the *osl.update()* method to update the state of the *OpenSourceLeg* object.

Note: When setting the impedance gains, we can also use the *set_motor_impedance* method to set the impedance of the motor instead of the joint. The difference between the two methods is that the *set_joint_impedance* method first divides the initial value by the gear ratio squared then sets the gains, while the *set_motor_impedance* method simply sets the gains. The impedance of the joint is the impedance of the motor plus the impedance of the joint.

Note: We can also set the gains directly using "convert" methods. We can set the impedance gains using the *set_impedance_gains* method and the *convert_to_joint_impedance* method for joint stiffness and damping using real units. Alternatively, we can set impedance gains using the *set_impedance_gains* method and the *convert_to_motor_impedance* method for setting the motor stiffness and damping values with real units. Finally, we can set the impedance gains using the *set_impedance_gains* method and the *convert_to_pid_impedance* method for setting the motor sitffness and damping using the built-in PID controller.

Warning: This code assumes that the *OpenSourceLeg* object is properly configured and calibrated, and that the joint is properly connected and functioning.

Here is the code for this tutorial:

```
from opensourceleg.osl import OpenSourceLeg

osl = OpenSourceLeg(frequency=200)  # 200 Hz
osl.add_joint(gear_ratio=9.0)

osl.units["position"] = "deg"
osl.log.info(osl.units)

test_stiffness_value = 20  # Nm/rad
test_damping_value = 20  # Nm/rad/s

set_point = 50  # motor ticks

with osl:

    osl.knee.set_mode("impedance")
    osl.knee.set_joint_impedance(K=test_stiffness_value, B=test_damping_value)
    osl.knee.set_motor_position(osl.knee.motor_position + set_point)

    for t in osl.clock:
        osl.log.info(osl.knee.motor_position)
        osl.update()
```

CHAPTER

THREE

ACTUATORS

class opensourceleg.actuators.**ActpackMode**(*control_mode: c_int*, *device:* DephyActpack)

Base class for Actpack modes

Parameters

- **control_mode** (*c_int*) – Control mode
- **device** (DephyActpack) – Dephy Actpack

enter() → None

Calls the entry callback

exit() → None

Calls the exit callback

property has_gains: bool

Whether the mode has gains

Returns
True if the mode has gains, False otherwise

Return type
bool

property mode: c_int

Control mode

Returns
Control mode

Return type
c_int

transition(*to_state:* ActpackMode) → None

Transition to another mode. Calls the exit callback of the current mode and the entry callback of the new mode.

Parameters
to_state (ActpackMode) – Mode to transition to

class opensourceleg.actuators.**ControlModes**(*voltage: c_int = c_int(1)*, *current: c_int = c_int(2)*, *position: c_int = c_int(0)*, *impedance: c_int = c_int(3)*)

Control modes for the Dephy Actpack.

Available modes are Voltage, Current, Position, Impedance.

class opensourceleg.actuators.**CurrentMode**(*device:* DephyActpack)

class opensourceleg.actuators.**DephyActpack**(*port: str = '/dev/ttyACM0'*, *baud_rate: int = 230400*, *frequency: int = 500*, *logger: ~opensourceleg.logger.Logger = <Logger opensourceleg.logger (DEBUG)>*, *units: ~opensourceleg.units.UnitsDefinition = {'acceleration': 'rad/s^2', 'current': 'mA', 'damping': 'N/(rad/s)', 'force': 'N', 'gravity': 'm/s^2', 'length': 'm', 'mass': 'kg', 'position': 'rad', 'stiffness': 'N/rad', 'temperature': 'C', 'time': 's', 'torque': 'N-m', 'velocity': 'rad/s', 'voltage': 'mV'}*, *debug_level: int = 0*, *dephy_log: bool = False*)

Class for the Dephy Actpack

Parameters
Device (*_type_*) – _description_

Raises

- **KeyError** – _description_
- **ValueError** – _description_
- **KeyError** – _description_

Returns
description

Return type
type

set_current(*value: float*) → None

Sets the q axis current

Parameters
value (*float*) – The current to set

set_current_gains(*kp: int = 40*, *ki: int = 400*, *ff: int = 128*) → None

Sets the current gains

Parameters

- **kp** (*int*) – The proportional gain
- **ki** (*int*) – The integral gain
- **ff** (*int*) – The feedforward gain

set_impedance_gains(*kp: int = 40*, *ki: int = 400*, *K: int = 200*, *B: int = 400*, *ff: int = 128*) → None

Sets the impedance gains

Parameters

- **kp** (*int*) – The proportional gain
- **ki** (*int*) – The integral gain
- **K** (*int*) – The spring constant
- **B** (*int*) – The damping constant
- **ff** (*int*) – The feedforward gain

set_motor_position(*position: float*) → None

Sets the motor position

Parameters
position (*float*) – The position to set

set_motor_torque(*torque: float*) → None

Sets the motor torque

Parameters
torque (*float*) – The torque to set

set_position_gains(*kp: int = 50*, *ki: int = 0*, *kd: int = 0*) → None

Sets the position gains

Parameters

- **kp** (*int*) – The proportional gain
- **ki** (*int*) – The integral gain
- **kd** (*int*) – The derivative gain

set_voltage(*value: float*) → None

Sets the q axis voltage

Parameters
value (*float*) – The voltage to set

class opensourceleg.actuators.**ImpedanceMode**(*device:* DephyActpack)

class opensourceleg.actuators.**PositionMode**(*device:* DephyActpack)

class opensourceleg.actuators.**VoltageMode**(*device:* DephyActpack)

CHAPTER

FOUR

CONSTANTS

- `PI` **float** = 3.14159
- `MOTOR_COUNT_PER_REV` **float** = 16384
- `NM_PER_AMP` **float** = 0.1133
- `NM_PER_MILLIAMP` **float** = `NM_PER_AMP` / 1000
- `RAD_PER_COUNT` **float** = 2 * `PI` / `MOTOR_COUNT_PER_REV`
- `RAD_PER_DEG` **float** = `PI` / 180
- `RAD_PER_SEC_GYROLSB` **float** = `PI` / 180 / 32.8
- `M_PER_SEC_SQUARED_ACCLSB` **float** = 9.80665 / 8192
- `IMPEDANCE_A` **float** = 0.00028444
- `IMPEDANCE_C` **float** = 0.0007812
- `NM_PER_RAD_TO_K` **float** = `RAD_PER_COUNT` / `IMPEDANCE_C` * 1e3 / `NM_PER_AMP`
- `NM_S_PER_RAD_TO_B` **float** = `RAD_PER_DEG` / `IMPEDANCE_A` * 1e3 / `NM_PER_AMP`
- `MAX_CASE_TEMPERATURE` **float** = 80.0

CHAPTER

FIVE

CONTROL

class opensourceleg.control.**Gains**(*kp: int = 0*, *ki: int = 0*, *kd: int = 0*, *K: int = 0*, *B: int = 0*, *ff: int = 0*)

Dataclass for controller gains

Parameters

- **kp** (*int*) – Proportional gain
- **ki** (*int*) – Integral gain
- **kd** (*int*) – Derivative gain
- **K** (*int*) – Stiffness of the impedance controller
- **B** (*int*) – Damping of the impedance controller
- **ff** (*int*) – Feedforward gain

CHAPTER

SIX

JOINTS

class opensourceleg.joints.**Joint**(*name: str = 'knee'*, *port: str = '/dev/ttyACM0'*, *baud_rate: int = 230400*, *frequency: int = 500*, *gear_ratio: float = 41.4999*, *has_loadcell: bool = False*, *logger: ~opensourceleg.logger.Logger = <Logger opensourceleg.logger (DEBUG)>*, *units: ~opensourceleg.units.UnitsDefinition = {'acceleration': 'rad/s^2', 'current': 'mA', 'damping': 'N/(rad/s)', 'force': 'N', 'gravity': 'm/s^2', 'length': 'm', 'mass': 'kg', 'position': 'rad', 'stiffness': 'N/rad', 'temperature': 'C', 'time': 's', 'torque': 'N-m', 'velocity': 'rad/s', 'voltage': 'mV'}*, *debug_level: int = 0*, *dephy_log: bool = False*)

Bases: *DephyActpack*

home(*homing_voltage: int = 2000*, *homing_frequency: int = 100*) → None

This method homes the joint by moving it to the zero position. The zero position is defined as the position where the joint is fully extended. This method will also make an encoder map if one does not exist.

Parameters

- **homing_voltage** (*int*) – voltage in mV to use for homing
- **homing_frequency** (*int*) – frequency in Hz to use for homing

make_encoder_map() → None

This method makes a lookup table to calculate the position measured by the joint encoder. This is necessary because the magnetic output encoders are nonlinear. By making the map while the joint is unloaded, joint position calculated by motor position * gear ratio should be the same as the true joint position.

Output from this function is a file containing a_i values parameterizing the map

Eqn: position = sum from i=0^5 (a_i*counts^i)

Author: Kevin Best, PhD
U-M Neurobionics Lab Gitub: tkevinbest, https://github.com/tkevinbest

set_joint_impedance(*kp: int = 40*, *ki: int = 400*, *K: float = 0.08922*, *B: float = 0.003807*, *ff: int = 128*) → None

Set the impedance gains of the joint in real units: Nm/rad and Nm/rad/s.

Conversion:
K_motor = K_joint / (gear_ratio ** 2) B_motor = B_joint / (gear_ratio ** 2)

Parameters

- **kp** (*int*) – Proportional gain. Defaults to 40.
- **ki** (*int*) – Integral gain. Defaults to 400.

- **K** (*float*) – Spring constant. Defaults to 0.08922 Nm/rad.
- **B** (*float*) – Damping constant. Defaults to 0.0038070 Nm/rad/s.
- **ff** (*int*) – Feedforward gain. Defaults to 128.

set_max_temperature(*temperature: float*) → None

Set the maximum temperature of the motor.

Parameters

temperature (*float*) – temperature in degrees Celsius

set_motor_impedance(*kp: int = 40*, *ki: int = 400*, *K: float = 0.08922*, *B: float = 0.003807*, *ff: int = 128*) → None

Set the impedance gains of the motor in real units: Nm/rad and Nm/rad/s.

Parameters

- **kp** (*int*) – Proportional gain. Defaults to 40.
- **ki** (*int*) – Integral gain. Defaults to 400.
- **K** (*float*) – Spring constant. Defaults to 0.08922 Nm/rad.
- **B** (*float*) – Damping constant. Defaults to 0.0038070 Nm/rad/s.
- **ff** (*int*) – Feedforward gain. Defaults to 128.

set_output_position(*position: float*) → None

Set the output position of the joint. This is the desired position of the joint, not the motor.

Parameters

position (*float*) – position in user-defined units

set_output_torque(*torque: float*) → None

Set the output torque of the joint. This is the torque that is applied to the joint, not the motor.

Parameters

torque (*float*) – torque in user-defined units

CHAPTER

SEVEN

LOADCELL

class opensourceleg.loadcell.**StrainAmp**(*bus*, *I2C_addr=102*)

A class to directly manage the 6ch strain gauge amplifier over I2C. Author: Mitry Anderson

read_compressed_strain()

Used for more recent versions of strain amp firmware

read_uncompressed_strain()

Used for an older version of the strain amp firmware (at least pre-2017)

static strain_data_to_wrench(*unpacked_strain*, *loadcell_matrix*, *loadcell_zero*, *exc=5*, *gain=125*)

Converts strain values between 0 and 4095 to a wrench in N and Nm

static unpack_compressed_strain(*data*)

Used for more recent versions of strainamp firmware

static unpack_uncompressed_strain(*data*)

Used for an older version of the strain amp firmware (at least pre-2017)

update()

Called to update data of strain amp. Also returns data.

static wrench_to_strain_data(*measurement*, *loadcell_matrix*, *exc=5*, *gain=125*)

Wrench in N and Nm to the strain values that would give that wrench

CHAPTER

EIGHT

LOGGER

class opensourceleg.logger.**Logger**(*file_path: str = './osl'*, *log_format: str = '[%(asctime)s] %(levelname)s: %(message)s'*)

Bases: `Logger`

Logger class is a class that logs attributes from a class to a csv file

__init__(self, class_instance

object, file_path: str, logger: logging.Logger = None) -> None

log(*self*) → None

add_attributes(*class_instance: object*, *attributes_str: list[str]*) → None

Configures the logger to log the attributes of a class

Parameters

- **class_instance** (*object*) – Class instance to log the attributes of
- **attributes_str** (*list[str]*) – List of attributes to log

close() → None

Closes the csv file

data() → None

Logs the attributes of the class instance to the csv file

set_file_level(*level: str = 'DEBUG'*) → None

Sets the level of the logger

Parameters
level (*str*) – Level of the logger

set_stream_level(*level: str = 'INFO'*) → None

Sets the level of the logger

Parameters
level (*str*) – Level of the logger

CHAPTER

NINE

OPEN-SOURCE LEG

class opensourceleg.osl.**OpenSourceLeg**(*frequency: int = 200*, *file_name: str = 'osl'*)

Bases: `object`

OSL class: This class is the main class for the Open Source Leg project. It contains all the necessary functions to control the leg.

Returns
none

Return type
none

add_joint(*name: str = 'knee'*, *port: str | None = None*, *baud_rate: int = 230400*, *gear_ratio: float = 1.0*, *has_loadcell: bool = False*, *debug_level: int = 0*, *dephy_log: bool = False*) → None

Add a joint to the OSL object.

Parameters

- **name** (*str, optional*) – The name of the joint, by default "knee"
- **port** (*str, optional*) – The serial port that the joint is connected to, by default None. If None, the first active port will be used.
- **baud_rate** (*int, optional*) – The baud rate of the serial communication, by default 230400
- **gear_ratio** (*float, optional*) – The gear ratio of the joint, by default 1.0
- **has_loadcell** (*bool, optional*) – Whether the joint has a loadcell, by default False
- **debug_level** (*int, optional*) – The debug level of the joint, by default 0
- **dephy_log** (*bool, optional*) – Whether to log the joint data to the dephy log, by default False

add_loadcell(*dephy_mode: bool = False*, *joint: Joint | None = None*, *amp_gain: float = 125.0*, *exc: float = 5.0*, *loadcell_matrix=array([[-3.872600e+01, -1.817747e+03, 9.849000e+00, 4.337400e+01, -4.454000e+01, 1.824670e+03], [-8.616000e+00, 1.041149e+03, 1.886100e+01, -2.098822e+03, 3.179400e+01, 1.058623e+03], [-1.047168e+03, 8.639000e+00, -1.047282e+03, -2.070000e+01, -1.073088e+03, -8.923000e+00], [2.057600e+01, -4.000000e-02, -2.460000e-01, 5.540000e-01, -2.140800e+01, -4.760000e-01], [-1.213400e+01, -1.108000e+00, 2.436100e+01, 2.300000e-02, -1.214100e+01, 7.920000e-01], [-6.510000e-01, -2.828700e+01, 2.200000e-02, -2.523000e+01, 4.730000e-01, -2.730700e+01]])*) → None

Add a loadcell to the OSL object.

Parameters

- **dephy_mode** (*bool, optional*) – Whether the loadcell is in dephy mode ie. connected to the dephy actpack with an FFC cable, by default False
- **joint** (*Joint, optional*) – The joint that the loadcell is attached to, by default None
- **amp_gain** (*float, optional*) – The amplifier gain of the loadcell, by default 125.0
- **exc** (*float, optional*) – The excitation voltage of the loadcell, by default 5.0
- **loadcell_matrix** (*np.ndarray, optional*) – The loadcell calibration matrix, by default None

add_state_machine(*spoof: bool = False*) → None

Add a state machine to the OSL object.

Parameters

spoof (*bool, optional*) – If True, the state machine will spoof the state transitions–ie, it will not check the criteria for transitioning but will instead transition after the minimum time spent in state has elapsed. This is useful for testing. Defaults to False.

add_tui(*configuration: str = 'state_machine'*, *layout: str = 'vertical'*) → None

Add a Terminal User Interface (TUI) to the OSL object. The TUI is used to visualize the data from the OSL and to send commands to the OSL. Additionally, the TUI also has a timer built-in to govern the frequency of the control loop, which is less accurate than the "osl.clock" (SoftRealTimeLoop) object but is more convenient.

Note: The timer/interface runs in a separate thread, so it does not affect the control loop. This causes the code to not respond to keyboard interrupts (Ctrl+C) when the TUI is running.

Parameters

- **configuration** (*str, optional*) – The configuration of the TUI, by default "state_machine"
- **layout** (*str, optional*) – The layout of the TUI, by default "vertical"

run(*set_state_machine_parameters: bool = False*, *log_data: bool = False*) → None

Run the OpenSourceLeg instance: update the joints, loadcell, and state machine. If the instance has a TUI, run the TUI. If the instance has a state machine and if set_state_machine_parameters is True, set the joint impedance gains and equilibrium angles to the current state's values.

Parameters

set_state_machine_parameters (*bool, optional*) – Whether to set the joint impedance gains and equilibrium angles to the current state's values, by default False

CHAPTER

TEN

STATE MACHINE

class opensourceleg.state_machine.**Event**(*name*)

Event class

class opensourceleg.state_machine.**FromToTransition**(*event: Event*, *source: State*, *destination: State*, *callback: Callable[[Any], bool] | None = None*)

class opensourceleg.state_machine.**Idle**

class opensourceleg.state_machine.**State**(*name: str = 'state'*, *is_knee_active: bool = False*, *knee_stiffness: float = 0.0*, *knee_damping: float = 0.0*, *knee_equilibrium_angle: float = 0.0*, *is_ankle_active: bool = False*, *ankle_stiffness: float = 0.0*, *ankle_damping: float = 0.0*, *ankle_equilibrium_angle: float = 0.0*, *minimum_time_in_state: float = 2.0*)

A class to represent a state in a finite state machine.

Parameters

- **name** (*str*) – Name of the state
- **is_knee_active** (*bool*) – Whether the knee is active. Default: False
- **knee_stiffness** (*float*) – Knee stiffness in Nm/rad
- **knee_damping** (*float*) – Knee damping in Nm/rad/sec
- **knee_equilibrium_angle** (*float*) – Knee equilibrium angle
- **is_ankle_active** (*bool*) – Whether the ankle is active. Default: False
- **ankle_stiffness** (*float*) – Ankle stiffness in Nm/rad
- **ankle_damping** (*float*) – Ankle damping in Nm/rad/sec
- **ankle_equilibrium_angle** (*float*) – Ankle equilibrium angle
- **minimum_time_in_state** (*float*) – Minimum time spent in the state in seconds. Default: 2.0

Note: The knee and ankle impedance parameters are only used if the corresponding joint is active. You can also set custom data using the *set_custom_data* method.

---

get_custom_data(*key: str*) → Any

Get custom data for the state. The custom data is a dictionary that can be used to store any data you want to associate with the state.

Parameters
key (*str*) – Key of the data

Returns
Value of the data

Return type
Any

`make_ankle_active()`

Make the ankle active

Note: The ankle impedance parameters are only used if the ankle is active.

`make_knee_active()`

Make the knee active

Note: The knee impedance parameters are only used if the knee is active.

`set_ankle_impedance_paramters(theta, k, b)` → None

Set the ankle impedance parameters

Parameters

- **theta** (*float*) – Equilibrium angle of the ankle joint
- **k** (*float*) – Stiffness of the ankle joint
- **b** (*float*) – Damping of the ankle joint

Note: The ankle impedance parameters are only used if the ankle is active. You can make the ankle active by calling the *make_ankle_active* method.

`set_custom_data(key: str, value: Any)` → None

Set custom data for the state. The custom data is a dictionary that can be used to store any data you want to associate with the state.

Parameters

- **key** (*str*) – Key of the data
- **value** (*Any*) – Value of the data

`set_knee_impedance_paramters(theta, k, b)` → None

Set the knee impedance parameters

Parameters

- **theta** (*float*) – Equilibrium angle of the knee joint
- **k** (*float*) – Stiffness of the knee joint
- **b** (*float*) – Damping of the knee joint

Note: The knee impedance parameters are only used if the knee is active. You can make the knee active by calling the *make_knee_active* method.

---

set_minimum_time_spent_in_state(*time: float*) → None

Set the minimum time spent in the state

Parameters
time (*float*) – Minimum time spent in the state in seconds

class opensourceleg.state_machine.**StateMachine**(*osl=None*, *spoof: bool = False*)

State Machine class

Parameters

- **osl** (*Any*) – The OpenSourceLeg object.
- **spoof** (*bool*) – If True, the state machine will spoof the state transitions–ie, it will not check the criteria for transitioning but will instead transition after the minimum time spent in state has elapsed. This is useful for testing. Defaults to False.

current_state

The current state of the state machine.

Type
State

current_state_name

The name of the current state of the state machine.

Type
str

states

The list of states in the state machine.

Type
list[*State*]

is_spoofing

Whether or not the state machine is spoofing the state transitions.

Type
bool

add_state(*state:* State, *initial_state: bool = False*) → None

Add a state to the state machine.

Parameters

- **state** (State) – The state to be added.
- **initial_state** (*bool, optional*) – Whether the state is the initial state, by default False

add_transition(*source:* State, *destination:* State, *event:* Event, *callback: Callable[[Any], bool] | None = None*) → *Transition* | None

Add a transition to the state machine.

Parameters

- **source** (State) – The source state.
- **destination** (State) – The destination state.
- **event** (Event) – The event that triggers the transition.

- **callback** (*Callable[[Any], bool], optional*) – A callback function that returns a boolean value, which determines whether the transition is valid, by default None

class opensourceleg.state_machine.**Transition**(*event:* Event, *source:* State, *destination:* State, *callback: Callable[[Any], bool] | None = None*)

Transition class

CHAPTER

ELEVEN

THERMAL

class opensourceleg.thermal.**ThermalModel**(*ambient: float = 21*, *params: dict = {}*, *temp_limit_windings: float = 115*, *soft_border_C_windings: float = 15*, *temp_limit_case: float = 80*, *soft_border_C_case: float = 5*)

Thermal model of a motor developed by Jianping Lin and Gray C. Thomas @U-M Locomotion Lab, directed by Dr. Robert Gregg

Assumptions:

1: The motor is a lumped system with two thermal nodes: the winding and the case. 2: The winding and the case are assumed to be in thermal equilibrium with the ambient. 3: The winding and the case are assumed to be in thermal equilibrium with each other.

Equations:

1: C_w * dT_w/dt = (I^2)R + (T_c-T_w)/R_WC 2: C_c * dT_c/dt = (T_w-T_c)/R_WC + (T_w-T_a)/R_CA

where:

C_w: Thermal capacitance of the winding C_c: Thermal capacitance of the case R_WC: Thermal resistance between the winding and the case R_CA: Thermal resistance between the case and the ambient T_w: Temperature of the winding T_c: Temperature of the case T_a: Temperature of the ambient I: Current R: Resistance

Implementation:

1: The model is updated at every time step with the current and the ambient temperature. 2: The model can be used to predict the temperature of the winding and the case at any time step. 3: The model can also be used to scale the torque based on the temperature of the winding and the case.

Parameters

- **ambient** (*float*) – Ambient temperature in Celsius. Defaults to 21.
- **params** (*dict*) – Dictionary of parameters. Defaults to dict().
- **temp_limit_windings** (*float*) – Maximum temperature of the windings in Celsius. Defaults to 115.
- **soft_border_C_windings** (*float*) – Soft border of the windings in Celsius. Defaults to 15.
- **temp_limit_case** (*float*) – Maximum temperature of the case in Celsius. Defaults to 80.
- **soft_border_C_case** (*float*) – Soft border of the case in Celsius. Defaults to 5.

update(*dt: float = 0.005*, *motor_current: float = 0*) → None

Updates the temperature of the winding and the case based on the current and the ambient temperature.

Parameters

- **dt** (*float*) – Time step in seconds. Defaults to 1/200.
- **motor_current** (*float*) – Current in mA. Defaults to 0.

Dynamics:
1: self.C_w * d self.T_w /dt = (I^2)R + (self.T_c-self.T_w)/self.R_WC 2: self.C_c * d self.T_c /dt = (self.T_w-self.T_c)/self.R_WC + (self.T_w-self.T_a)/self.R_CA

update_and_get_scale(*dt*, *motor_current: float = 0*, *FOS=3.0*)

Updates the temperature of the winding and the case based on the current and the ambient temperature and returns the scale factor for the torque.

Parameters

- **dt** (*float*) – Time step in seconds.
- **motor_current** (*float*) – Current in mA. Defaults to 0.
- **FOS** (*float*) – Factor of safety. Defaults to 3.0.

Returns
Scale factor for the torque.

Return type
float

Dynamics:
1: self.C_w * d self.T_w /dt = (I^2)R + (self.T_c-self.T_w)/self.R_WC 2: self.C_c * d self.T_c /dt = (self.T_w-self.T_c)/self.R_WC + (self.T_w-self.T_a)/self.R_CA

CHAPTER

TWELVE

TUI

CHAPTER

THIRTEEN

UNITS

class opensourceleg.units.**UnitsDefinition**

UnitsDefinition class is a dictionary with set and get methods that checks if the keys are valid

__setitem__(key

str, value: dict) -> None

__getitem__(key

str) -> dict

convert(value

float, attribute: str) -> None

convert_from_default_units(*value: float*, *attribute: str*) → float

convert a value from the default unit to another unit

Parameters

- **value** (*float*) – Value to be converted
- **attribute** (*str*) – Attribute to be converted

Returns
Converted value in the default unit

Return type
float

convert_to_default_units(*value: float*, *attribute: str*) → float

convert a value from one unit to the default unit

Parameters

- **value** (*float*) – Value to be converted
- **attribute** (*str*) – Attribute to be converted

Returns
Converted value in the default unit

Return type
float

CHAPTER

FOURTEEN

UTILITIES

class opensourceleg.utilities.**CSVLog**(*file_name*, *variable_name*, *container_names*)

Logging class to make writing to a CSV file easier. See if __name__ == "__main__" for an example. At instantiation, pass a list of lists corresponding to the variable names you wish to log, as well as the name of their containers. The container name is prepended in the log so you know which object the variable came from. These variables should live as attributes within some object accessible to the main loop. To update the log, simply call log.update((obj1, obj2, ...)).

Author: Kevin Best https://github.com/tkevinbest

class opensourceleg.utilities.**EdgeDetector**(*bool_in*)

Used to calculate rising and falling edges of a digital signal in real time. Call edgeDetector.update(digitalSignal) to update the detector. Then read edgeDetector.rising_edge or falling edge to know if the event occurred.

Author: Kevin Best https://github.com/tkevinbest

class opensourceleg.utilities.**LoopKiller**(*fade_time=0.0*)

Soft Realtime Loop—a class designed to allow clean exits from infinite loops with the potential for post-loop cleanup operations executing.

The Loop Killer object watches for the key shutdown signals on the UNIX operating system (which runs on the PI) when it detects a shutdown signal, it sets a flag, which is used by the Soft Realtime Loop to stop iterating. Typically, it detects the CTRL-C from your keyboard, which sends a SIGTERM signal.

the function_in_loop argument to the Soft Realtime Loop's blocking_loop method is the function to be run every loop. A typical usage would set function_in_loop to be a method of an object, so that the object could store program state. See the 'ifmain' for two examples.

This library will soon be hosted as a PIP module and added as a python dependency. # https://github.com/UM-LoCoLab/NeuroLocoMiddleware/blob/main/SoftRealtimeLoop.py

Author: Gray C. Thomas, Ph.D https://github.com/GrayThomas, https://graythomas.github.io

class opensourceleg.utilities.**SaturatingRamp**(*loop_frequency=100*, *ramp_time=1.0*)

Creates a signal that ramps between 0 and 1 at the specified rate. Looks like a trapezoid in the time domain Used to slowly enable joint torque for smooth switching at startup. Call saturatingRamp.update() to update the value of the ramp and return the value. Can also access saturatingRamp.value without updating.

Example usage:

ramp = saturatingRamp(100, 1.0)

 # In loop

torque = torque * ramp.update(enable_ramp)

Author: Kevin Best https://github.com/tkevinbest

update(*enable_ramp=False*)

Updates the ramp value and returns it as a float. If enable_ramp is true, ramp value increases Otherwise decreases.

Example usage:
torque = torque * ramp.update(enable_ramp)

Parameters
enable_ramp (*bool, optional*) – If enable_ramp is true, ramp value increases. Defaults to False.

Returns
Scalar between 0 and 1.

Return type
value (float)

class opensourceleg.utilities.**SoftRealtimeLoop**(*dt=0.001*, *report=False*, *fade=0.0*)

Soft Realtime Loop—a class designed to allow clean exits from infinite loops with the potential for post-loop cleanup operations executing.

The Loop Killer object watches for the key shutdown signals on the UNIX operating system (which runs on the PI) when it detects a shutdown signal, it sets a flag, which is used by the Soft Realtime Loop to stop iterating. Typically, it detects the CTRL-C from your keyboard, which sends a SIGTERM signal.

the function_in_loop argument to the Soft Realtime Loop's blocking_loop method is the function to be run every loop. A typical usage would set function_in_loop to be a method of an object, so that the object could store program state. See the 'ifmain' for two examples.

This library will soon be hosted as a PIP module and added as a python dependency. https://github.com/UM-LoCoLab/NeuroLocoMiddleware/blob/main/SoftRealtimeLoop.py

Author: Gray C. Thomas, Ph.D # https://github.com/GrayThomas, https://graythomas.github.io

opensourceleg.utilities.**get_active_ports**()

Lists active serial ports.

We welcome contributions to the **opensourceleg** library! Whether you want to report a bug, request a feature, or submit a pull request, we appreciate your help in making this library better.

CHAPTER

FIFTEEN

REPORTING BUGS

If you encounter a bug or have a feature request, please open an issue on the **opensourceleg** GitHub repository. When reporting an issue, please include as much detail as possible, including:

- A clear and descriptive title
- A detailed description of the issue or feature request
- Steps to reproduce the issue (if applicable)
- Any error messages or stack traces (if applicable)
- Your operating system and Python version

CHAPTER

SIXTEEN

CONTRIBUTING CODE

If you want to contribute code to **opensourceleg**, please follow these steps:

1. Fork the **opensourceleg** repository on GitHub.
2. Clone your forked repository to your local machine.
3. Create a new branch for your changes.
4. Make your changes and commit them with clear and descriptive commit messages.
5. Push your changes to your forked repository.
6. Open a pull request on the **opensourceleg** repository.

When submitting a pull request, please include:

- A clear and descriptive title
- A detailed description of the changes you made
- Any relevant issue numbers (if applicable)

We will review your pull request as soon as possible and provide feedback if necessary.

CHAPTER

SEVENTEEN

CODE STYLE

Please follow the `PEP-8` style guide when contributing code to **opensourceleg**. We also recommend using an automated code formatter like `black` to ensure consistent formatting.

CHAPTER

EIGHTEEN

TESTING

Please ensure that your code changes include tests to cover the new functionality or bug fix. We use `pytest` for testing, and you can run the tests locally by running `pytest` in the root directory of the repository.

CHAPTER

NINETEEN

CODE OF CONDUCT

Please note that **opensourceleg** has a code of conduct, and we expect all contributors to follow it. You can find the code of conduct in the `CODE_OF_CONDUCT.md` file in the root directory of the repository.

That's it! Thank you for your interest in contributing to **opensourceleg**. We appreciate your help in making this library better.

PYTHON MODULE INDEX

O

INDEX

O

P

R

S

T

U

V

W