Project Leptan Docs Documentation

Release 1.0.0

Pascal Voser

Feb 11, 2022

CONTENTS

Project Leptan is a fully open source, 3D printed hexapod robot, based on the open source Robot Operating System ROS2.

I started the project to learn some more about robotics. These walking robots always intrigued me and I found them particularly fascinating. Since it is much cooler to build the robot yourself than to simply buy it, I started the project. The name Lepta or Leptan is an abbreviation of an ant subfamily, the natural inspiration to the walking pattern of Leptan.

If you have any question or would like to contribute feel free to contact me.

CHAPTER

ONE

CONTENTS

1.1 Overview

1.1.1 Ethymology

Lepta or Leptan is an abbreviation of Leptanillinae, which is a subfamily of ants. Wikipedia The walking pattern of Lepta is inspired by the way these ants move.

Hexapods are 6 legged robots, resembling ants in many ways. In addition Lepta/Leptan was quite free when performing web searches or regarding domains, preventing confusions in the future.

Also Lepta is female, because male ants are called Drones and a Hexapod is not a Drone.

1.1.2 How Does it Walk?

The controller updates the position of each foot regularly in order to achieve a so called gait.

Gait

A gait generally consists of two phases, the swing and the stance phase. During the swing phase, the foot is in the air, swinging forward. In the stance phase, the the food is on the ground, moving backwards and pushing the body forward while doing so.

Fig. 1: A single leg of Lepta performing a gait with swing and stance phase.

In bipedal creatures one foot is in stance phase while the other is in swing phase. Consider the human, who has always one foot on the ground, while the other is swinging forward.

With multi legged creatures there are more options. A Hexapod can for example have 3 legs in stance while the other 3 are in the swing phase. In this configuration the left front leg is in stance while the right front leg is in swing and with alternating rows for the subsequent pairs of legs.

This is called a tripod gait and is commonly used by ants and other 6 legged animals.

Fig. 2: Motion principle of Lepta: While three legs are in stance, the three opposing legs are in swing phase. Note how the roles alternate for each pair of legs.

Inverse Kinematics

Now that we have determined the motion of each foot, the Raspberry Pi has to control each servo in order for every foot to reach its destination. This is achieved with inverse kinematics: you input the coordinates of a foot and it outputs the required angle for each joint.

These angles are than communicated to each servo.

1.1.3 Construction

Hardware

The brain of Lepta is a Raspberry Pi 4. 18 pcs Dynamixel AX-12A digital Servo motors together with a U2D2 compose the 6 legs. The frame and legs are assembled from 3D printed parts. Power delivery is handled by a 3S LiPo battery through DC/DC converters. A voltage and current meter supervises power delivery. An LED ring serves as status and power indicator.

Software

The software for Lepta consists of several components with different responsibilities.

Primarily, there is the main code, that translates control inputs to servo commands and controls motion and control of Lepta. This code is writtin in C/C++ and provides the core functionallities of Lepta.

In addition, there is a supervisor, that observes current, voltages, temperatures and signals to the user using the LED ring. The supervisor is written in python with many libraries for parts were readily available.

Control of the robot is done using a joypad. Currently, I use a XBOX One controller, which can be directly connected to the Raspberry Pi via Bluetooth.

There is also a visualization (RVIZ) which you can run on your PC without the physical robot. A proper simulation (Gazebo) is not yet implemented, but planned for the future.

The full code can be found on Gitlab.

1.2 Hardware

1.2.1 Parts list

Electronic components

The following table lists all electronic components, that I used to set up Lepta. Some of these components are optional, such as the LED ring. I ordered the parts at Mouser due to price and availability, however other suppliers may serve you as well.

Table 1: Electronic components

Qty.	Part name	Description
1	Raspberry Pi 4	
1	LiPo 3S	Battery
1	WS2812B LED ring	24 RGB LEDs
1	40mm PWM Fan	
1	Adafruit INA260	Voltage/Current meter
18	Dynamixel AX-12A	Servo
1	Dynamixel U2D2	Servo Interface
1	ROBOTIS Robot Cable-3P 140mm 10 pcs	Cable
1	ROBOTIS Robot Cable-3P 180mm 10 pcs	Cable
1	ROBOTIS Robot Cable-3P 200mm 10 pcs	Cable
1	ROBOTIS P04-F2 10pcs	
1	ROBOTIS P04-F3 10pcs	
2	ROBOTIS BPF-WA/BU 10pcs	
10	MOLEX 22-03-5035	PCB Connectors

Warning: The AX-12A Bulk packs (6 Pack) ships without the P04-F2/F3 or BPF-WA/BU. If you order the AX-12A in single pack they ship with P04-F2, P04-F3, BPF-WA/BU and screws each. I recommend to order the 6 pack as listed in the table and oder the P04-F2, P04-F3, BPF-WA/BU separately. This leaves no spare parts and additional screws at the end of the build.

Body Parts

The bodywork of Lepta consists of a set of custom 3D-printed parts. Personally, I printed all parts on a Prusa MK3S in PETG with no supports requiered. All part files can be found at Prusa Printers.

The table below lists all parts with the color, that I chose for printing.

Table 2: Custom body parts

Qty.	Description	Color
1	Lower Body	black
1	Lower Body top	black
1	Upper Body	black
6	Femur (2 parts)	orange
6	Tibia	orange
1	Ring Bar	black
1	Ring Bar Spacer	black
1	Ring Cover	white

Note that the Ring Cover should be printed in a light, colorless material to allows light from the LED ring to pass through.

Screws

Lepta is held together my metric M2 and M3 machine screws. These screws distribute over the 6 legs and the body. The table below lists the amount of screws needed for each leg as well as the body and the total quantity necessary for the assembly.

Table 3: Machine screws

	per leg	body	total
	Qty.	Qty.	Qty.
M2 x 6 mm	36	56	272
M2 x 10 mm	–	4	4
M2 x 12 mm / 16 mm	–	4	4
M3 x 10 mm	3	–	18
M2 Nut	16	48	144

Machine screws can usually be sourced at your local hardware store or on the internet.

1.2.2 Assembly

Body

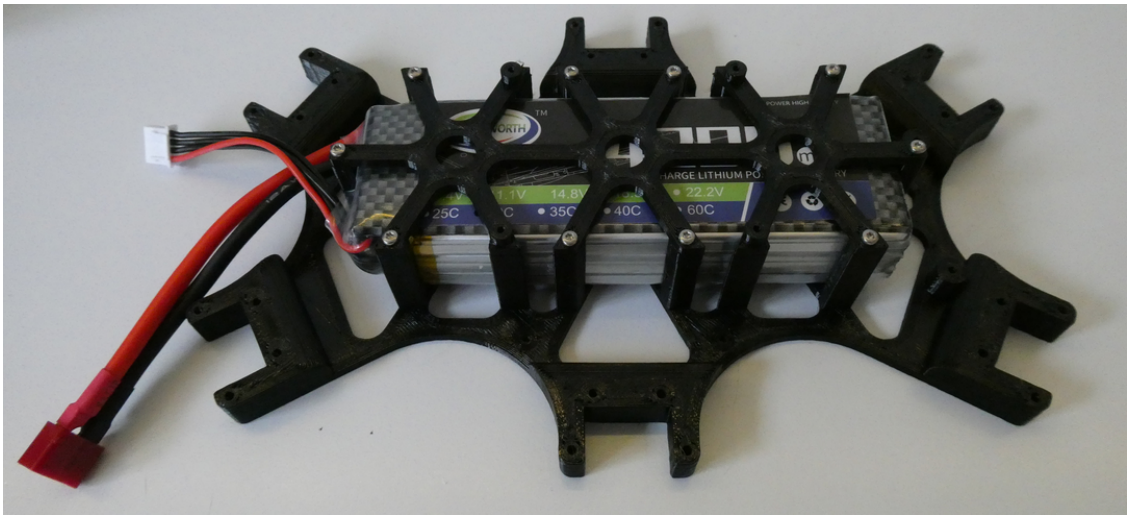
The custom parts are printed in PETG, no supports required. The printable files can be found at Prusa Printers. To assemble Lepta's body, we need the lower body, battery cage cover and the LiPo Battery

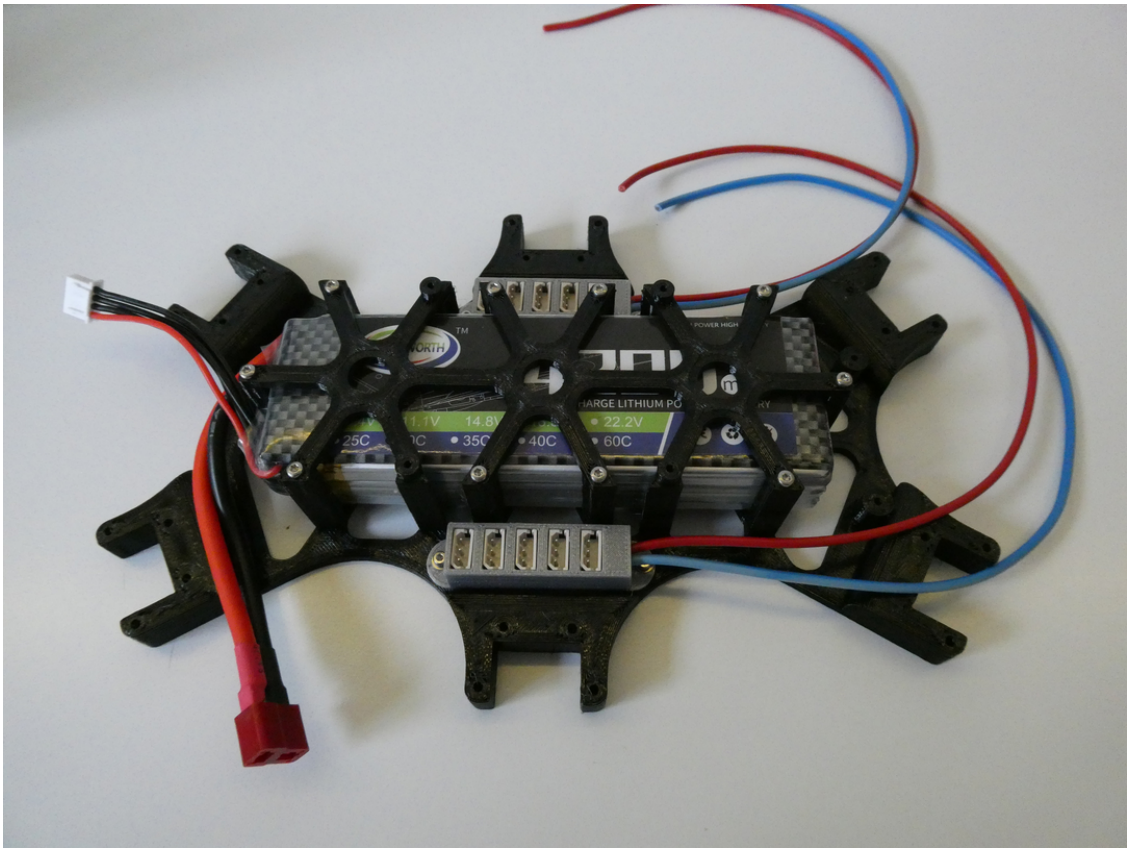
1. First the Battery is placed in the lower body. The body part is symmetric, so orientation does not matter.

2. The battery cage cover is screwed on top of the battery. The 4 holes with a small standoff are intended for the raspberry and therefore left without a screw for now.
3. Then the powered Molex connectors are screwed into place to the left and right of the battery.

Also you can screw on the U2D2 to the standoff at the front of the robot. This concludes the body assembly for the moment.

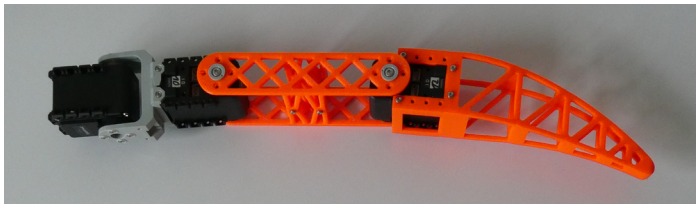
Before the upper part of the body can be completed the legs have to be assembled.



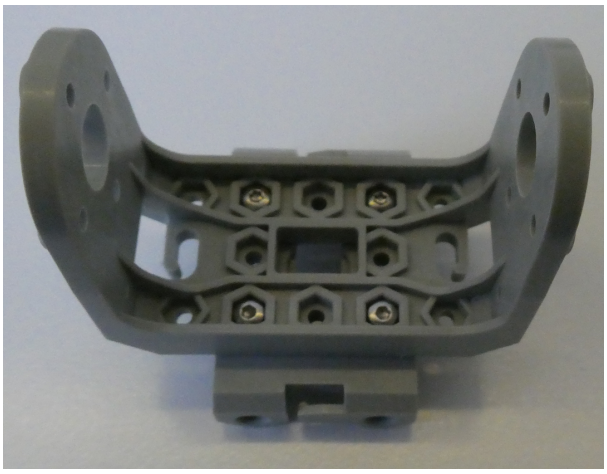


Legs

A complete leg consists of coxa, femur and tibia as shown in the image below (comment in rst-file). Coxa and Femur require some pre-assembly, before they can be incorporated in the leg.

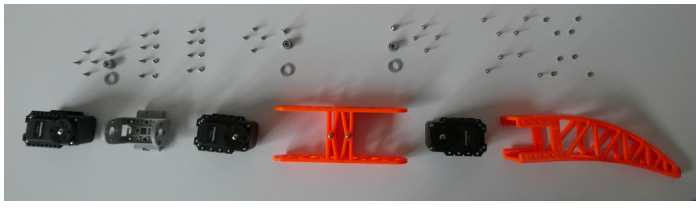


1. The coxa is assembled by connecting the P04-F3 and the P04-F2 like this using 4 pcs M2x6 mm screws with 4 pcs M2 nuts.



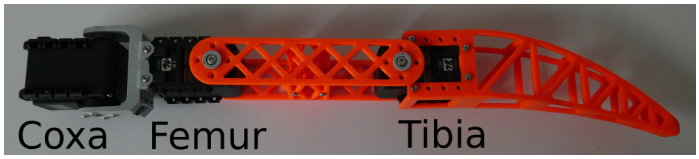
2. The two parts of the femur are connected with 4 pcs M2x6 mm screws directly into the plastic without a nut.
3. Now the servos can be installed between tibia and femur, femur and coxa, and to the other end of the coxa. Wherever a servo is connected to the custom parts be sure to use the BPF-WA/BU on the opposite side than the Servo Horn.

   This exploded view of the leg shows you the screws and parts, that you need for every joint. Be sure to keep the orientation of the servos the same, otherwise you would have to change the control. (All screws in the image are M2x6 mm)



Servo Numbering

To keep track of the individual servos, I name them identical to the leg segment, that they control. Thus the up most servo in the leg, that controls the coxa, is also referred to as coxa (comment in .rst).



Each Servo requires an unique ID in order to be addressed by the system. This ID can be assigned using the Dynamixel Wizard.

Connect each Servo individually to your PC using the serial interface, and assign these IDs:

Table 4: Servo IDs

Leg ID	Coxa	Femur	Tibia
Right Rear	8	10	12
Right Middle	14	16	18
Right Front	2	4	6
Left Rear	9	11	13
Left Middle	15	17	19
Left Front	3	5	7

Connect the servo motors on each leg in series (?) together, from femur to tibia to coxa.

Body revisited

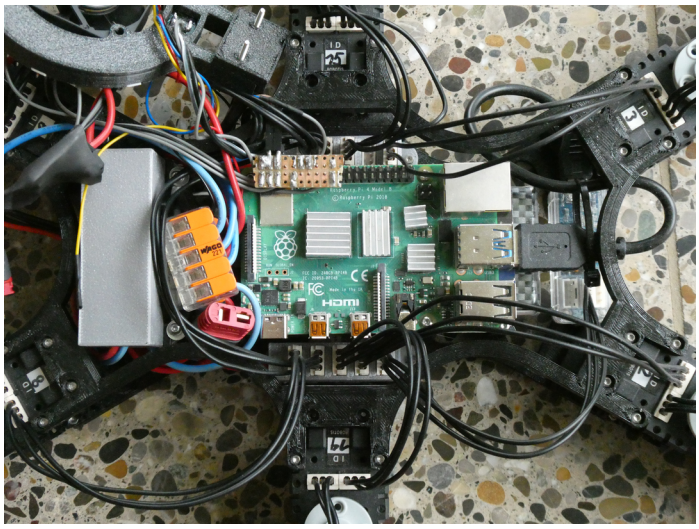
1. After all 6 legs have been assembled they are attached to the lower body, by using 4 pcs M2 x 6 mm screws with Nuts.
2. Connect the 6 Coxa motors with the Molex connectors (3 on each side).
3. Now screw the raspberry pi into place.

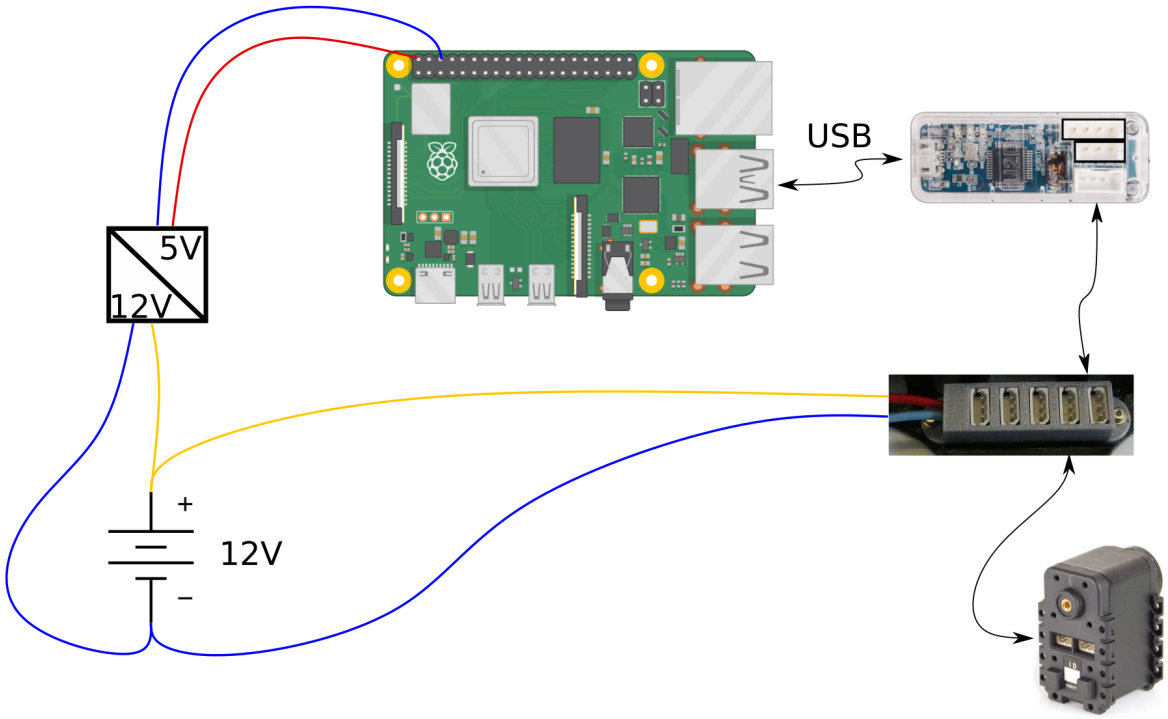
Before we mount the body top, it is the time to finish the cabling.

Cabling

This image shows the body with the Raspberry Pi installed and all major cable runs connected.

In a first step, we supply power to the Raspberry Pi and the legs and connect the servo controller. The connections are done as shown in the schematic below.



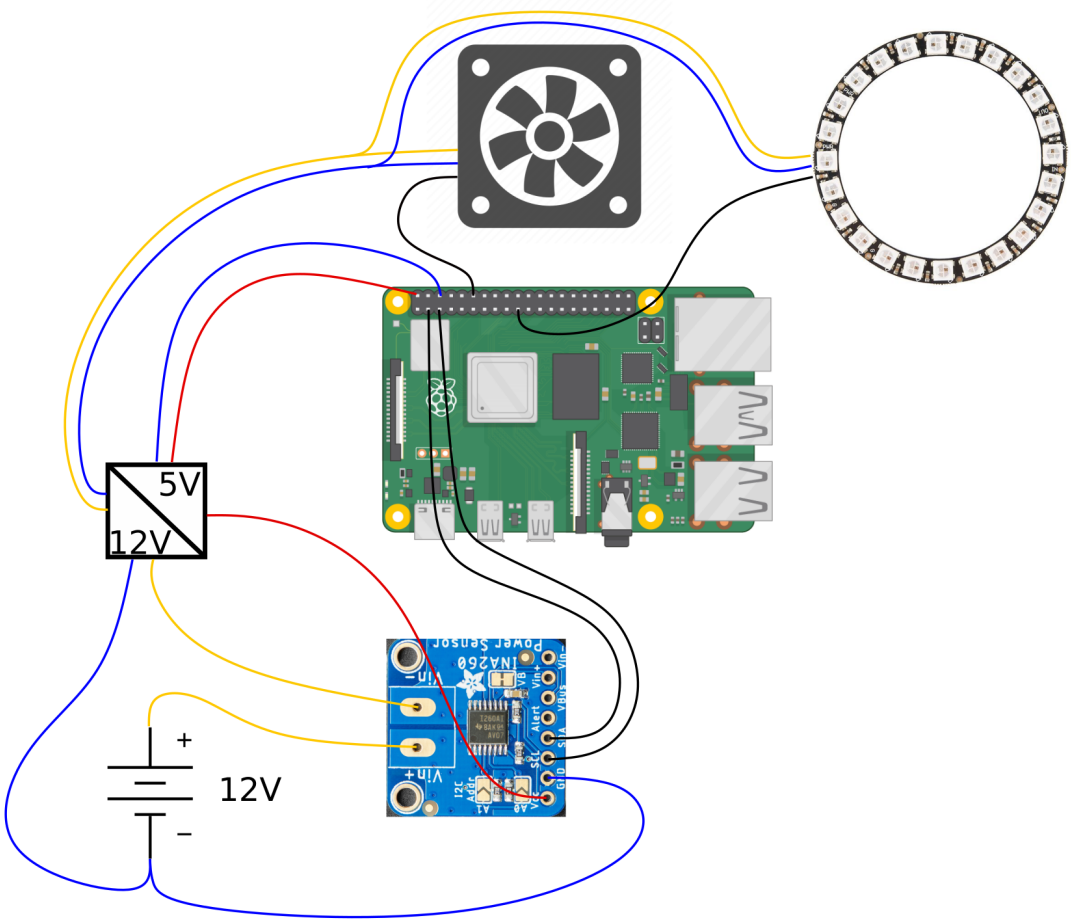


1. The battery is connected to the DC/DC converter. The DC/DC converter should supply enough current for the raspberry pi, it is recommended to provide 3A.
2. The Raspberry Pi receives power from the 5 V outputs of the DC/DC converter to its power pins 4 (5 V) and 6 (GND).
3. To supply power to the legs, we connect the powered Molex connectors to the 12 V supply.
4. Finally, connect the servo controller to the Raspberry Pi and the Molex rails.

Supervisor Cabling

These parts are not mandatory for the robot, but can be a big help. I would recommend to use at least the INA260 or a similar device to supervise the battery and power delivery.

The schematic below shows the power and data connections for the supervisor



1. Connect power as shown above.
2. For the data runs to the GPIO pinout header of the Raspberry Pi, follow the pins described in the table below

Table 5: Data connections

Part	GPIO pin	Label
Fan	GPIO 18	PWM0
LED Ring	GPIO 10	SPI MOSI
INA260 Voltage monitor	GPIO 2	I2C Data
	GPIO 3	I2C Clock

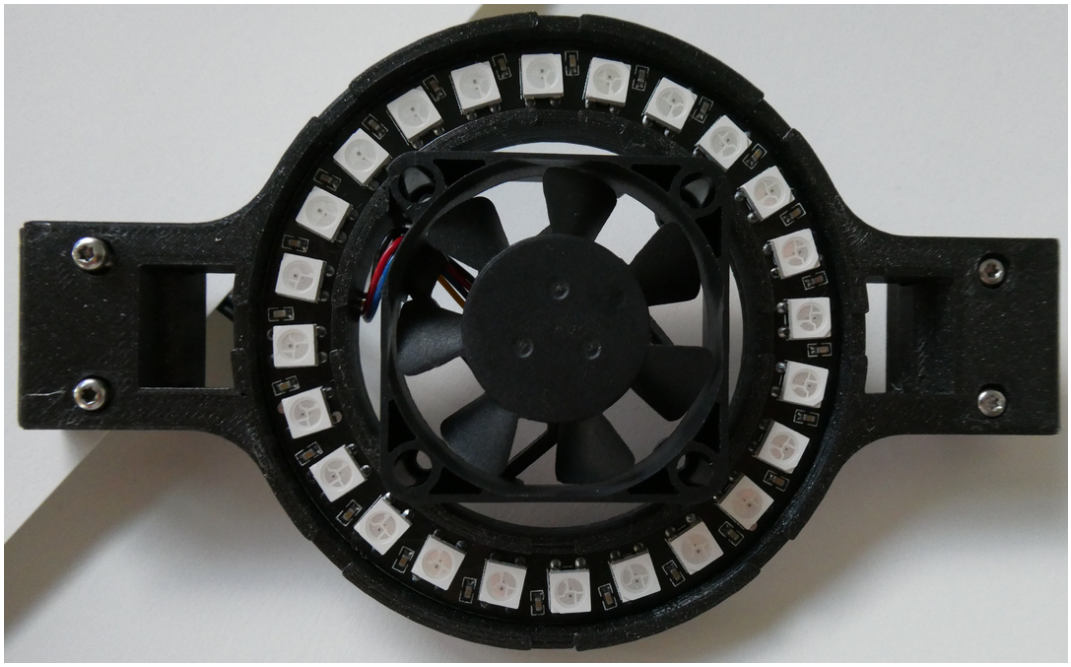
Upper Body

After the cabling is done the upper part of the body can be screwed on top, including either the straight bridge or the bridge including the Fan and LED ring, depending on your choice of Hardware.

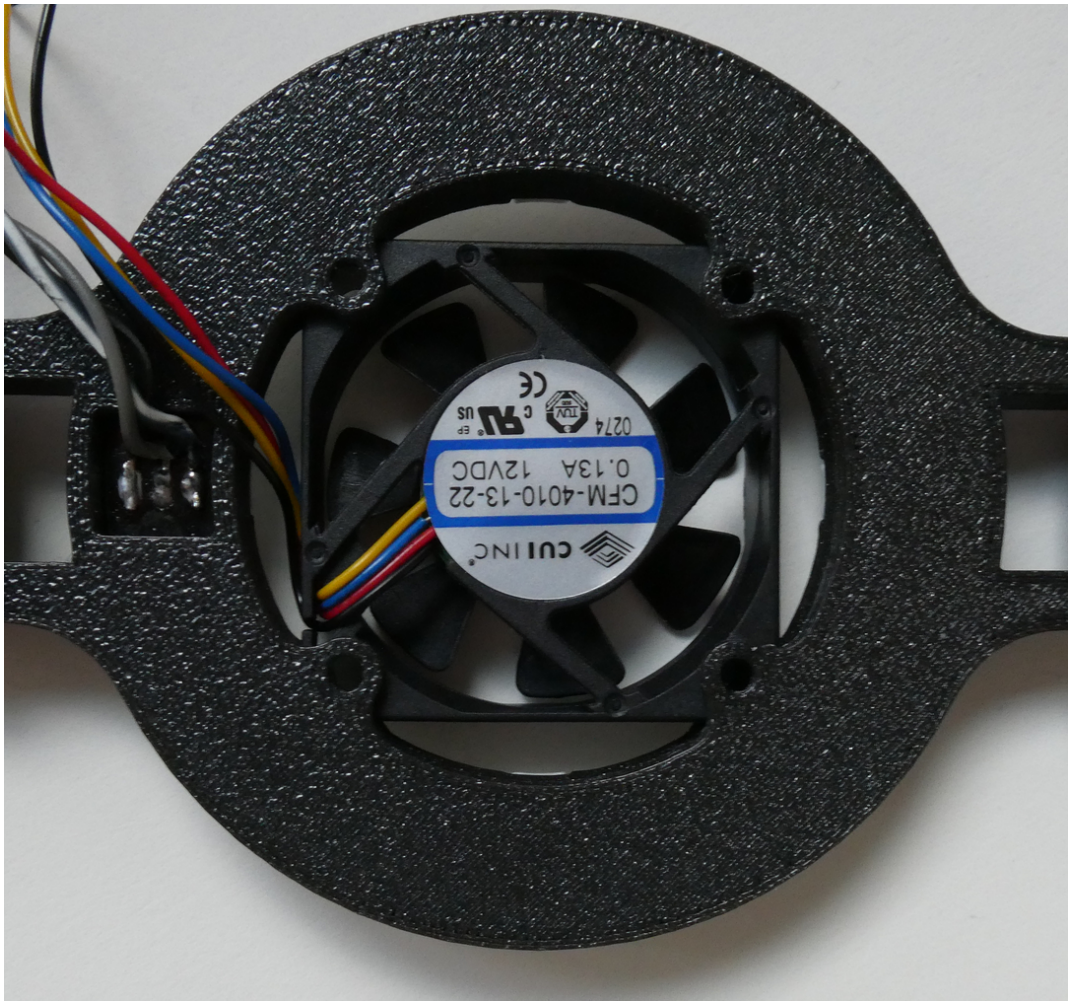
The bridge makes the body more rigid, while allowing easy access to the electronics.

The straight bridge can simply be screwed on top of the body, while the bridge containing the LED ring and fan requires some assembly.

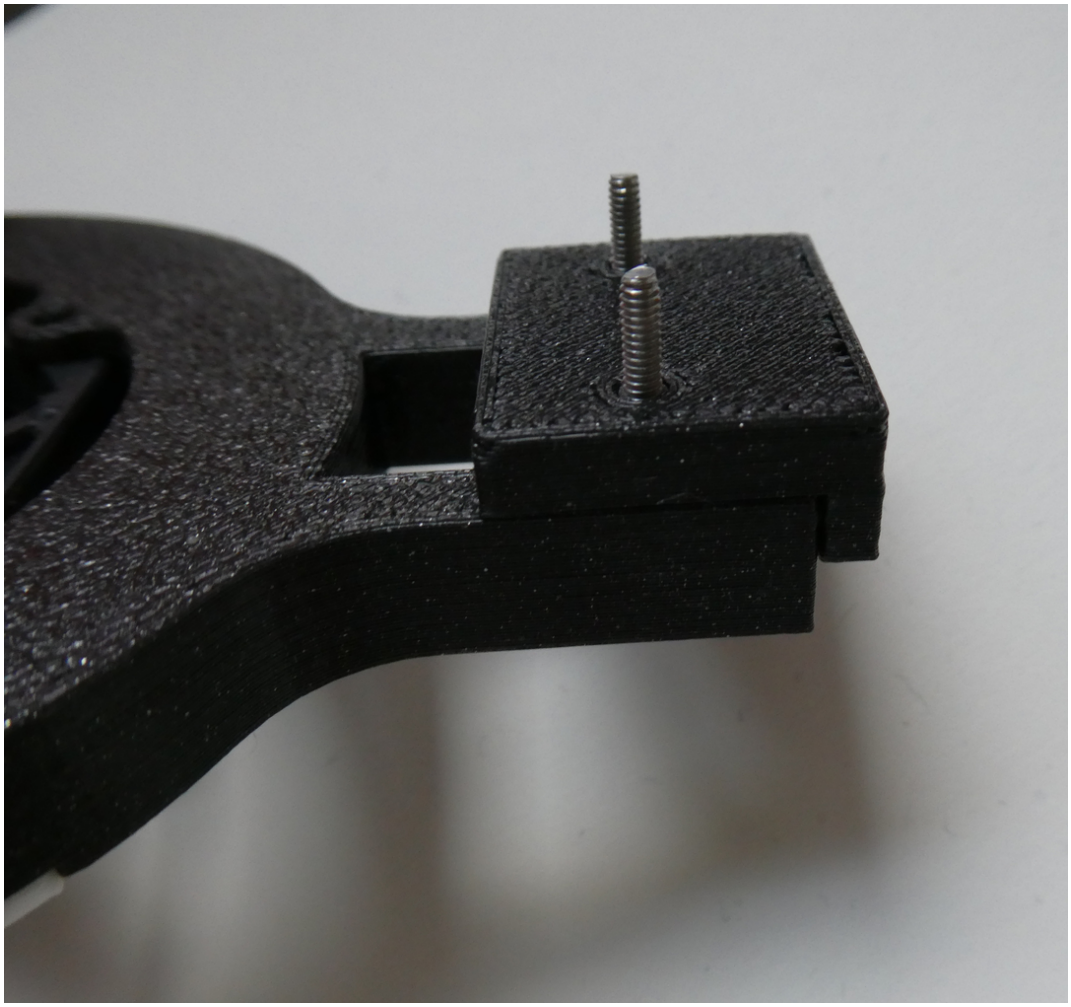
1. Push the LED ring and the Fan inside the ring:



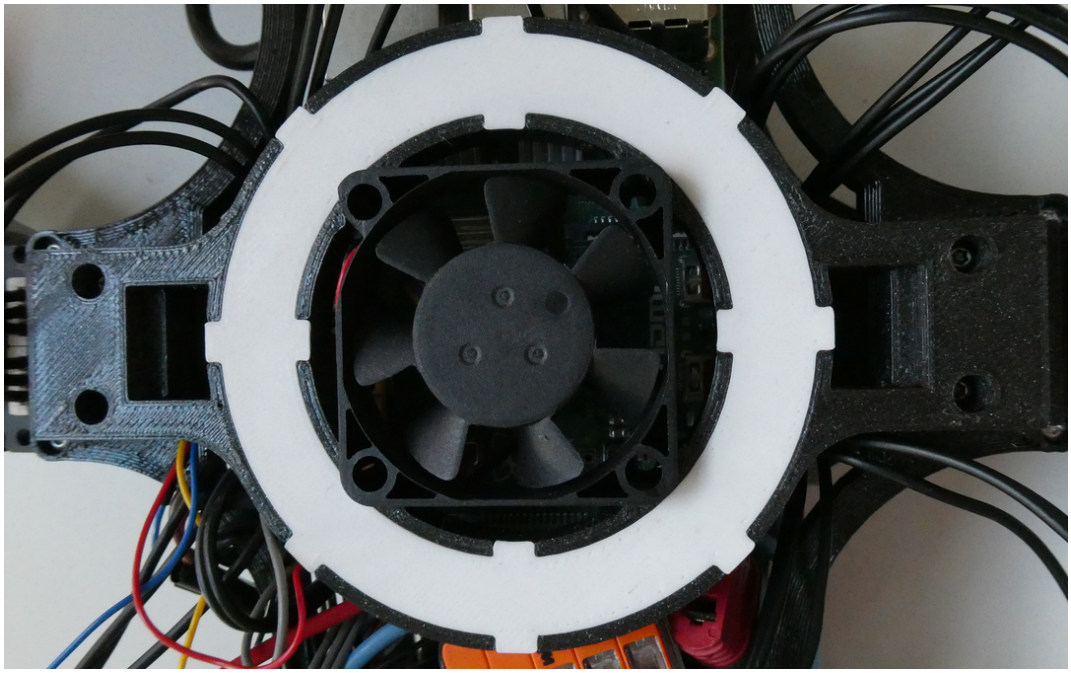
2. Solder the wires to the LED ring through the hole in the back:



3. Snap the Ring Spacer to the LED ring. The spacer is required because otherwise the Ring would be obstructed by the USB ports of the Raspberry Pi:



4. Push the Ring cover on top of the Ring, no glue required as friction keeps everything in place.
5. Don't forget to connect the LED ring and the Fan to the Raspberry Pi. Screw the RingBar on top of the robot. The result should look something like this:



And with this, the assembly of Lepta is completed.

1.3 Installation

Leptan can be installed in two ways. (Check comment in code.)

While the first way is a mere visualization of Lepta, it provides insights into Lepta's motion and technology without the requirement of hardware being available. This may help you to decide whether you wish to build Lepta on your own.

The second method requires the assembled hardware of Lepta. I recommend to move through the hardware and assembly completely, before starting the installations.

1.3.1 Visualization Setup

The visualization is intended to test the robot behavior without the actual robot.

For using the PC visualization a joypad is required. Right now only an XBOX One joypad has been programmed. Other input device will be implemented as requests arise and hardware for testing is available.

Install ROS

Install the full Desktop version of ROS2 Foxy.

Install Colcon, which is used to build the project and Xacro, which is used for model building.

```
sudo apt install python3-colcon-common-extensions
sudo apt install ros-foxy-xacro
```

Note: You may need to install further packages, if they are not yet present on your system. Carefully read the console output to install missing packages.

Download and build

Clone the Repo:

```
git clone --recurse-submodules https://gitlab.com/Combinatrix/project-leptan.git
```

Navigate into the project directory and build the project:

```
cd project-leptan
colcon build
```

Or build the project and run the tests:

```
colcon build && colcon test && colcon test-result --verbose
```

Usage

Source ROS2 as described in the installation package. Then source the generated files.

```
source /opt/ros/foxy/setup.bash
source install/setup.bash
```

Connect the joypad.

Launch the visualization:

```
ros2 launch launch_files hexapod_launch.py
```

The launch file checks if it is run on a Raspberry or not. According to this either the robot or the visualization is started.

Control

As of time of writing the controls are (Button guide):

- **Menu** Turn on, go to neutral position
- **A** Stand up
- **B** Sit down
- **X** Switch between walking mode and body rotation mode
- **Joy Stick Left** Walk
- **Joy Stick Right** Turn/Rotate

1.3.2 Raspberry Pi Setup

The Raspberry Pi 4 in the center of the assembly is the literal brain of Lepta and runs all the software. This in turn requires the following operations to be executed on the Raspberry Pi itself.

Setup Leptas backbone

Install Ubuntu Server 20.04 LTS on Raspberry

1. Download the image ubuntu.com
2. Installation instruction

Install ROS2 Foxy

These instructions follow closely the manuals provided by roboticsbackend.com or on the ROS2 homepage. Refer to these sources in case of trouble.

Setup locale.

```
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
```

Setup sources.

```
sudo apt update && sudo apt install curl gnupg2 lsb-release
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key  -o /usr/
↪share/keyrings/ros-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-
↪keyring.gpg] http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" | sudo tee /
↪etc/apt/sources.list.d/ros2.list > /dev/null
```

Install ROS2.

```
sudo apt update
sudo apt install ros-foxy-ros-base
```

To automatically source the ROS files and the Project Leptan files, add these lines to the end of the .bashrc file.

```
source /opt/ros/foxy/setup.bash
source /home/ubuntu/project-leptan/Software/install/setup.bash
```

Finally, we need to install Colcon and the CPP compiler.

```
sudo apt install python3-colcon-common-extensions
sudo apt install g++
```

Clone the Git repo

```
git clone --recurse-submodules https://gitlab.com/Combinatrix/project-leptan.git
```

For the sake of this installation, we will assume, that you clone project leptan into the home directory of the ubuntu installation, i.e. the project directory is `/home/ubuntu/project-leptan`.

Install Supervisor Requirements

The supervisor is an independent program which observes and indicates the state of Lepta. It monitors the battery voltage and indicates its status on the LED ring, and governs the bluetooth connections. It also starts the hexapod Controller as soon as a joypad is connected.

The supervisor is written in Python 3. Install pip, if not yet present, to install further packages for the supervisor.

```
sudo apt install python3-pip
```

WS2812 LEDs

The LED ring uses WS2812 LEDs. Install the neopixel SPI driver for the LED ring.

```
sudo pip3 install adafruit-circuitpython-neopixel-spi
```

SPI

SPI is required to drive the LED ring. Detailed instructions can be found at SPI and Neopoixel.

For SPI-bus access create the file `/etc/udev/rules.d/50-spi.rules` with the following contents.

```
SUBSYSTEM=="spidev", GROUP="spiuser", MODE="0660"
```

To make your changes take effect, reboot the Raspberry Pi or reload the udev rules.

Next, copy and paste or type these lines into the terminal as the user you want to give access to the SPI bus.

```
sudo groupadd spiuser
sudo adduser "$USER" spiuser
```

INA260

The INA260 is a power meter chip which can measure not only voltage but also current and the according power consumption. Communication is using I2C.

install the git package directly with this command:

```
pip3 install git+https://github.com/jveitchmichaelis/ina260.git
```

For I2C-bus access create the file `/etc/udev/rules.d/51-i2c.rules` with the following contents:

```
SUBSYSTEM=="i2c-dev", GROUP="i2cuser", MODE="0660"
```

Again, reboot the Raspberry Pi or reload the udev rules, to make these changes take effect.

Also grant your user access to the I2C bus.

```
sudo groupadd i2cuser
sudo adduser "$USER" i2cuser
```

Temperature

The temperature of the raspberry pi chip is used to regulate the speed of the fan for cooling. This is not strictly required, but a raspberry pi 4 runs quite hot without any additional cooling.

```
pip3 install gpiozero
```

For GPIO access without root create the file `/etc/udev/rules.d/52-gpio.rules` with the following contents

```
SUBSYSTEM=="bcm2835-gpiomem", GROUP="gpio", MODE="0660"
SUBSYSTEM=="gpio", GROUP="gpio", MODE="0660"
SUBSYSTEM=="gpio*", PROGRAM="/bin/sh -c '\
        chown -R root:gpio /sys/class/gpio && chmod -R 770 /sys/class/gpio;\
        chown -R root:gpio /sys/devices/virtual/gpio && chmod -R 770 /sys/devices/
↪virtual/gpio;\
        chown -R root:gpio /sys$devpath && chmod -R 770 /sys$devpath\'"
```

Again, grant access to the GPIO interface to your user.

```
sudo groupadd gpio
sudo adduser "$USER" gpio
```

Enable shutdown

```
sudo chmod u+s /sbin/shutdown
```

Enable Linux event interface in Python

```
pip3 install evdev
```

Setup Joypad

Instruction can be found pimylifeup.com.

Install required Software for XBOX controller.

```
sudo apt install xboxdrv
```

This command disables the Enhanced Re-Transmission Mode (ERTM) of the Bluetooth module. With it enabled, the Xbox Controller won't pair correctly.

```
echo 'options bluetooth disable_ertm=Y' | sudo tee -a /etc/modprobe.d/bluetooth.conf
```

Install Bluetooth

```
sudo apt install pi-bluetooth
```

Follow a guide like this one in order to pair & connect the controller.

Add the user to the input group in order to access the joypad:

```
sudo adduser "$USER" input
```

Further Requirements

```
sudo apt install ros-foxy-joy
```

Run the Supervisor

The following command starts the supervisor:

```
ros2 run supervisor supervisor_node
```

Start at boot

Only thing left to do is to auto-start the supervisor at boot using Crontab.

Open the crontab file:

```
crontab -e
```

Add this line:

```
@reboot bash -ic "ros2 run supervisor supervisor_node"
```

1.4 Code

This section documents the interesting, critical or complicated parts of project leptans code base. Thereby this section is structured in agreement with the code's file structure. Thus looking for the documentation to a specific piece of code just means looking up at the same location as in the directory tree.

1.4.1 Dynamixel SDK

The Dynamixel SDK is an external repo provided by Dynamixel.

> The ROBOTIS Dynamixel SDK is a software development kit that provides Dynamixel control functions using packet communication. The API is designed for Dynamixel actuators and Dynamixel-based platforms. For more information on Dynamixel SDK, please refer to the e-manual below.
>
> —ROBOTIS Dynamixel SDK

1.4.2 Hexapod Control

Main Directory, contains the movements and everything important.

Overview of the different files:

Hexapod Controller

Main file. Starts the node, adds the subscriber & publishers, initializes everything.

Details

Main Loop

This is the main hexapod controller, managing the timers and starting all required functions.

The main code runs the motion loop in a timer:

hexapod_controller.cpp:

```
99  motion_control.update_vel(cmd_vel_);
100 motion_control.motion_loop();
101 motion_control.publish_joint_state(this->now());  // For RVIZ
```

The timer is started here:

```
99  timer_ = this->create_wall_timer(10ms, std::bind(&HexapodController::timer_callback,␣
    ↪this));
```

Create the main ROS node:

```
127 auto hexa_node = std::make_shared<HexapodController>();
```

Subscribe to the joy pad input and start the node:

```
175 auto subscription_ = hexa_node->create_subscription<sensor_msgs::msg::Joy>(
176   "/joy", 1, joy_callback);
177
178 spin(hexa_node);
```

Gait

This module generates the feet position sequence used for walking for each leg individually.

Details

Gait

The main function of the Gait is to cycle through the walking motion. This is done using a FSM with states dependent on the position of the foot.

If we look at one leg, animal or human gait consist of 2 phases:

1. **Swing Phase:** The foot is in the air, "swings" forward.

2. **Stance Phase:** The foot is on the ground, "pushing" back. Here the foot supports the weight while moving the body forward.

Stance Phase

This is where the leg moves the body, and it is actually the less complicated of the motions. It directly corresponds to the movement of the body and is therefore directly linked to the input.

Swing Phase

The swing phase is further divided into sub-phases, here I have chosen sub-phases which seemed to make to most sense to me:

Stance_to_Swing

Transition state where foot moves of the ground while keeping in line with the movement of the body. In case of soft or uneven ground this means the leg can lift itself before moving in the opposite direction.

Swing_Lift

Lifting the foot until it reaches a certain height.

Swing_Plateau

Moves the foot to an estimated suitable starting position according to the current movement direction.

Swing_Lower

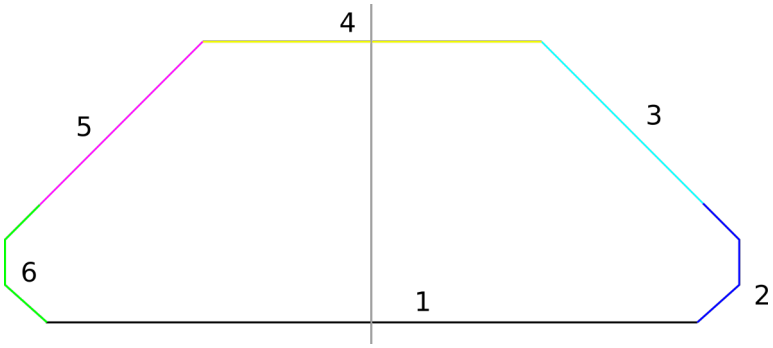
Lowers the foot in preparation of the next stance phase

Swing_To_Stance

Opposite motion of the stance_to_swing phase.

Complete Motion

The planed motion looks like this:



1. Stance
2. Stance_to_Swing
3. Swing_Lift
4. Swing_Plateau
5. Swing_Lower
6. Swing_To_Stance

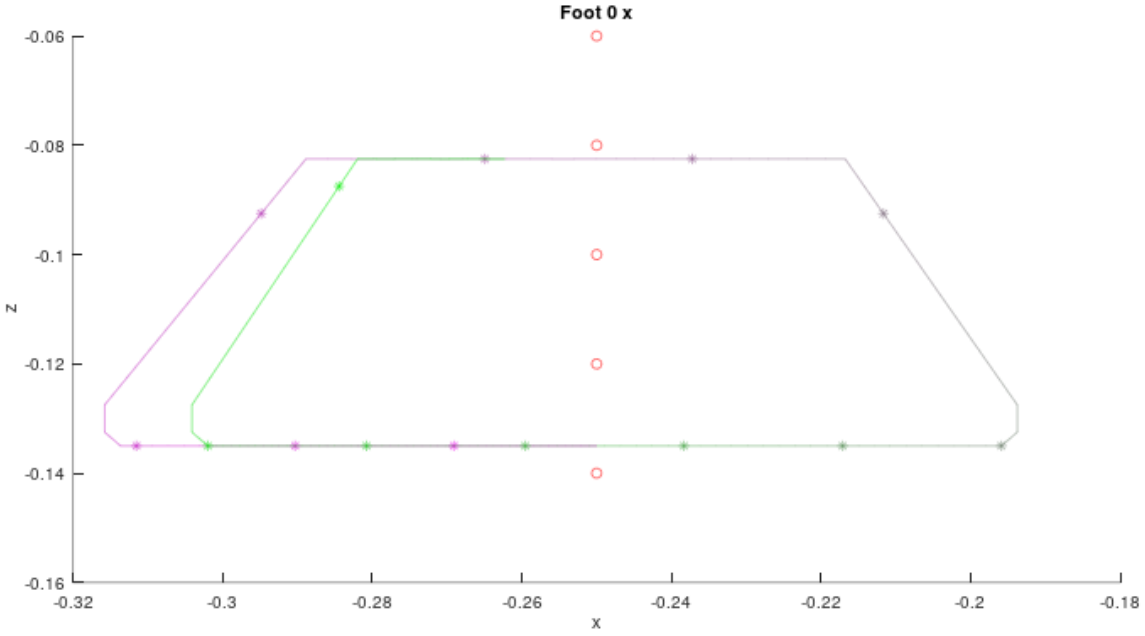
The gray line depicts the neutral position for the Coxa joint.

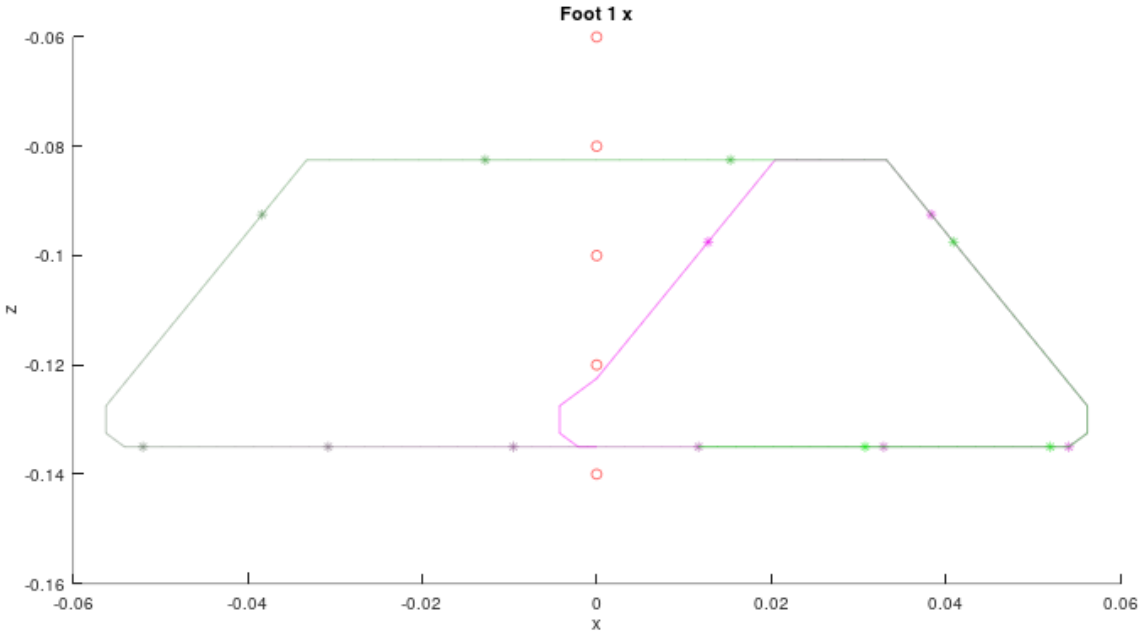
This motion is the desired steady state when the robot is walking without changing direction or speed.

Start Walking

Let us assume the feet are i a neutral position, this means all feet touch the ground and the Coxa joint is in a neutral position, as indicated by the gray line.

This means some legs have to move in Stance state while others have to transition to the swing state. The goal is to reach a steady state as soon as possible.





Foot 0 starts in stance, while foot 1 transitions directly to swing. With the current implementation a steady state is reached in 2 cycles.

Stop Walking

When the robot stops walking the legs should move into a neutral position, in order to be ready for further movements or sitting down.

This means that the legs cannot move anymore during stance phase, but have to reach the neutral point in the swing phase.

Walking Patterns

For stability reasons at least 3 legs have to touch the ground at any time. Otherwise there is no guarantee that the hexapod is stable.

This restriction leaves several possible gaits. Ants and other insects usually walk with a tripod gait:

Tripod Gait

Tripod gait means that 3 legs are in the stance phase while the other 3 are in swing state.

Code

One step cycle is the movement from a neutral point through stance & swing back to the neutral point. The step cycle is divided into a number of sub-steps. I have chosen 50 sub-steps for the stance as well as the swing phase, which results in at most 100 sub-steps per cycle.

Gait distinguishes between straight walking (angular velocity z = 0) and walking in circles.

Walking Straight

All legs on the ground have the same speed and direction, which is directly proportional to the input.

gait.cpp:

```
148 if (cmd_vel.angular.z == 0) {
149   dx[leg_index] = -vx;
150   dy[leg_index] = -vy;
151 }
```

```
158 dz[leg_index] = 0;  // feet stay on the ground
```

dx and **dy** are the relative distances each leg moves. This means the legs on the ground move directly opposite to the robot's direction.

The swing phase is divided in states according to the description above:

Stance_to_Swing/Swing_To_Stance

Both states are pretty much the same.

gait.cpp:

```
188 if (cmd_vel.angular.z == 0) {
189   dx[leg_index] = -transition_factor[0][transition_index] * vx;
190   dy[leg_index] = -transition_factor[0][transition_index] * vy;
191 }
```

```
198 dz[leg_index] = transition_factor[1][transition_index] * height_per_step;
```

Where **transition_factor** is a hard coded transition sequence, where the foot is lifted/lowered slowly and moving another two sub-step in sync with the stance feet while moving off the ground:

gait.hpp:

```
73 std::vector<std::vector<double>> transition_factor =
74 {
75   {1, 1, 0},  // x/y movement
76   {0, 0.5, 1}   // z movement
77 };
```

Swing_Lift/Swing_Plateau/Swing_Lower

During this part of the swing the foot should reach a good position where the next stance phase can start. In a steady state situation the end position would be opposite of the start position. But the input can change any time, which complicates things.

In order to reach a steady state movement in any case, it has been decided that the swing phase has to pass through the neutral point. This prevents too large deviations from a steady state.

In order to achieve the pass through neutral point the swing phase has been further divided into *move to neutral* and *move from neutral*.

This is achieved using the **reached_neutral** flag:

gait.cpp:

```
312 // dx/dy movement to/from neutral, only for these 3 states.
313 // 1. Reach neutral point
314 // 1a if neutral point reached before steps/2 goto steps/2 to shorten the walking period
315 // 2. move from neutral to good starting point for next step
316 if (gait_state == Swing_lift ||
317   gait_state == Swing_plateau ||
318   gait_state == Swing_lower)
319 {
320   if (!reached_neutral) {
321     bool close_neutral = true;
322     // check if we are close to the neutral point
323     for (int leg_index : leg_groups[leg_group_index]) {
324       if (std::abs(foot_pos_x[leg_index] - neutral_foot_pos_x[leg_index]) > stability_
    ↪radius ||
325         std::abs(foot_pos_y[leg_index] - neutral_foot_pos_y[leg_index]) > stability_
    ↪radius)
```

(continues on next page)

(continued from previous page)

```
      {
        close_neutral = false;
      }
    }
```

Checks if all feet are close to the neutral point. If this is the case the sub-step-index is moved ahead if necessary in order to shorten the step cycle. This happens in case of the start walking, where the initial position is close to neutral. If the step cycle would make use of the full number of steps in this case, the neutral position could not be reached fast enough and the steady state would be out of reach.

```
    // Skip a few steps if we are already close to the neutral point
    if (close_neutral) {
      // std::cerr << "Reached neutral point" << std::endl;
      reached_neutral = true;
      if (sub_step_index < SUB_STEPS / 2) {
        sub_step_index = SUB_STEPS / 2;
      }
    }
  }
```

The goal is to achieve the neutral point fast, but at a similar time for all legs. Because it does not help if some legs are faster than others. The motion is smoother if the legs reach the neutral point around the same time.

Therefore, we estimate the max. nr of steps the slowest foot requires to reach the target, on either the x or y axis:

gait.cpp:

```
  if (!reached_neutral) {
    // move to neutral point, try to do so before reaching mid cycle
    step_nr = 0;
    // find max nr of steps required to reach neutral point
    for (int leg_index : leg_groups[leg_group_index]) {
      step_nr = std::max(
        step_nr,
        std::abs(
          (foot_pos_x[leg_index] - neutral_foot_pos_x[leg_index]) /
          (distance_per_step * scaling_factor))
      );
      step_nr = std::max(
        step_nr,
        std::abs(
          (foot_pos_y[leg_index] - neutral_foot_pos_y[leg_index]) /
          (distance_per_step * scaling_factor))
      );
    }
```

This is reevaluated on each new sub-step.

Assign the dx/dy to move to the neutral point, but not faster than *(distance_per_step * scaling_factor)*.

gait.cpp:

```
  for (int leg_index : leg_groups[leg_group_index]) {
    // move foot the fastet way to neutral position
    dx[leg_index] = -sgn((foot_pos_x[leg_index] - neutral_foot_pos_x[leg_index])) *
```

(continues on next page)

(continued from previous page)

```
      std::min(
      std::abs((foot_pos_x[leg_index] - neutral_foot_pos_x[leg_index])) / step_nr,
      (distance_per_step * scaling_factor)
      );
    dy[leg_index] = -sgn((foot_pos_y[leg_index] - neutral_foot_pos_y[leg_index])) *
      std::min(
      std::abs((foot_pos_y[leg_index] - neutral_foot_pos_y[leg_index])) / step_nr,
      (distance_per_step * scaling_factor)
      );
  }
```

This concludes the move_to_neutral part of the swing state.

The move from neutral is considerably simpler, as the movement is simply in the opposite direction of the feet in stance phase.

gait.cpp:

```
  } else {
    // move from neutral point, as many steps as there are still left in the cycle
    double scaling_factor = (SUB_STEPS + transition_steps) / (SUB_STEPS - 2 * transition_
↪steps);
    for (int leg_index : leg_groups[leg_group_index]) {
      if (cmd_vel.angular.z == 0) {
        dx[leg_index] = vx * scaling_factor;
        dy[leg_index] = vy * scaling_factor;
      }
```

The dz movement in swing is extremely simple: **Swing_Lift:** dz = height_per_step **Swing_Plateau:** dz = 0 **Swing_Lower:** dz = -height_per_step

Switch Between Swing and Stance

After a finished swing/stance phase the foot switches to stance/swing phase. All legs are grouped, and an index of each group dictates if they are in swing or stance phase, and when to switch:

gait.cpp:

```
  if (new_gait_interval) {
    sub_step_index = 0;
    leg_group_index = (leg_group_index + 1) % 2;
    leg_group_index1 = (leg_group_index + 1) % 2;

    // std::cerr << "[----------] Switch leg group " << std::endl;
  }
```

Assign Feet Positions

After the relative movements of all feet are calculated the values are assigned:

gait.cpp:

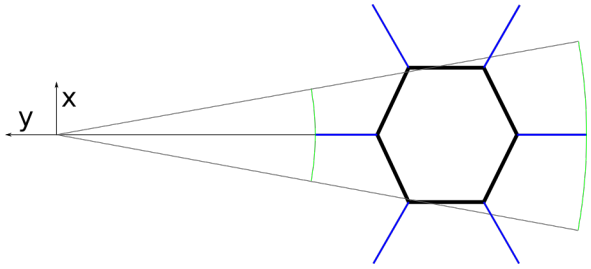
```
396 // Calculate the position for each foot
397 for (int leg_index = 0; leg_index < NUMBER_OF_LEGS; leg_index++) {
398   next_foot_pos_x[leg_index] = foot_pos_x[leg_index] + dx[leg_index];
399   next_foot_pos_y[leg_index] = foot_pos_y[leg_index] + dy[leg_index];
400   next_foot_pos_z[leg_index] = foot_pos_z[leg_index] + dz[leg_index];
401   // TODO(voserp): Test if z goes below ground
402
403   feet->foot[leg_index].position.x = next_foot_pos_x[leg_index];
404   feet->foot[leg_index].position.y = next_foot_pos_y[leg_index];
405   // This way the floor is at height 0
406   feet->foot[leg_index].position.z = next_foot_pos_z[leg_index] - walking_height;
407 }
```

Walking Curves

Walking in curved lines is considerably more complicated than walking in a straight line.

While performing a turn the robot should have a center point around which it turns. This center point is directly calculated from the input.

If we consider the robot walking around a given center point each leg has to move around this point. This means each leg has a different radius because each leg has a different distance to the center point:



If we consider the the center of the robot moves $d\phi$ around the center point, each leg has to travel a different distance since the radii are different for each leg, as illustrated above.

Center Point

One of the first steps required is to calculate the center point *(x0, y0)* according to the input:

gait.cpp:

```
97  if (cmd_vel.angular.z != 0) {
98    if (vx == 0 && vy == 0) {
99      x0 = 0;
100     y0 = 0;
101   } else {
102     // 2.5/exp(vz) linearizes the input and scales it to a radius from approx. 0.22m to␣
↪1.8m
```

(continues on next page)

(continued from previous page)

```
    x0 = -vy / sqrt(pow(vy, 2) + pow(vx, 2)) *  std::cout << "title('Foot 1 y'); xlabel('y'), ylabel('z')" << std::endl;

int Plot_Start = 00;
int Plot_Stop = 400;
int Plot_Stop = 150;
int Plot_Steps = Plot_Stop - Plot_Start;

Gait gait;
      sgn(cmd_vel.angular.z) * (2.5 / exp(abs(cmd_vel.angular.z)) - 0.7);
    y0 = vx / sqrt(pow(vy, 2) + pow(vx, 2)) *
      sgn(cmd_vel.angular.z) * (2.5 / exp(abs(cmd_vel.angular.z)) - 0.7);
    // std::cerr << "x0/y0 = " << x0 << "/" << y0 << std::endl;
  }
```

If *vx* and *vy* are 0, the robot rotates without moving. therefore *x0* and *y0* are 0. Otherwise the center is calculated like this:

$$x_0 = \frac{-v_y}{\sqrt{v_x^2 + v_y^2}} * sign(v_z) * factor$$

Where the factor is used for linearizing and scaling of the center point position:

$$factor = \frac{2.5}{\exp |v_z|} - 0.7$$

Using the center point the distance of each foot is calculated:

gait.cpp:

```
for (int leg_index = 0; leg_index < NUMBER_OF_LEGS; leg_index++) {
  rho[leg_index] = sqrt(
    pow(foot_pos_x[leg_index] - x0, 2) +
    pow(foot_pos_y[leg_index] - y0, 2)
  );
```

Also the angle of each foot corresponding to the current position:

```
if ((foot_pos_x[leg_index] - x0) < 0) {
  phi[leg_index] = M_PI - asin((foot_pos_y[leg_index] - y0) / rho[leg_index]);
} else {
  phi[leg_index] = asin((foot_pos_y[leg_index] - y0) / rho[leg_index]);
}
```

Then we need to calculate how much the robot moves, in terms of angle difference dphi:

```
    if (vx == 0 && vy == 0) {
      dphi = cmd_vel.angular.z * dphi_max_standing;
    } else {
      dphi = sgn(cmd_vel.angular.z) * atan(sqrt(pow(vx, 2) + pow(vy, 2)) / max_rho);
      // dphi has to be limited so that the fastest foot is not faster than the allowed limit
    }
```

If we have the radius of each foot and dphi we can calculate the movement of the feet on the ground:

```
for (int leg_index : leg_groups[leg_group_index1]) {
  if (cmd_vel.angular.z == 0) {
    dx[leg_index] = -vx;
    dy[leg_index] = -vy;
  } else {
    dx[leg_index] = -foot_pos_x[leg_index] +
      rho[leg_index] * cos(phi[leg_index] - dphi) + x0;
    dy[leg_index] = -foot_pos_y[leg_index] +
      rho[leg_index] * sin(phi[leg_index] - dphi) + y0;
  }

  dz[leg_index] = 0;  // feet stay on the ground
}
```

If the angular velocity of z is 0 (walking straight), the the legs move directly as the input dictates. The rotation motion is calculated in the *else* clause. Movement in the swing phase can be the same for waling straight or walking in circles, since the legs don't touch the ground in the swing phase.

This describes roughly the gait implementation. The detailed math might follow later if requested.

Inverse Kinematics

Here is where the math works.

Inverse Kinematics takes the coordinates of a foot an translates it to servo angels, which are required to reach this target point.

There are several explanations of how the math works online, a quick web search can help here.

Motion Control

Manages the different states like "Standing", "Walking", "Standing up", "Sitting down" etc.

Details

Motion Loop

Motion Control contains the main Finite State Machine, with these states:

The main loop of the motion controller. It is a FSM (Finite State Machine) which governs the state transitions.

There are following states present:

motion_control.hpp:

```
typedef enum
{
  Unknown,
  Idle,
  Sitting,  // Sitting, torque off
  Standing_up,
  Standing,
  Walking,
```

(continues on next page)

(continued from previous page)

```
  Sitting_down
} tState;
```

State: Unknown

Default state. Turn off servo motors. This is a save state that should never be reached.

motion_control.cpp:

```
// Loop in which the motions are coordinated
void MotionControl::motion_loop(void)
{
  switch (hex_state) {
    case Unknown:   // check in which state the robot is and go to idle
      servo_driver.free_servos();
      break;
```

State: Idle

Idle state is the default starting state.

State transitions

The robot waits for an input in form of the "start_cmd". If the command is received the state changes to Sitting while the servos are turned on.

motion_control.cpp:

```
    case Idle:
      if (start_cmd) {
        next_hex_state = Sitting;
        servo_driver.turn_on_servos();
      }
      break;
```

State: Sitting

In the sitting state the servo motors move to the neutral sitting position with a reduced speed. This enables a smooth transition from unknown servo positions to the neutral sitting position. (In the visualization this step is unconstrained, meaning it will take 0 time to execute).

State transitions

The stand_up_cmd resets the servos to full speed again, while transitioning to the standing up state. If the start_cmd is reset the robot goes back to the idle state.

motion_control.cpp:

```
    case Sitting:
      current_height = sitting_height;
      servo_driver.set_servo_speed(0.2);  // reduce speed to prevent jerking motion
```

(continues on next page)

(continued from previous page)

```
      for (int c = 0; c < NUMBER_OF_LEGS; c++) {
        feet.foot[c].position.x = neutral_foot_pos_x[c];
        feet.foot[c].position.y = neutral_foot_pos_y[c];
        feet.foot[c].position.z = current_height;
      }

      if (stand_up_cmd) {
        servo_driver.set_servo_speed(0);  // max speed
        next_hex_state = Standing_up;
      } else if (!start_cmd) {
        next_hex_state = Idle;
        servo_driver.free_servos();
      }
      break;
```

State: Standing_up

In the standing up state the height of the body is increased until it reaches the desired standing height.

State transitions

As soon as the standing height is reached the FSM transitions to the walking state.

motion_control.cpp:

```
    case Standing_up:
      current_height -= sit_step_height;
      if (current_height <= -walking_height) {
        next_hex_state = Walking;
        current_height = -walking_height;
      }

      for (int c = 0; c < NUMBER_OF_LEGS; c++) {
        // feet.foot[c].position.x = neutral_foot_pos_x[c];
        // feet.foot[c].position.y = neutral_foot_pos_y[c];
        feet.foot[c].position.z = current_height;
      }
      break;
```

State: Standing

In the standing state the robot only moves its body without moving the feet positions. This also serves as a proof of the inverse kinematics.

State transitions

If the stand_up_cmd is reset, the robot will sit down again. If the walking_cmd is set the robot goes to the Walking state.

motion_control.cpp:

```
    case Standing:
      body.orientation.pitch = 0.4 * cmd_vel_.linear.x;
      body.orientation.roll = 0.5 * cmd_vel_.linear.y;
      body.orientation.yaw = 0.5 * cmd_vel_.angular.z;

      if (!stand_up_cmd) {
        if (gait.is_standing()) {
          next_hex_state = Sitting_down;
        }
      }

      if (walking_cmd) {
        next_hex_state = Walking;
      }
      break;
```

State: Walking

This state simply invokes the gait_cycle, which is where it starts walking according to the user input.

State transitions

If the stand_up_cmd is reset, the robot will sit down again. If the walking_cmd is reset the robot goes to the Standing state.

motion_control.cpp:

```
    case Walking:
      gait.gait_cycle(cmd_vel_, &feet);
      /*
      std::cerr << "[          ]      feet(x,y,z) G1: (" <<
        feet.foot[0].position.x << ", " <<
        feet.foot[0].position.y << ", " <<
        feet.foot[0].position.z << ")" <<
        std::endl;
      */

      if (!stand_up_cmd) {
        if (gait.is_standing()) {
          next_hex_state = Sitting_down;
        }
      }
      if (!walking_cmd) {
        if (gait.is_standing()) {
          next_hex_state = Standing;
        }
      }
      break;
```

State: Sitting_down

This is the reversed motion from standing up.

motion_control.cpp:

```
    case Sitting_down:
      current_height += sit_step_height;
      if (current_height >= sitting_height) {
        current_height = sitting_height;
        next_hex_state = Sitting;
      }

      for (int c = 0; c < NUMBER_OF_LEGS; c++) {
        // feet.foot[c].position.x = neutral_foot_pos_x[c];
        // feet.foot[c].position.y = neutral_foot_pos_y[c];
        feet.foot[c].position.z = current_height;
      }
      break;
```

Transmitting the Results

After the State logic the new states are assigned:

motion_control.cpp:

```
hex_state = next_hex_state;
```

And the leg joint angles are translated into joint_states which are then transmitted to the servo driver:

motion_control.cpp:

```
// transmit the servo positions
if (hex_state != Idle) {
  if (inverse_kinematics.calculate_ik(feet, body, &legs)) {
    int i = 0;
    for (int leg_index = 0; leg_index < NUMBER_OF_LEGS; leg_index++) {
      joint_state_.position[i] = legs.leg[leg_index].coxa;
      i++;
      joint_state_.position[i] = legs.leg[leg_index].femur;
      i++;
      joint_state_.position[i] = legs.leg[leg_index].tibia;
      i++;
    }
    servo_driver.transmit_servo_positions(joint_state_);
  } else {
    RCLCPP_WARN(rclcpp::get_logger("Motion Control"), "Out of Range");
    feet = old_feet;  // Do not move any foot if one is out of range
  }
}

old_feet = feet;
```

Functions

Not all functions are listed here, there are some more in the code, some of which are unused, obsolete or used for testing.

Sit Down/Stand Up

Commands used to trigger transitions. Only works if the start_cmd flag is set.

motion_control.cpp:

```
126 void MotionControl::stand_up(void)
127 {
128   if (start_cmd) {
129     RCLCPP_INFO(rclcpp::get_logger("Motion Control"), "Stand up");
130     stand_up_cmd = true;
131   }
132 }
133
134 void MotionControl::sit_down(void)
135 {
136   if (start_cmd) {
137     RCLCPP_INFO(rclcpp::get_logger("Motion Control"), "Sit down");
138     stand_up_cmd = false;
139   }
140 }
```

Start & Stop Command

This command first checks if the hexapod is sitting, if yes it sets the flag to either turn the servo motors on or off.

motion_control.cpp:

```
143 void MotionControl::start_stop(void)
144 {
145   if (stand_up_cmd) {return;}
146   if (start_cmd) {
147     RCLCPP_INFO(rclcpp::get_logger("Motion Control"), "Stop");
148     start_cmd = false;
149   } else {
150     RCLCPP_INFO(rclcpp::get_logger("Motion Control"), "Start");
151     start_cmd = true;
152   }
153 }
```

Stand/Walk Command

This command sets the flag to either walk or move the body while standing.

motion_control.cpp:

```
155 void MotionControl::stand_walk(void)
156 {
157   if (!stand_up_cmd) {return;}
158   if (walking_cmd) {
159     RCLCPP_INFO(rclcpp::get_logger("Motion Control"), "Stand");
160     walking_cmd = false;
161   } else {
162     RCLCPP_INFO(rclcpp::get_logger("Motion Control"), "Walk");
163     walking_cmd = true;
164   }
165 }
```

Update Velocity Command

Sets a new velocity if the robot is in the standing position.

motion_control.cpp:

```
93  void MotionControl::update_vel(geometry_msgs::msg::Twist cmd_vel)
94  {
95    // if the sitdown commands arrives the velocity is set to 0.
96    // So the Hexapod can enter a standing state
97    if (stand_up_cmd) {
98      cmd_vel_ = cmd_vel;
99    } else {
100     cmd_vel_.linear.x = 0;
101     cmd_vel_.linear.y = 0;
102     cmd_vel_.angular.z = 0;
103   }
104 }
```

Servo Driver

Takes the joint angles as input, translates them into servo positions. Packages these positions and forwards them to the servo motors through the Dynamixel SDK interface.

1.4.3 Hexapod Description

Files and Data used to describe the Robot. Mainly used for RVIZ.

1.4.4 Hexapod Messages

Custom messages defined for the Hexapod communication.

Feet Position

Position of all feet, consist of 6 times a Pose:

```
hexapod_msgs/Pose[6] foot
```

Pose is used here instead of point because in the case of 4 joints the system is under-constraint and requires additional information. This additional information in this case is the RPY of the foot.

Legs Joints

Groups 6 leg joints together to describe the whole robot:

```
hexapod_msgs/LegJoints[6] leg
```

Leg Joints

All joint angles of a leg:

```
float64 coxa
float64 femur
float64 tibia
float64 tarsus
```

The tarsus joint is not used if the robot only has 3 joints.

Pose

A pose consists of a point and an orientation:

```
geometry_msgs/Point position
hexapod_msgs/RPY orientation
```

RPY

Roll Pitch Yaw expressed as float64:

```
float64 roll
float64 pitch
float64 yaw
```

1.4.5 Launch Files

Instructions to launch the application.

1.4.6 Supervisor

The Supervisor is an independent node which supervises battery power and temperature.

1.5 Status of the Project

1.5.1 Current Capabilities

Lepta can walk straight, turn, walk in circles or move its body without moving the feet. The robot is controlled using a Bluetooth joy pad, as time of writing only the XBOX One joy pad has been configured.

Lepta can perform the following actions as of time of writing:

- Stand up/Sit down
- Rotate the body without moving the feet (prove of inverse kinematics)
- Walking straight in arbitrary direction with arbitrary speed
- Turning around arbitrary point with arbitrary speed

1.5.2 Planned Features

These are just a few things I intend to implement in the future

Hardware

- Removable Battery (Make it easier)
- Relay to turn off the Servos completely
- Smaller DC/DC converter
- Sensors - LIDAR - IMU - Camera
- ...

Software

- Smooth leg motion
- Adjustable leg step height
- Further optimize gait
- Different gait styles
- …