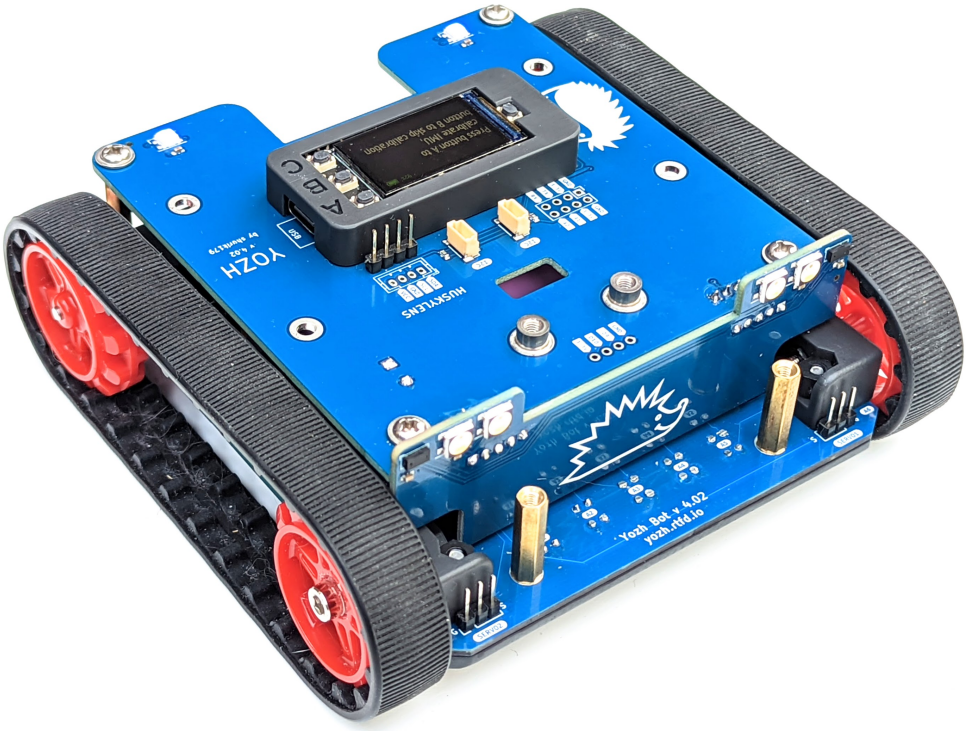
Yozh Robot

Alexander Kirillov

Nov 24, 2023

TABLE OF CONTENTS



Yozh is a small (about 13cm*13cm) robot, inspired by Pololu's Zumo robot. It was created by shurik179 for a robotics class at SigmaCamp. Below are the key features of this robot.

- **Dimensions:** Length: 12.4 cm; width: 13 cm; height: 4.9 cm
- **Power:** one or two 18650 Li-Ion batteries
- **Wheels and motors:** uses silicone tracks and 6V, HP, 75 gear ratio micro metal gearmotors, both by Pololu.
- **Main controller:** ESP32-S3 Feather board by Adafruit, which serves as robot brain. It is programmed by the user in CircuitPython, using a provided CircuitPython library. This library provides high-level commands such as *move forward by 30cm*
- **Electronics:** a custom electronics board, containing a secondary MCU (SAMD21) preprogrammed with firmware, which takes care of all low-level operations such as counting encoder pulses, controlling the motors using closed-loop PID algorithm to maintain constant speed, and more.
- Included sensors and other electronics
  - 240*135 **color TFT display** and 3 **buttons** for user interaction
  - Bottom-facing **reflectance array** with 7 sensors, for line-following and other similar tasks
  - Two front-facing **distance sensors**, using VL53L0X laser time-of-flight sensors, for obstacle avoidance
  - A 6 DOF **Inertial Motion Unit** (IMU), which can be used for determining robot orientation in space for precise navigation
  - Two RGB **LEDs** for light indication and a **buzzer** for sound signals
  - Four RGBW LEDs used as headlights
- Expansion ports and connections:
  - Two ports for connecting **servos**

- Two I2C ports, using Qwiic/Stemma QT connector
- Several available pin headers for connecting other electronics
- Yozh is compatible with mechanical attachments (grabber, forklift,...) by DFRobot.

All robot design is open source, available in github repository under MIT License, free for use by anyone. We also plan to create a Yozh kit which would be sold on Tindie for those who want to build the robot but do not have time or skill to assemble their own PCBs.

You can view photos and videos of Yozh here:

https://photos.app.goo.gl/grQfWu86DGW8zRTT8

CHAPTER

ONE

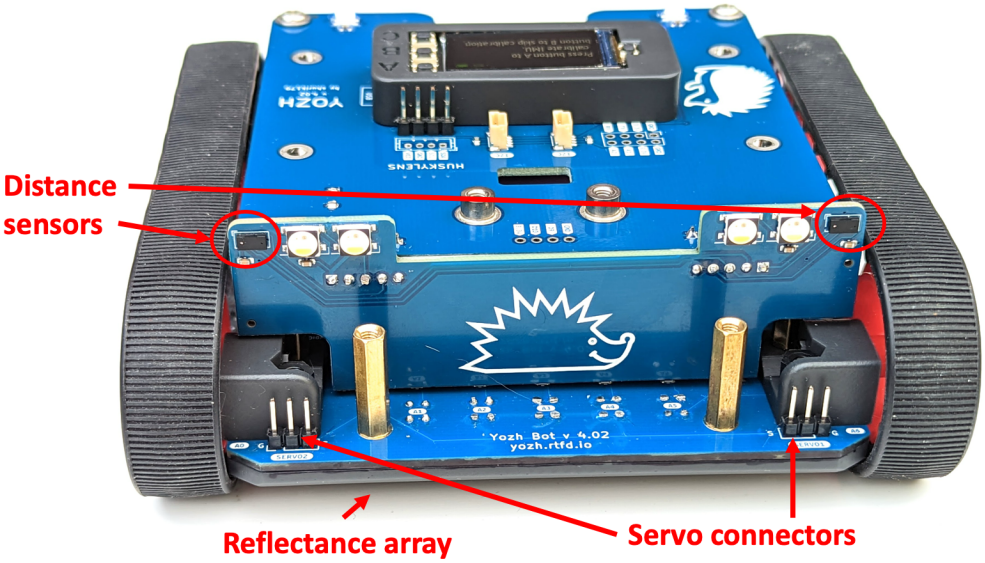
QUICK START GUIDE

If you got the Yozh robot as a kit or are building your own from scratch, please follow the instructions in Yozh Assembly Guide to put your robot together.

Once the robot is assembled, follow the steps below to get started quickly.

1.1 Yozh at a glance

The photos below show main features of Yozh:

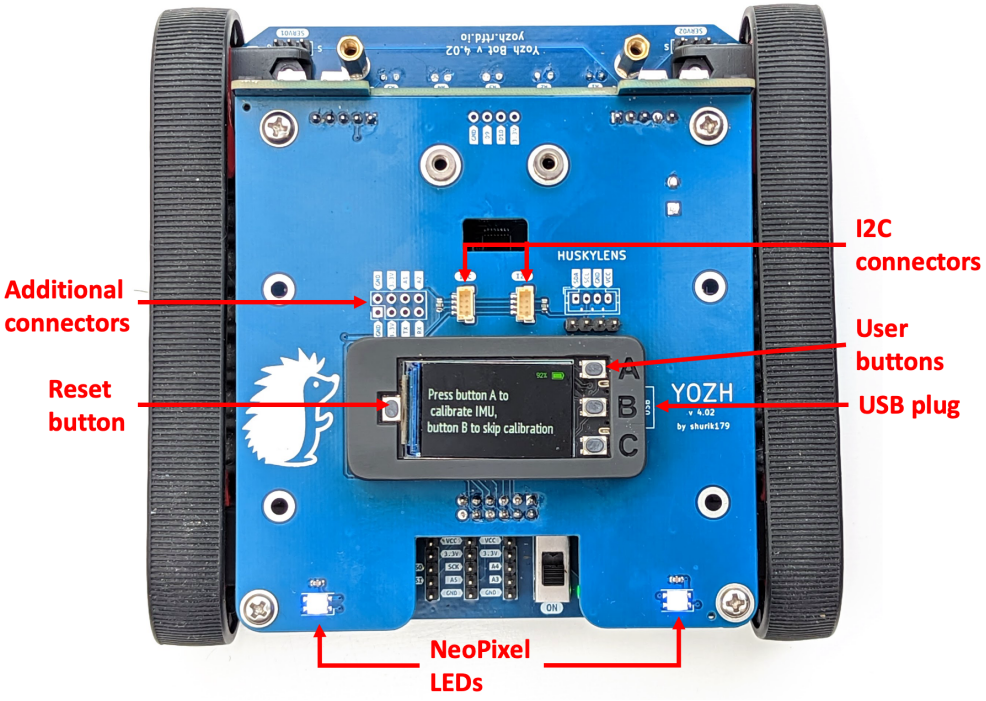


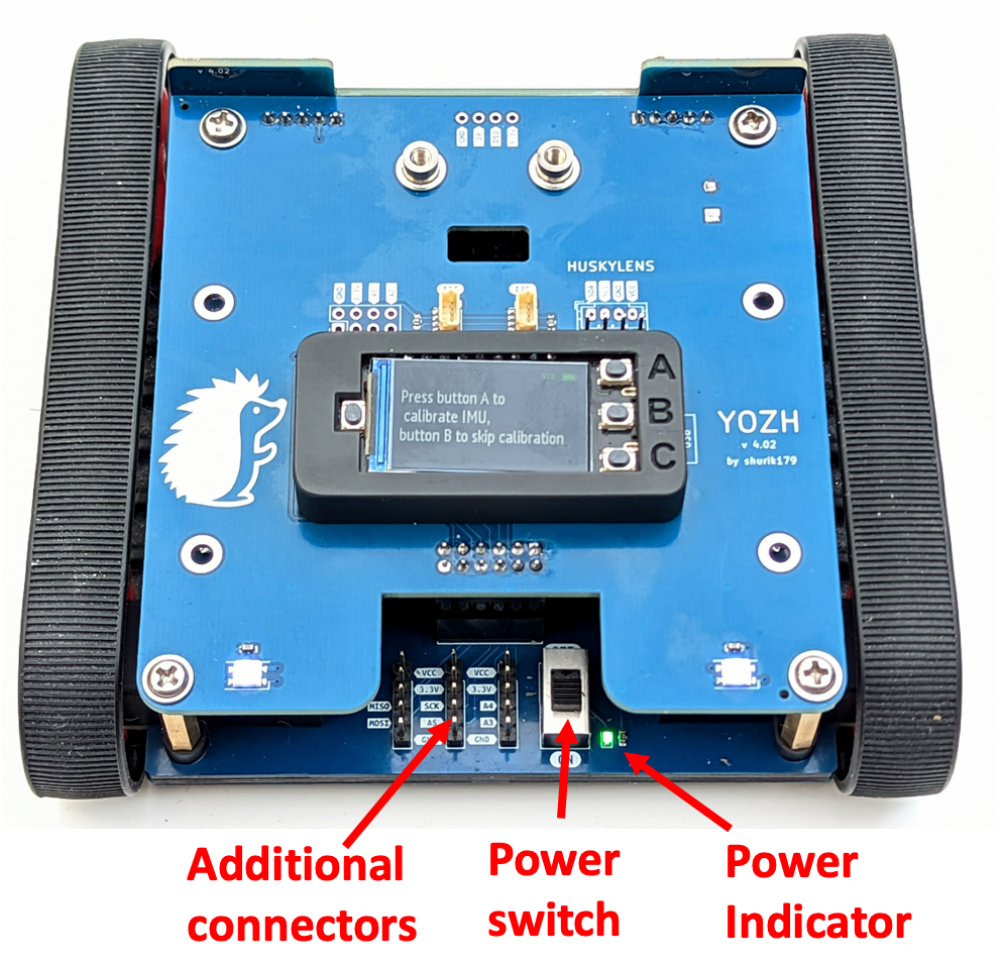
1.2 Installing the batteries

Yozh is powered by one or two 18650 Li-Ion batteries, inserted in the battery compartment inside the robot; to access it, you need to remove the top plate.

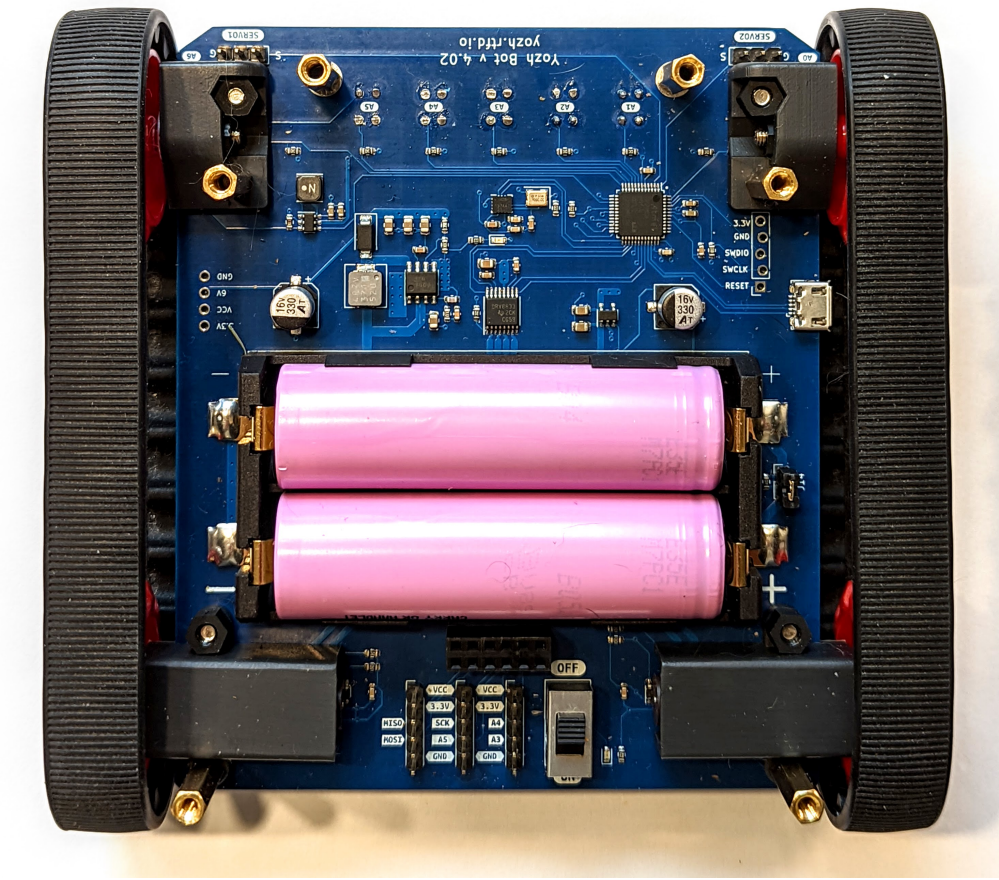
Warning: Li-Ion batteries can be dangerous if not handled right! please make sure to place them with correct polarity. Always **turn the power switch off** before removing the top plate, replacing the battery, or doing any other work on the robot.

For most purposes, it suffices to use one battery; place it in the slot closest to the back of the robot (of course, power switch should be off while you are doing it). Make sure to observe correct polarity as labeled on the PCB!


Additional connectors
Reset button
I2C connectors
User buttons
USB plug
NeoPixel LEDs
HUSKYLENS
Press button A to calibrate IMU, button B to skip calibration
A
B
C
YOZH
v 4.02
by shurik179


Additional connectors
Power switch
Power Indicator
HUSKYLENS
Press button A to calibrate IMU, button B to skip calibration
A
B
C
YOZH
v 4.02
by shurik179


YOZH robot v 4.02


yozh.rtfd.io
Yozh Bot v 4.02
SERVO1
SERVO2
3.3V
GND
SWDIO
SWCLK
RESET
OFF

See section `Power` in *Yozh Features in Detail* document for suggestions on choosing the best 18650 battery or proper method of installation if you want to use two batteries – there are some precautions to be observed!

1.3 Circuit Python library installation

Yozh is intended to be programmed in CircuitPython 8 - an implementation of Python programming language for microcontrollers, created by Adafruit (based on Micropython, another Python implementation). For general background on Circuit Python, please visit What is CircuitPython? page.

CircuitPython must already be installed on the ESP32-S3 microcontroller serving as the brains of Yozh Robot. If not, please do so now following Adafruit's instructions.

Next, you need to install Yozh Circuit Python library. Go to github repository and click on green `Code` button to download the zip file containing all Yozh designs and software. Extract the zip file to your computer. Find in the extracted archive folder *circuitpython_library*

Connect Yozh robot to the computer using a USB-C cable. It should appear in your file browser as an external drive with the name *CIRCUITPY*. Open it to view contents. Now, copy the following files and folders from the downloaded *circuitpython_library* folder to the *CIRCUITPY* folder:

- *yozh.py*
- *yozh_registers.py*
- *hedgehog.bmp*
- *fonts* folder
- *lib* folder (*CIRCUITPY* may already contain folder *lib*; if so, copy all contents of *circuitpython_library/lib* to *CIRCUITPY/lib*)

if you intend to use Yozh with Huskylens camera by DFrobot, you also need to copy file *huskylens.py*.

Please note that extracted Yozh archive also contains a folder *circuitpython_library/examples*. Move this folder to some convenient location on your computer - you will use it shortly.

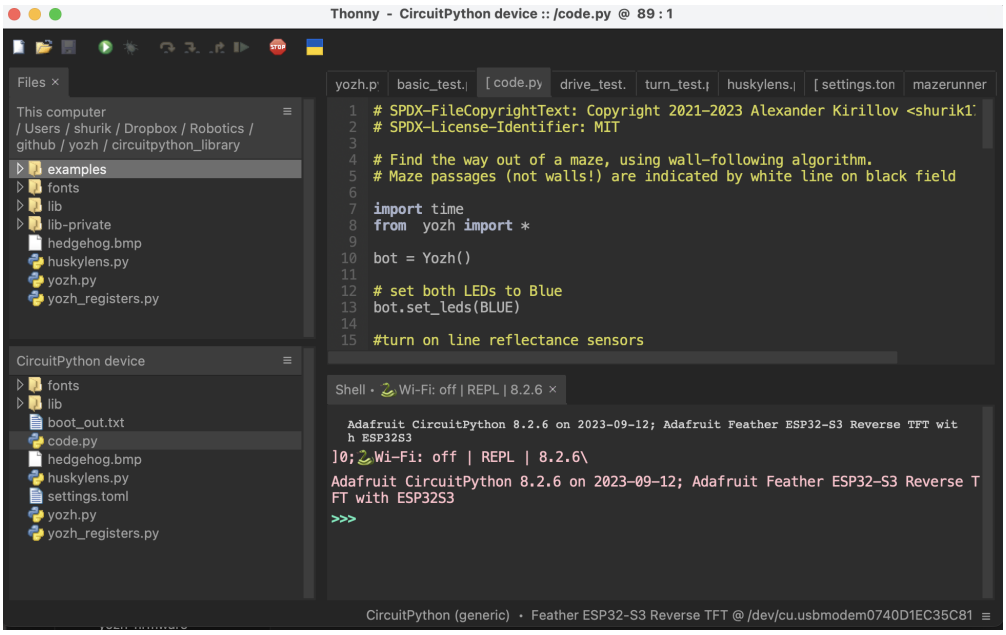
1.4 Thonny editor

We suggest using Thonny editor for creating and editing programs for your robot. Experienced programmers can use their favorite text editor instead - but please check this page for some common problems and list of recommended editors.

To install Thonny, visit Thonny webpage, download the installer for your system and run it.

After installation, you also need to configure Thonny to use the CircuitPython interpreter. Go to `Run` menu and select `Configure Interpeter`. Select `Circuit Python (Generic)`. While you are there, you can also adjust the theme and font to your liking – for example, you can switch to dark theme, shown below. After making your choices, click `OK` to save the configuration.

To verify the installation, connect your computer to the ESP32-S3 MCU of Yozh robot. Hit the red `Stop` button to reconnect to the board. Thonny should now show the files on the CURCUITPY drive, as shown below:



1.5 First program

The *CIRCUITPY* drive which was created during installation of CircuitPython on EPS32-S3 contains a special file, *code.py*. This file always contains the code of the current program running on the board. As soon as you turn the robot on or hit reset, the robot starts executing this program.

To test the robot, connect it to the computer, using the USB-C connector of the ESP32-S3. **Make sure the robot switch is in ON position**. Start Thonny editor and open *code.py* file in *CIRCUITPY* drive, by clicking on it in the navigation pane on the left or using `File->Open` menu item. Erase everything in that file so it is blank.

Now, use the top part of the navigataion panel to open the folder with examples from Yozh library you downloaded previously. In that folder, find the file *basic_test.py* and click on it to open it in another tab of Thonny editor. Copy the whole contents of *basic_test.py* file and then paste it in *code.py* file.

Now click on the green `Run` button on Thonny toolbar to save *code.py* file and run it. The robot will execute your first program! Look at the OLED screen, read the prompts, press the buttons, and have fun.

The code is amply commented, so it is easy to make changes.

Note: after running the code, or after disconnecting and reconnecting the robot to the computer, you need to again hit the `Stop` button for the editor to reconnnect to the robot. Try modifying the code (e.g. changing the text printed to screen) and then re-save it.

1.6 Serial console

For debugging the program, one needs to print some information such as variable values and error messages. Python has built-in command *print()* which does that. The output of print command is sent to *serial console* - which in practice just means that it is sent over USB to the computer.

Thonny editor has built-in serial console, so you can see these messages. By default, the serial console appears at the bottom of the screen.

Among other features it provides is the ability to enter Python commands interactively in the console, without saving them to a file - this is very useful for testing various things. This is called REPL (Read-Evaluate-Print Loop); see

https://learn.adafruit.com/welcome-to-circuitpython/the-repl for more info.

1.7 Commonly used functions

Below is the list of most commonly used functions from Yozh CircuitPython library. This is not a full list! See *Library reference* for full list and details.

To begin using the library, you need to put the following in the beginning of your *code.py* file:

```
import time
from  yozh import *
bot = Yozh()
```

This creates and initializes an object with name `bot`, representing your robot. From now on, all commands you give to the robot will be functions and properties of `bot` object. We will not include the name bot in our references below; for example, to use a command `stop_motors()` described below, you would need to write `bot.stop_motors()`.

1.7.1 Display, buttons

`clear_display()`

Clears all text and graphics from display

`set_text(line_number, message)`

Print given message on a given line of the display. Line number can range 0–5. Note: this command supports more options; check the library reference.

`wait_for(button)`

Waits until the user presses the given button. There are three possible pre-defined buttons: `BUTTON_A`, `BUTTON_B`, `BUTTON_C`.

There are also functions for checking if a given button is currently pressed, or waiting until the user presses one of 3 buttons.

1.7.2 LEDs, buzzer, headlights

`set_lights(power)`

Turns the headlights on/off. Power should be between 0-100; setting the power to zero turns the headlights off.

`set_leds(color_l, color_r)`

Set colors of both LEDs at the same time. Each color can be a triple giving values of red, green, and blue (each 0-255) or a hex number: `set_leds( (255,0,0), 0xFF0000)`. You can also use one of predefined colors: RED, GREEN, BLUE, YELLOW, WHITE, OFF, e.g. `set_leds(BLUE)`. Parameter `color_r` is optional; if omitted, both LEDs will be set to the same color.

`buzz(freq, dur=0.5)`

Buzz at given frequency (in hertz) for given duration (in seconds). Second parameter is optional; if omitted, duration of 0.5 seconds is used.

1.7.3 Driving

go_forward (distance, speed=60)

go_backward(distance, speed=60)

Move forward/backward by given distance (in centimeters). Parameter `speed`, which ranges between 0-100, is optional; if not given, default speed of 60 is used. Note that distance and speed should always be positive, even when moving backward.

Behind the scenes, these commands try to maintain constant robot speed and direction. To learn more about how it is done check section FIXME.

turn(angle, speed=60)

Turn by given angle, in degrees. Positive values correspond to turning right (clockwise). Parameter `speed` is optional; if not given, default speed of 50 (i.e. half of maximal) is used.

set_motors(power_L, power_R)

Set power for left and right motors. `power_L` is power to left motor, `power_R` is power to right motor. Each of them should be between 100 (full speed forward) and -100 (full speed backward).

Note that because no two motors are exactly identical, even if you give both motors same power (e.g. `set_motors(60,60)`), their speeds might be slightly different, causing the robot to veer to one side instead of moving straight. To avoid this, use `go_forward()` command described above.

stop_motors()

Stop both motors.

1.7.4 Inertial Motion Unit (IMU)

Before use, the IMU needs to be calibrated. The calibration process determines and then applies corrections (offsets) to the raw data; without these corrections, the data returned by the sensor is very inaccurate.

If you haven't calibrated the sensor before (or want to recalibrate it), use the following function:

IMU_calibrate()

This function will determine and apply the corrections; it will also save these corrections in the flash storage of the Yozh secondary microcontroller, where they will be stored for future use. This data is preserved even after you power off the robot (much like the usual USB flash drive).

This function will take about 10 seconds to execute; during this time, the robot must be completely stationary on a flat horizontal surface.

If you had previously calibrated the sensor, you do not need to repeat the calibration process - by default, upon initialization the IMU loads previously saved calibration values.

IMU_yaw()

Returns robot yaw, i.e. heading in horizontal plane. Note that zero heading is rather random (it is not the starting position of the robot!). Positive values correspond to turning right (clockwise).

1.7.5 Reflectance array

`linearray_on()`

`linearray_off()`

Turns reflectance array on/off. By default, it is off (to save power).

`calibrate()`

Calibrates the sensors, recording the black values. This command should be called when all of the sensors are on the black area of the field. This is necessary for the commands below.

`sensor_on_white(i)`

`sensor_on_black(i)`

Returns `True` if sensor `i` is on white (respectively, black) and false otherwise. Index *i* ranges from 0 (rightmost sensor) to 6 (leftmost)

`all_on_white()`

`all_on_black()`

Returns `True` if all sensors are on white (respectively, black) and false otherwise.

`line_position_white()`

`line_position_black()`

Returns a number showing position of the line under the robot, assuming white line on black background (respectively, black line on white background). The number ranges between -4 (line far to the left of the robot) to 4 (line far to the right of the robot). 0 is central position: line is exactly under the center of the robot. See *Library reference* for details.

1.8 More examples

Now that you have learned how to write and save programs to the robot, it is time to explore Yozh capabilities. To help with that, we have provided a number of examples, which can be found in *examples* folder of the Yozh library you had downloaded previously. Try opening and running them to see what the robot can do.

Below is the list of provided examples (as of Nov 1, 2023):

- *basic_test.py* - basic test of robot operation, including OLED display, LEDs, and buttons
- *motor_test.py* - testing basic operation of motors and encoders
- *drive_test.py* - testing higher-level drive commands, such as *go forward for 10cm* or *turn 90 degrees*
- *servo_test.py* - testing servos (if you have any attached).
- *imu_test.py* - testing IMU
- *pid_test.py* - testing PID control of motors
- *distancesensors_test.py* - testing operation of front-facing distance sensors
- *linearray_test.py* - testing reflectance sensor array
- *line_following.py* - following a line (requires a white line 1-3 cm wide, on black background)
- *obstacle_avoidance.py* - moving around and using distance sensors to avoid obstacles

All of these examples are amply commented, so it should be easy to understand how the code works and how to modify it.

1.9 Next steps

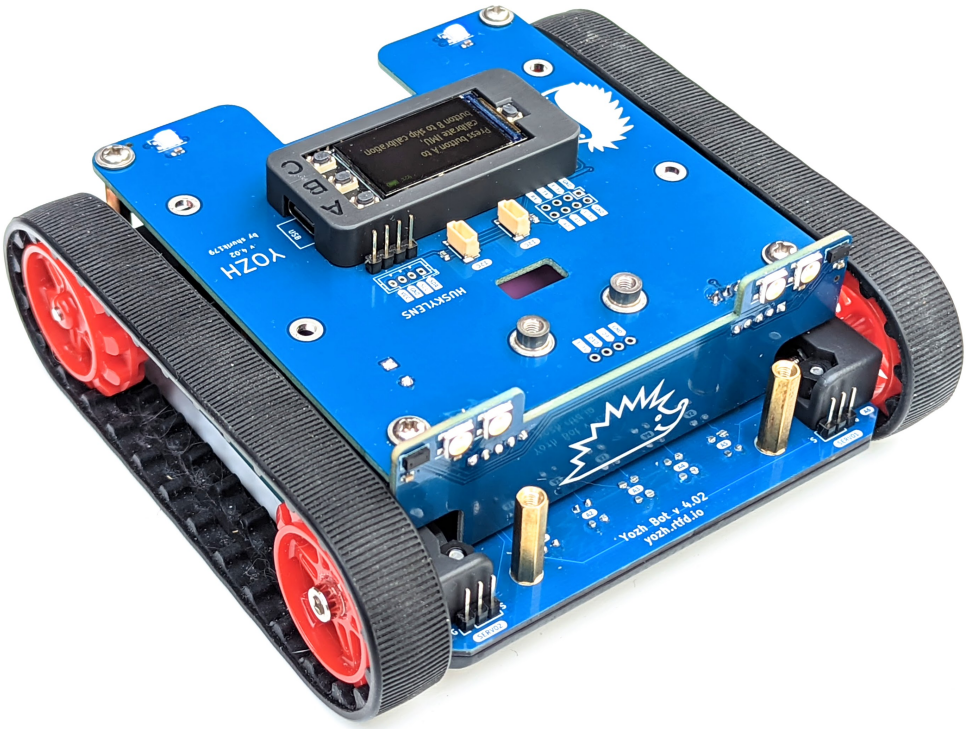
These examples give you some idea of what Yozh is capable for. But if you need to go deeper, check *Library guide* for full list of available commands, and *Yozh feature description* for a detailed description of Yozh hardware and specs.

And if you have any questions or comments, please reach out to us at irobotics.store@gmail.com.

CHAPTER

TWO

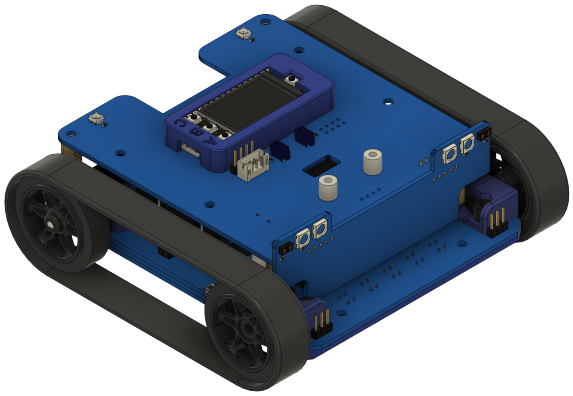
YOZH FEATURES IN DETAIL

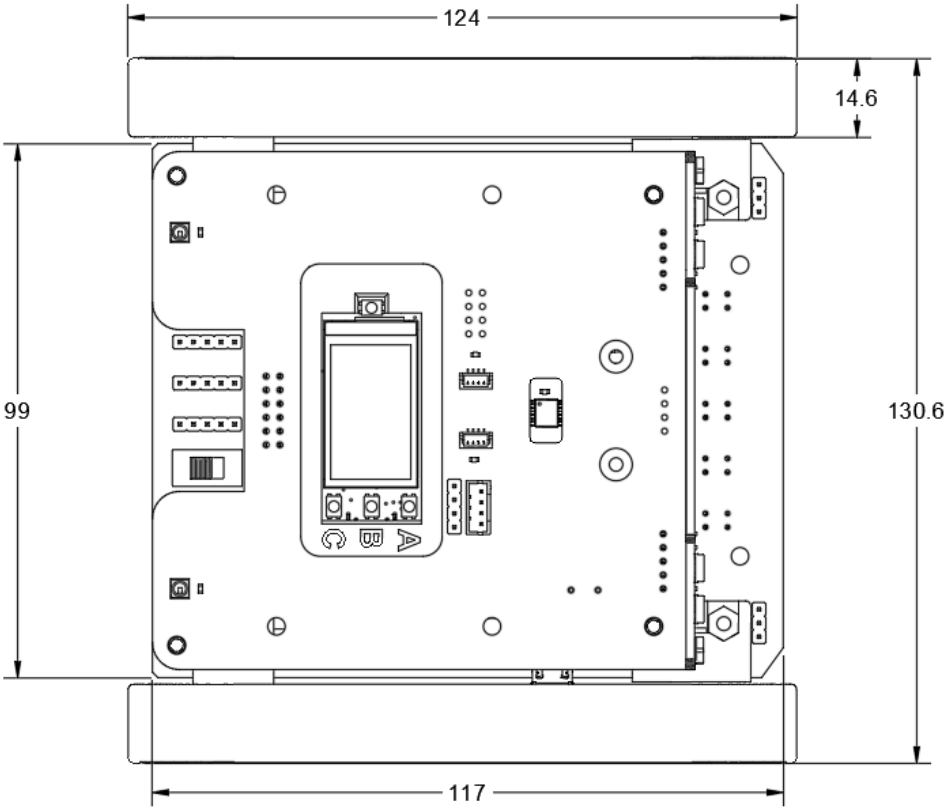


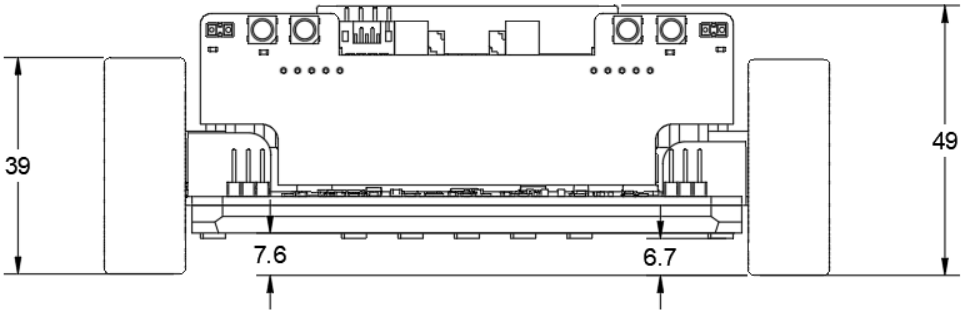
2.1 Dimensions and 3d model

Yozh dimensions are 12.4 cm (Length) * 13 cm (width) * 4.9 cm (height). Ground clearance is about 7 mm; however, since all sensors and everything else is inside the tracks profile (looking from the side), the robot is capable of going over (long) obstacles up to about 1.5 cm.

You can view the 3d model, created in Fusion360, at https://a360.co/40igAyW




124
14.6
99
130.6
117



2.2 Power

2.2.1 Batteries

Yozh is powered by one or two 18650 Li-Ion batteries, inserted in the battery compartment inside the robot; to access it, you need to remove the top plate. See the section below for discussion of whether you need one or two batteries.

Warning: Li-Ion batteries can be dangerous if not handled right! please make sure to place them with correct polarity, and follow the instructions in the next section if you use two batteries. Always turn the power switch off before removing the top plate or doing any other work on the robot.

It is highly recommended that you use batteries from a trusted manufacturer, such as Panasonic, Samsung, or Sanyo; do not try to save a couple of dollars by buying a no-name battery from Amazon or eBay - instead, use one of specialized shops such as https://www.18650batterystore.com/. You need **flat top unprotected batteries**; look for batteries with capacity 3000 mAh or more. Current rating is less important (you need a battery rated for 4A continuous current or more – this is low by the standards of 18650 battery cells). A good choice is this battery: https://www.18650batterystore.com/products/samsung-35e

The robot also contains power switch, for disconnecting the battery, and a power indicator LED. You can check the battery voltage in software, using `battery_voltage()` function as described in *Yozh Library Guide*. Fully charged Li-Ion batteries should read about 4.2v.

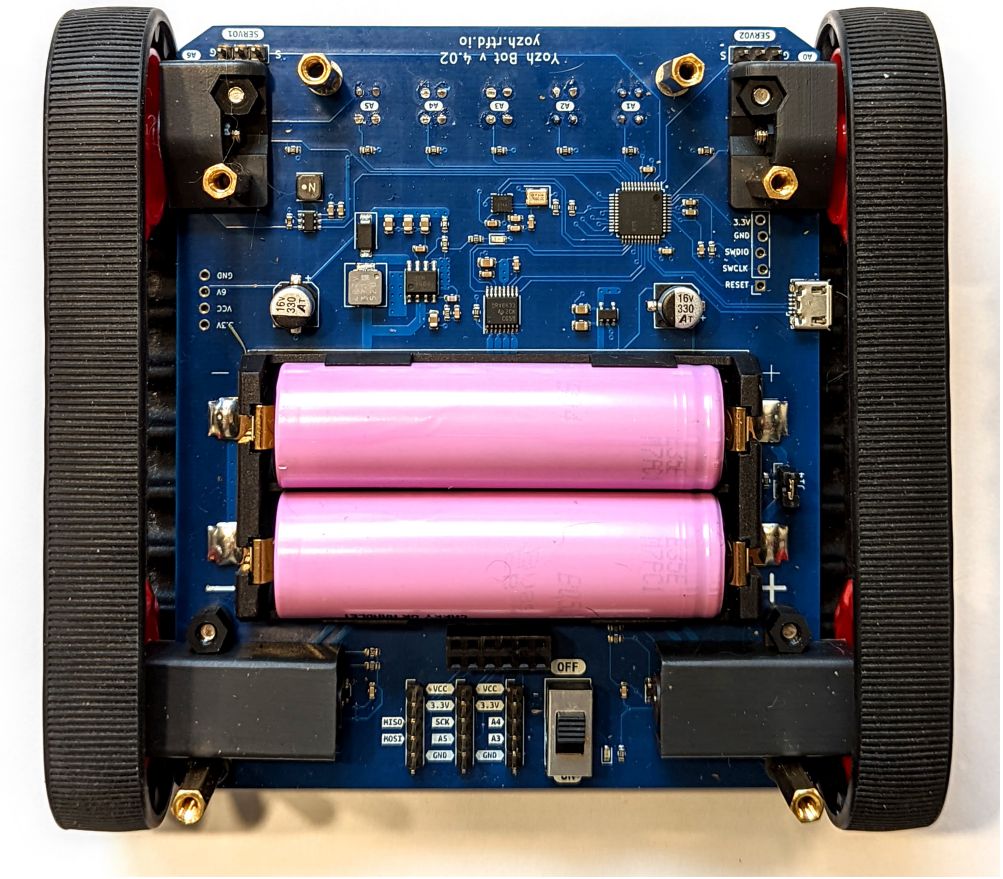
2.2.2 One or two batteries?

For most purposes, one 18650 battery is sufficient. Place it in the slot closest to the back of the robot. **Make sure to place it with correct polarity;** positive and negative terminals are labeled on the robot PCB.

If you are planning on using accessories that might use significant current, such as large size servos or AI cameras, or if you want to run the robot for long periods, you might want to add a second battery; these two batteries will be connected in parallel.

If you want to use two batteries, please observe these precautions. Please take them seriously!

- it is best to use the two identical batteries (same manufacturer and model)
- before inserting the batteries, turn the power switch to off and **remove the jumper J14** next to the batteries. After this, insert the batteries; **make sure to place them with correct polarity** as labeled on the PCB. Leave them inserted for a couple of hours or so, keeping the power switch off. After two hours, put the jumper J14 back on. (This allows the two batteries to equalize the voltage. The positive terminals are connected through on-board 1 Ohm resistor. Jumper J14 shorts it.)


Yozh Bot v 4.02
yozh.rtfd.io
SERVO1
SERVO2
3.3V
GND
SWDIO
SWCLK
RESET
OFF
+VCC
3.3V
SCK
MISO
MOSI
A5
GND
VCC
3.3V
A4
A3
GND

2.2.3 Voltages used by the robot

The robot contains a voltage regulator, which converts battery voltage to regulated 3.3v. This regulator provides power to the secondary MCU, IMU and distance sensors, leaving about 300 mA available for use by extra sensors you might connect to Yozh.

Yozh also contains a boost converter, converting battery voltage to regulated 6V. This is used to power the motors and servos.

Finally, some of the on-board electronics are powered directly from the battery voltage: the main MCU and TFT screen, Neopixels and headlights.

Connecting the ESP32-S3 microcontroller to a computer by USB cable provides power to the MCU even if the main battery is off. This would activate the main MCU and some of the electronics, but not the secondary MCU, motors or servos.

2.3 Chassis and motors

Yozh robot uses custom-made chassis, using silicone tracks by Pololu together with two 75:1 High Power 6V microgear motors. The motors are equipped with encoders (rotation counters), for speed control. The motors are controlled by DRV8833 motor driver by TI.

Maximal speed of the robot is FIXME m/s.

2.4 Electronics

The robot is controlled by two microcontrollers (MCU):

- Main (master) MCU: ESP32-S3 Reverse TFT Feather. This MCU is programmed by the user in CircuitPython. Provided CircuitPython library, documented in *Yozh Library Guide*, provides convenient functions for using all features of the robot.
- Secondary (slave) MCU: SAMD21G18A. This MCU is responsible for all low-level operations, converting high-level commands coming from main MCU into signals sent to motors, servos and more, thus freeing pins and other resources of the main MCU for other purposes. Secondary MCU is also responsible for counting the encoder pulses and running the PID control loop maintaining motor speed. This MCU comes preloaded with firmware, written in C++ (using Arduino IDE). Normally, the user shouldn't need to touch this firmware.

The two MCUs talk to each other using I2C communication protocol; main MCU acts as the master on the I2C bus, and the secondary acts as slave.

Some of Yozh hardware is directly controlled by the main MCU, without going through the secondary one:

- TFT display
- Buttons
- Buzzer
- Distance sensors
- Two indicator NeoPixel leds
- battery voltage monitor

Everything else – motors, encoders, servos, reflectance sensor array, Inertial Motion Unit – is handled by the secondary MCU.

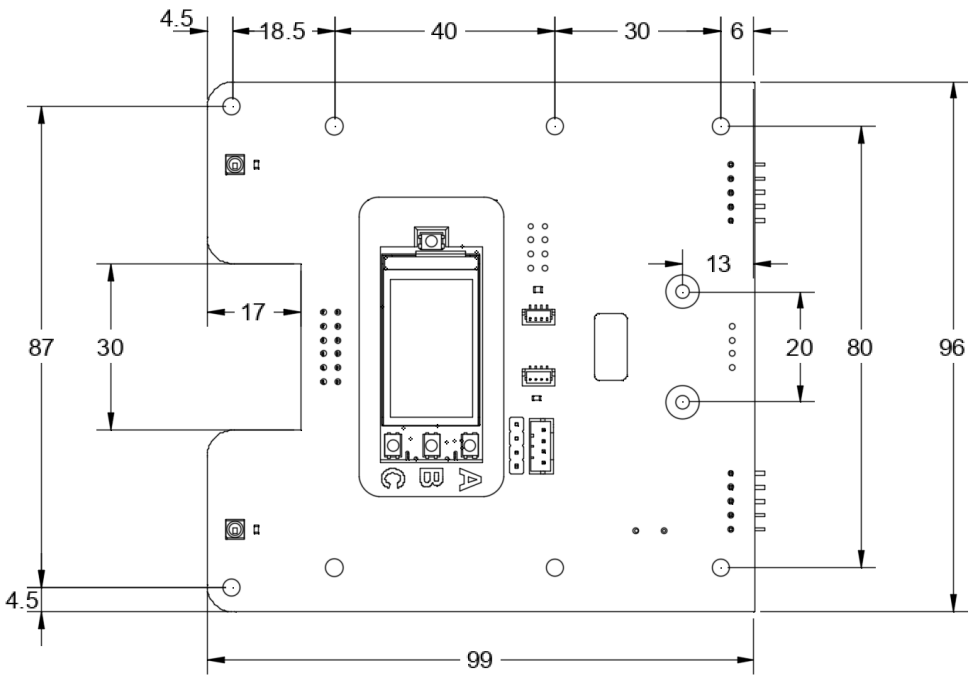
2.5 Top plate

On the top of Yozh robot, there is a top plate containing the following elements:

- The main MCU (ESP32-S3 Feather), with 240x135 color TFT display, reset button, and three buttons (labeled A, B, C) for user interaction
- two Qwiic/Stemma QT I2C connectors
- several connectors for connecting additional electronics (see FIXME for details)
- a number of 3mm mounting holes for attaching additional electronics

The top plate is mounted on the robot using 20mm long M3 standoffs. If necessary, it can be removed to access the robot interior. **Warning**: always turn the robot off before removing the top plate or doing any other work on the robot.

The diagram below shows dimensions and hole locations.



2.6 Buzzer and LEDs

Yozh contains a buzzer and two addressable RGB LEDs (commonly called NeoPixels), which can be used for showing information to the user.

Finally, Yozh contains four RGBW NeoPixels of the front panel, which are intended to be used as headlights. Normally, one only uses the white LED part of these NeoPixels; however, if necessary, you can also use the RGB part to create headlights of any color.

2.7 Inertial Motion Unit

The robot also contains an inertial motion unit (IMU): ICM 42688 by Invensense. This chip contains an accelerometer and a gyroscope, allowing the user to measure acceleration and rotation velocity. In addition to raw readings, the secondary MCU also runs a sensor fusion algorithm which uses the accelerometer and gyro data to constantly compute robot orientation in space, giving yaw, pitch, and roll angles. This can be used for precise navigation.

2.8 Distance and reflectance sensors

Yozh robot has several built-in sensors.

2.8.1 Reflectance array

In the front of the robot, there is an array of 7 down-facing reflectance sensors for detecting field borders, following the line, and other similar tasks. It uses ITR9909 sensors by Everlight. The sensors are labeled A0 (rightmost) through A6 (leftmost).

2.8.2 Distance sensors

Yozh also contains a front-facing board with two VL53L0X Time-of-Flight laser distance sensors by ST Microelectronics. These sensors have maximal distance of 2m; reliable sensing distance is closer to 1.5m. Each sensor has 25 degree field of view; this leaves a very small "blind spot" immediately in front of the robot, but provides complete coverage enabling the robot to detect any obstacle placed between 15-150 cm away.

These sensors can be used for obstacle avoidance, object tracking, or other similar purposes.

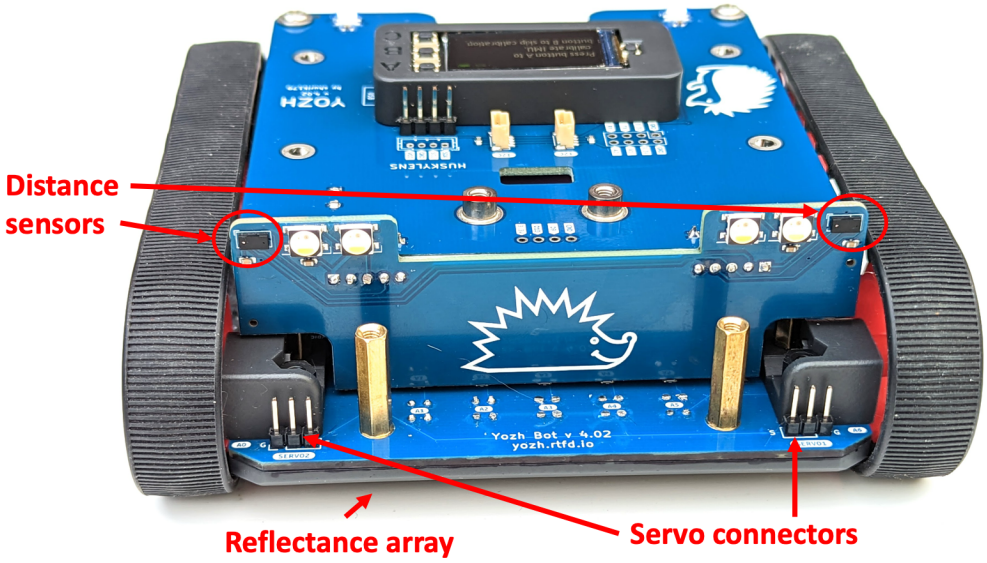
2.9 Connecting additional devices

Youzh provides connectors for additional sensors and other eelctroncis, as described below.

2.9.1 Top plate

The top plate contains the following connectors:

- two Qwiic/StemmaQT connectors, for connecting I2C sensors. These connectors use the default I2C pins of ESP32-S3 board. The same I2C bus is also used for connecting to secondary MCU (I2C address 0x11) and distance sensors (I2C addresses 0x29, 0x30); thus, make sure that the devices you connect have I2C addresses different from those listed above. Pull-up resistors (3.3K) on SCL and SDA lines are already provided, so there is no need to add your own.
- 8-pin connector (unpopulated), containing pins for 3.3v, GND, and pins A1, A2, TX, RX of ESP32-S3



- Huskylens connector. This is also gives access to the same I2C bus; however, instead of 3.3V, it provides VCC, which is directly connected to the battery (thus, voltage ranges from 4.2-3.5V depending on battery charge.) In addition, the pin order is different from the one used by Qwiic. This connector provides both standard male pin headers and JST PH 4-pin connector, which share the same pins. This connector is primarily intended for connecting Huskylens AI camewra by DFrobot, but coudl also be utilized by other devices as long as they are capable of using the battery voltage.

  Note that the SDA and SCL lines are pulled up to 3.3v, not to VCC.

2.9.2 Main board connectors

In the rear of the main robot board, there are even more connectors, providing access to SPI (MISO/MOSI/SCK) pns of ESP32-S3 and pins A3, A4, A5 (you can use one of them as chip select line for SPI).

Note that the SPI bus is also used for communication with the display.

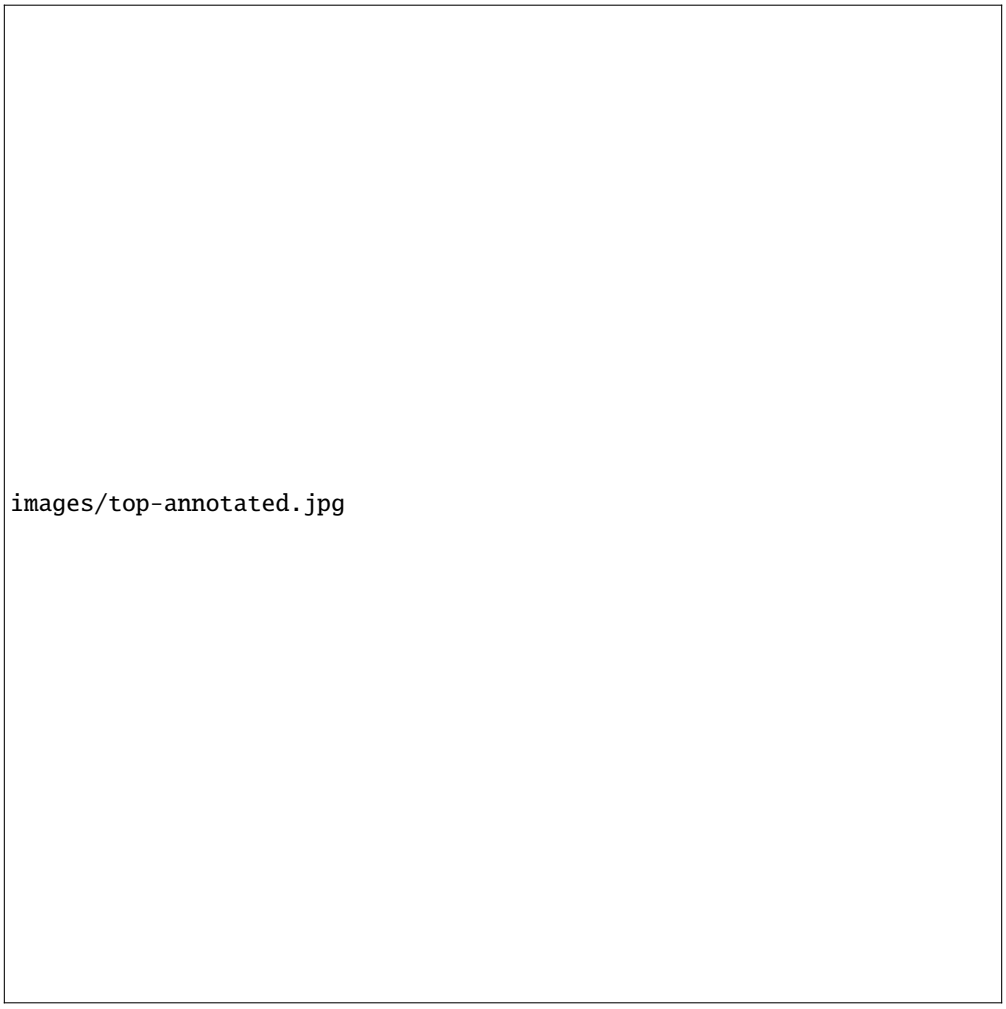
2.10 Servos and attachments

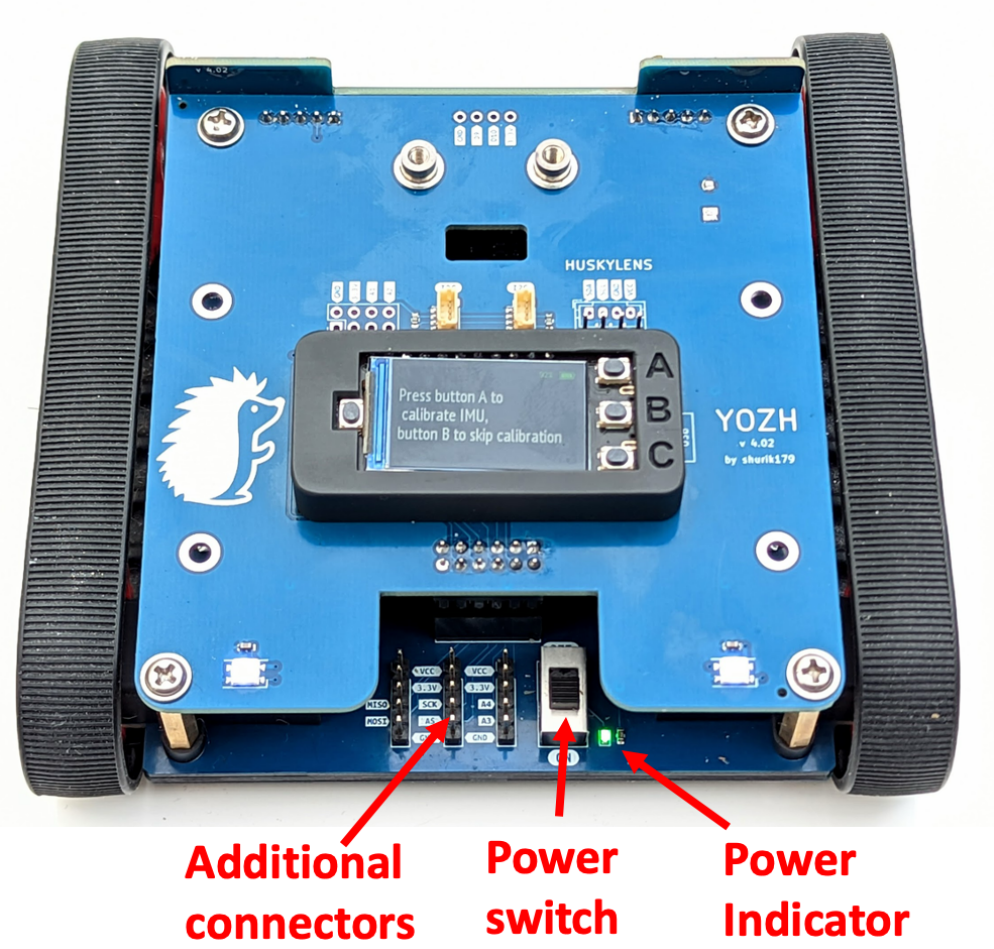
Yozh contains two ports for connecting servos, as shown in the photo below. These ports use standard pin order: GND, 6V, SIGNAL. To help you identify the pins, there are small letters S (Signal) and G (GND) next to corresponding pins. 6V pin is connected to the boost converter that converts battery voltage to 6V.

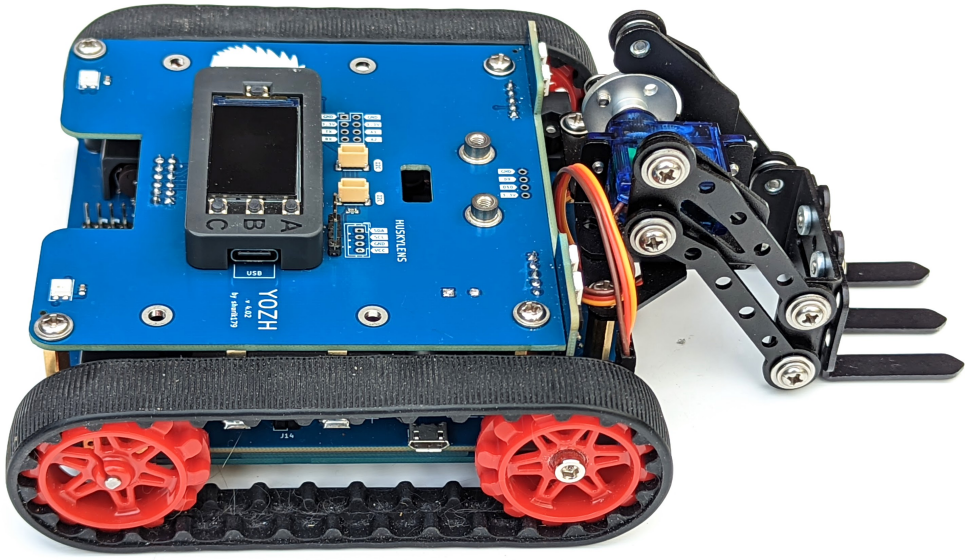
Yozh can also be used with mechanical attachments (grabber, forklift,...) by DFRobot – see photo below for Yozh with the forklift attachment. US customers might find it easier to order DFrobot kits from DigiKey:

- Grabber
- Forklift
- Loader

Note that some of these attachments might interfere with the distance sensors.


images/top-annotated.jpg


HUSKYLENS
Press button A to
calibrate IMU,
button B to skip calibration
A
B
C
YOZH
v 4.02
by shurik179
Additional
connectors
Power
switch
Power
Indicator


YOZH
HUSKYLENS
USB

CHAPTER

THREE

YOZH LIBRARY REFERENCE

In this chapter, we give full list of all commands provided by Yozh Circuit Python library. We assume that the user has already installed Yozh library on the robot, as described in the *Quickstart Guide*.

This document describes version 4.0 of the library. It is intended to be used with CircuitPython 7.

3.1 Initialization and general functions

To begin using the library, you need to put the following in the beginning of your *code.py* file:

```
import time
from  yozh import *
bot = Yozh()
```

This creates and initializes an object with name `bot`, representing your robot. From now on, all commands you give to the robot will be functions and properties of `bot` object. We will not include the name bot in our references below; for example, to use a command `stop_motors()` described below, you would need to write `bot.stop_motors()`.

Here are some basic functions:

`fw_version()`

Returns firmware version as a string. e.g. *2.1*.

`battery_voltage()`

Returns battery voltage, in volts. For normal operation it should be at least 3.7 V. Fully charge battery produces about 4.1 - 4.2 V.

`battery_percent()`

Returns charge percent (0 - 100).

3.2 Display, buttons, LEDs

Yozh contains a buzzer, two NeoPixel LEDs in the back and an 240x135 color TFT screen and three buttons on the top plate, for interaction with the user. To control them, use the functions below.

3.2.1 Headlights

set_lights(power)

Turns the headlights on/off. Power should be between 0-100; setting the power to zero turns the headlights off.

Note: headlights are capable of being controlled individuallly, and in fact one could set them to any color (they are RGBW Neopixels).mHowever, at the moment this is not supported by yozh library.

3.2.2 LEDs

set_led_L(color)

set_led_R(color)

These commands set the left (respectively, right) LED to given color. Color must be one of the following:

- a tuple of 3 numbers, showing the values of Red, Green, and Blue colors, each ranging between 0–255, e.g. `bot.set_led_L( (255,0,0))` to set the left LED red.
- A 32-bit integer, usually written in the hexadecimal form: `0xRRGGBB`, where each letter stands for a hexadecimal digit 0…F. E.g. `0xFF0000` is the same as `(255,0,0)` ande defines the red color.
- One of predefined colors, e.g. `RED`. Full list of predefined colors is: RED, GREEN, BLUE, YELLOW, WHITE, OFF. You can also define your own colors, e.g.

```
ORANGE=0xFFA500

bot.set_led_L(ORANGE)
```

set_leds(color_l, color_r)

Set colors of both LEDs at the same time. Parameter `color_r` is optional; if omitted, both LEDs will be set to the same color.

3.2.3 Buzzer

buzz(freq, dur=0.5)

Buzz at given frequency (in hertz) for given duration (in seconds). Second parameter is optional; if omitted, duration of 0.5 seconds is used.

3.2.4 Buttons

wait_for(button)

Waits until the user presses the given button. There are three possible pre-defined buttons: `BUTTON_A`, `BUTTON_B`, `BUTTON_C`.

is_pressed(button)

Returns `True` if given button is currently pressed and `False` otherwise.

choose_button()

Waits until the user presses one of the buttons. This function returns string literal `A`, `B`, or `C` depending on the pressed button:

```
bot.set_text(1, "Press any button")
#wait until user presses one of buttons
if (bot.choose_button()=="A"):
    # do something
else:
    # do something else
```

3.2.5 Display

The easiest way to interact with the TFT display is by using the commands below.

clear_display(hide_battery = False)

Clears all text and graphics from display. Optional parameter `hide_battery` indicates whether the battery level indicator should be removed as well; by default, it is false, so the battery level indicator is preserved.

set_text(line_number, message, font, color)

Print given message on a given line of the display. Line number can range 0–5. Parameters `font` and `color` are optional: if omitted, default font and white color are used.

The basic use of this command is

```
bot.set_text(0, "Press A to continue")
```

You can print multi-line messages, separating lines by \n, e.g.

```
bot.set_text(1, "Put robot on black \nand press A to continue")
```

This will print `Put robot on black` on line 1 and `and press A to continue` on line 2.

To use a different font, use optional parameter `font`. Posible choices are:

- `FONT_REGULAR`: usual font
- `FONT_BOLD`: slightly larger bold font
- `FONT_SMALL`: really small font, useful for long messages

update_battery_display()

Updates the battery level indicator.

Advanced users may also use any commands from CircuitPython `displayio` module to put text and graphics on the TFT display as described in https://learn.adafruit.com/circuitpython-display-support-using-displayio. The display object of the robot can be accessed as `bot.display`, and the root group of the display is `bot.canvas`. E.g., one could use

```
label=bitmap_label.Label(font = FONT_BOLD, text="DANGER", color = 0xFF0000, scale = 2,␣
↪x=50, y=30)
bot.canvas.append(label)
bot.display.refresh()
```

Note that `display.auto_refresh` property is set to `False`, so you need to explicitly call `display.refresh()` function. Also, the battery level indicator is not automatically updated: you need to call `update_battery_display()` to update it.

3.3 Motor control

Of course, main use of this robot is to drive around, and for this, we need to control the motors.

3.3.1 Basic driving

Yozh python library provides high level commands for controlling the robot.

go_forward (distance, speed=60)

go_backward(distance, speed=60)

Move forward/backward by given distance (in centimeters). Parameter `speed`, which ranges between 0-100, is optional; if not given, default speed of 60 is used. Note that distance and speed should always be positive, even when moving backward. Behind the scenes, these commands try to maintain constant robot speed and direction. To learn more about how it is done check section FIXME.

You can use special value *UNLIMITED* for distance; in this case, the command starts the robot moving forward without any distance limit. You will need to issue a separate command to stop it, e.g.

```
bot.go_forward(UNLIMITED, 50) #start moving forward a 50% speed
time.sleep(1.0)  # wait for 1 second
bot.stop_motors()
```

turn(angle, speed=60)

Turn by given angle, in degrees. Positive values correspond to turning right (clockwise). Parameter `speed` is optional; if not given, default speed of 50 (i.e. half of maximal) is used. Note that this fucntion relies on Inertial Motion Unit (IMU) for operation, so you need to calibrate IMU at least once prior to using it. See Section on IMU later.

3.3.2 Driving using heading

If you need to make repeated turns, the errors at each turn add up, so at the end we might get a significant course deviation. To help combat that, you can use the following modification of drive and turn commands:

turn_to(heading, direction, speed=60)

Turn to a given heading (yaw angle), in degrees. Parameter *heading* can be one of two predefined values: either *CW* (clockwise) or *CCW* (counterclockwise). As before, parameter `speed` is optional. Below is an example:

```
# bad: accumulating errors
bot.turn(90)
bot.go_forward(50)
bot.turn(-90)
bot.go_forward(50)
```

3.3.3 Low level commands

You can also control robot motors directly:

set_motors(power_L, power_R)

Set power for left and right motors. `power_L` is power to left motor, `power_R` is power to right motor. Each of them should be between 100 (full speed forward) and -100 (full speed backward).

Note that because no two motors are exactly identical, even if you give both motors same power (e.g. `set_motors(60,60)`), their speeds might be slightly different, causing the robot to veer to one side instead of moving straight. To avoid this, use `go_forward()` command described above.

stop_motors()

Stop both motors.

3.3.4 Encoders

Both motors are equipped with encoders (essentially, rotation counters). For 75:1 HP motors, each motor at full speed produces about 4200 encoder ticks per second.

reset_encoders()

Resets (sets to zero) both encoders. Note that encoders are also reset by commands *go_forward()*, *go_backward()*, *turn()*.

get_encoders()

Gets values of both encoders and saves them. These values can be accessed as described below

encoder_L

encoder_R

Value of left and right encoders, in ticks, as fetched at last call of `get_encoders()`. Note that these values are not automatically updated: you need to call `get_encoders()` to update them

distance_traveled()

Returns the distance traveled by the robot since the last encoder reset. It can be very useful in combination with *go_forward(UNLIMITED)*, e.g.

```
bot.go_forward(UNLIMITED, 50) #start moving forward a 50% speed
while (bot.all_on_black() and bot.distance_traveled() < 20):
   pass
# stop once we have traveled 20 cm or one of reflectacne sensors sees white, whatever␣
↪comes first
bot.stop_motors()
```

get_speeds()

Gets the speeds of both motors and saves them. These values can be accessed as described below

speed_L

speed_R

Speed of left and right motors, in ticks/second, as fetched at last call of `get_speeds()`. Note that these values are not automatically updated: you need to call `get_speeds()` to update them

3.3.5 PID

FIXME

PID is an abbreviation for Proportional-Integral-Differential control. This is the industry standard way of using feedback (in this case, encoder values) to maintain some parameter (in this case, motor speed) as close as possible to target value.

Yozh bot has PID control built-in; however, it is not enabled by default. To enable/disable PID, use the functions below.

Before enabling PID, you need to provide some information necessary for its proper operation. At the very minimum, you need to provide the speed of the motors when running at maximal power. For 75:1 motors, it is about 4200 ticks/second; for other motors, you can find it by running `motors_test.py` example.

`configure_PID(maxspeed)`

Configures parameters of PID algorithm, using motors maximal speed in encoder ticks/second.

`PID_on()`

`PID_off()`

Enables/disables PID control (for both motors).

Once PID is enabled, you can use same functions as before (`set_motors()`, `stop_motors()`) to control the motors, but now these functions will use encoder feedback to maintain desired motor speed.

3.4 Servos

Yozh has two ports for connecting servos. To control them, use the commands below.

`set_servo1(position)`

`set_servo2(position)`

Sets servo 1/servo 2 to given position. Position ranges between 0 and 1; value of 0.5 corresponds to middle (neutral) position.

Note that these commands expect that the servo is capable of accepting pulsewidths from 500 to 2500 microseconds. Many servos use smaller range; for example, HiTec servos have range of 900 to 2100 microseconds. For such a servo, it will reach maximal turning angle for position value less than one (e.g., for HiTec servo, this value will be 0.8); increasing position value from 0.8 to 1 will have no effect. Similarly, minimal angle will be achieved for `position = 0.2`.

Warning: please remember that if a servo is unable to reach the set position because of some mechanical obstacle (e.g., grabber claws can not fully close because there is an object between them), it will keep trying, drawing significant current. This can lead to servo motor overheating quickly; it can also lead to voltage drop of Yozh battery, interfering with operation of motors or other electronics. Thus, it is best to avoid such situtations.

3.5 Reflectance sensor array

Yozh has a built-in array of reflectance sensors, pointed down. These sensors can be used to detect field borders, for following the line, and other similar tasks.

3.5.1 Basic usage

`linearray_on()`

`linearray_off()`

Turns reflectance array on/off. By default, it is off (to save power).

`linearray_raw(i)`

Returns raw reading of sensor `i` (i = 0...6). Readings range 0-1023 depending on amount of reflected light: the more light reflected, the **lower** the value. Typical reading on white paper is about 50, and on black painted plywood, 850. Note that black surfaces can be unexpectedly reflective; on some materials which look black to human eye, the reading can be as low as 400.

3.5.2 Calibration

Process of calibration refers to learning the values corresponding to black areas of the field and then using these values to rescale the raw readings. (We do not calibrate white readings, as they do not vary that much).

`calibrate()`

Calibrates the sensors, recording the black values. This command should be called when all of the sensors are on the black area of the field.

`linearray_cal(i)`

Returns reading of sensor `i`, rescaled to 0-100: white corresponds to 0 and black to 100. It uses the calibration data, so should only be used after the sensor array has been calibrated.

`sensor_on_white(i)`

Returns `True` if sensor `i` is on white and false otherwise. A sensor is considered to be on white if calibrated value is below 50.

`sensor_on_black(i)`

Returns `True` if sensor `i` is on black and false otherwise.

`all_on_white()`

`all_on_black()`

Returns `True` if all sensors are on black (respectively, white) and false otherwise.

3.5.3 Line following

A common task for such robots is following the line. To help with that, Yozh library provides the helper function.

`line_position_white()`

Returns a number showing position of the line under the robot, assuming white line on black background. The number ranges between -4 (line far to the left of the robot) to 4 (line far to the right of the robot). 0 is central position: line is exactly under the center of the robot.

Slightly simplifying, this command works by counting how many sensors are to the left of the line, how many are to the right, and then taking the difference. It works best for lines of width 1-2cm; in particular, electric tape or gaffers tape (1/2" or 3/4") works well.

This command only uses the central 5 sensors; rightmost and leftmost sensor (0 and 6) are not used.

If there is no line under these sensors, the function returns `None`. Thus, before using the returned value in computations, you must test whether it is `None`.

`line_position_black()`
: Same as above, but assuming black line on white background.

3.6 Distance sensors

The robot is equipped with two front-facing distance sensors, using Time-of-Flight laser technology, which can be accessed using the commands below.

`distance_L.range`

`distance_R.range`
: Distance reading of left (respectively, right) sensor, in mm. Note that it is a property, not a function - do not use parentheses.

3.7 Inertial Motion Unit

This section describes the functions for using the built-in Inertial Motion Unit (IMU).

Yozh contains a built-in Inertial Motion Unit (IMU), which is based on ICM42688 chip from TDK Invensense. This chip combines a 3-axis accelerometer and a 3-axis gyro sensor, which provide information about acceleration and rotational speed. Yozh firmware processes the sensor data to provide information about robot's orientation in space, in the form of Yaw, Pitch, and Roll angles. (Yozh firmware is based on the work of Kris Winer and uses data fusion algorithm invented by Sebastian Madgwick.)

Below is the description of functions related to IMU. You can also check sample code in *imu_test* example sketch included with Yozh CircuitPython library.

3.7.1 Initialization

By default, the IMU is active. To stop/restart it, use the functions below.

`IMU_stop()`
: Stop the IMU

`IMU_start()`
: Restart the IMU

`IMU_status()`
: Returns IMU status. This function can be used to verify that IMU activation was successful. Possible values are:

  - 0: IMU is inactive
  - 1: IMU is active
  - 2: IMU is currently in the process of calibration

3.7.2 Calibration

Before use, the IMU needs to be calibrated. The calibration process determines and then applies corrections (offsets) to the raw data; without these corrections, the data returned by the sensor is very inaccurate.

If you haven't calibrated the sensor before (or want to recalibrate it), use the following function:

IMU_calibrate()

This function will determine and apply the corrections; it will also save these corrections in the flash storage of the Yozh secondary microcontroller, where they will be stored for future use. This data is preserved even after you power off the robot (much like the usual USB flash drive).

This function will take about 10 seconds to execute; during this time, the robot must be completely stationary on a flat horizontal surface.

If you had previously calibrated the sensor, you do not need to repeat the calibration process - by default, upon initialization the IMU loads previously saved calibration values.

Note that the IMU is somewhat sensitive to temperature changes, so if the temperature changes (e.g., you moved your robot from indoors to the street for testing), it is advised that you recalibrate the IMU.

3.7.3 Reading Values

Yozh allows you to read both the raw data (accelerometer and gyro readings) and computed orientation, using the following functions:

void **IMU_get_accel()**

Fetches from the sensor raw acceleration data and saves it using member variables `ax`, `ay`, `az`, which give the acceleration in x-, y-, and z- directions respectively in in units of 1g (9.81 m/sec^2) as floats.

void **IMU_get_gyro()**

Fetches from the sensor raw gyro data and saves it using member variables `gx`, `gy`, `gz`, which give the angular rotation velocity around x-, y-, and z- axes respectively, in degree/s (as floats).

float **IMU_yaw()**

float **IMU_pitch()**

float **IMU_roll()**

These functions return yaw, pitch, and roll angles for the robot, in degrees. These three angles determine the robot orientation as described below:

- yaw is the rotation around the vertical axis (positive angle corresponds to clockwise rotation, i.e. right turns). Note that zero value is rather random (it is not the starting position of the robot!)
- pitch is the rotation around the horizontal line, running from left to right. Positive pitch angle corresponds to raising the front of the robot and lowering the back
- roll is the rotation around the horizontal line running from front to back. Positive roll angle corresponds to raising the left side of the robot and lowering the right.

For more information about yaw, pitch, and roll angles, please visit https://en.wikipedia.org/wiki/Aircraft_principal_axes

normalize(angle)

A helper function; adds or subtracts 360 to the angle as needed to bring it to the range

[-180,180]. Useful for computing difference of headings, e.g.

```
start_yaw = bot.IMU_yaw()
# some driving instructiosn here
angle_turned = bot.normalize(bot.IMU_yaw()-start_yaw) # angle will be between -180 and␣
↪180
```

CHAPTER

FOUR

PROJECTS

In this chapter, we list several simple projects that can be done using Yozh.

4.1 Stay inside the field

We begin with a very simple project: staying in the field. Here, we assume that we have a black field (such as black painted plywood) with boundary marked by white tape. The goal is to program the robot to stay within the field boundaries.

First attempt (in pseudocode, not including the intialization):

```
go forward until robot sees white boundary
turn around
```

To see the boundary, we use reflectance sensor array, namely function *all_on_black()*: if this function returns *False*, at least one of the sensors sees the white boundary. We also replace "go forward until…" by more common *while* loop:

```
bot.set_motors(30,30)
while bot.all_on_black():
    pass
#if we are here, it means at least one of sensors sees white
bot.stop_motors()
bot.turn(180)
```

Note that there is no need to set motor speed inside *while bot.all_on_black()* loop: the motors are already running and will continue doing so until you explicitly stop them . `

Finally, we enclose it in *while True* loop to make it repeat forever:

```
while True:
    bot.set_motors(30,30)
    while bot.all_on_black():
        pass
    #if we are here, it means at least one of sensors sees white
    bot.stop_motors()
    bot.turn(180)
```

This is far from optimal. For example, if it is the right sensor that sees the boundary, it makes sense to turn left rather than turn 180 degrees:

```
while True:
    bot.set_motors(30,30)
    while bot.all_on_black:
        pass
    #if we are here, it means at least one of sensors sees white
    if bot.sensor_on_white(bot.A1):
        turn(-120)
    else:
        turn(120)
```

4.2 Line follower

In this chapter, we program the robot to follow a line on the floor. We will make a line by putting 1/2-inch wide white gaffers tape on a black surface (a sheet of plywood painted black). You can make your own field; just make sure the line is at least half inch wide and doesn't have sharp turns.

Before we start writing code, we need to describe the algorithm the robot will be using - first in human language, then translate it to Python.

The obvious algorithm is "start on the line; go forward until you get off the line; turn to get back on the line; repeat".

However, this algorithm will result in very jerky movement: the robot will only start correcting its course when it gets completely off the line. Since we have a whole array of front line sensors, we can use them to detect even small deviation from the right course - when the robot is still on the line, but the line is not exactly under the center of the robot - and start correcting before we get off the line. Yozh library provides a function that allows one to determine the position of the line relative to the center of the robot: *line_position_white()*, which returns values ranging from -5 to 5.

To correct, we would be going forward but steering more to the left or right as needed: if the line is to the left of the robot center, we must be steering left; if the line is to the right, we must be steering right.

This leads to the following algorithm

```
while True:
    get the line position
    go forward steering left or right as needed to correct the position
```

Note that here we are continuously correcting our steering using the sensor feedback. To translate this algorithm to an actual program, we need to explain how one steers left or right. This is easy: to have the robot steer to the right, we need left motor to have more power than the right. Thus, instead of having both motors running at 50%, we could use

setMotors(50+correction, 50-correction).

It makes sense to have the parameter *correction* **proportional** to the difference between the actual line position and the desired one: the farther off we are, the more we need to turn.

This gives the following program

```
Kp = 9
while True:
    error = bot.line_position_white()
    bot.set_motors(50+Kp*error,50-Kp*error)
```

Double-check the sign: if *error* is negative (line to the left), we need to be steering left, so the left motor should have less power than the right; if *error* is positive, we will be steering right.

The value of the coefficient *Kp=9* was chosen so that when the line is all the way to one side (error= -5), the motors will be given power 50+45=95, 50-45=5

You can test what happens if Kp=9 is replaced by another value. If the value is too large, the robot will turn very quickly even for small errors, which can lead to the robot spending most time turning left and right, with very little headway. If the value is too small, the robot will be turning very little, which can cause it to miss a sharp turn. You can experiment to find the best value.

The same idea of correcting the course using sensor feedback, with the correction proportional to the error, can be used in many other situations. Instead of following the line, we could use it to turn to face an obstacle (using front proximity sensors), or face up on an inclined surface, or many other similar situations.

The code above still has one problem. Namely, when we reach the end of the line, function *line_posiiton_white()* will return *None*, which will cause an error in the next line: you can't use *None* in an arithmetic expression. Thus, we need an extra check to catch that.

A natural idea would be to replace *while True* by *while error is not None*:

```
Kp = 9
while bot.line_position_white() is not None:
    error = bot.line_position_white()
    bot.set_motors(50+Kp*error,50-Kp*error)
```

This, however, is not enough - do you see why?

Better version is using `break` command of Python:

```
Kp = 9
while True:
    error = bot.line_position_white()
    if error is None:
        break
    bot.set_motors(50+Kp*error,50-Kp*error)
bot.stop_motors()
```

As before, you also need to include the code for initialization and sensor calibration.

4.3 Maze runner: wall following

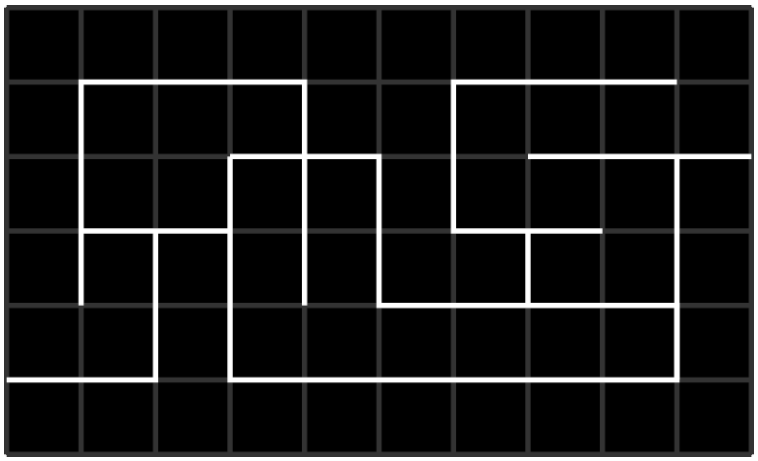
In this challenge, we will teach the robot find its way out of a maze. The maze is made of approx. 3x5 ft sheet of plywood, painted black. White masking tape (3/4 inch wide) is used to mark passages forming the maze; these lines follow rectangular grid with 0.5 ft squares.

Finding a way out of a maze is a classic problem, and there is a number of algorithms for doing that. The simplest of them is the wall following rule.

Start following passages, and whenever you reach a junction always follow the leftmost open passage. This is equivalent to a human walking in the a maze by putting his hand on the left wall and keeping it on the wall as he walks through.

This method is guaranteed to find an exit if we start at the entrance to the maze; then this method allows us to explore a section of the maze and find our way out. However, it is not guaranteed to find an exit if we start in the middle of the maze: the robot could be going in circles around an "island" inside the maze.

The first draft of the program looks as follows (not including initialization and setup):



```
while True:
    go_to_intersection()
    check_intersection()
    if there is a passage to the left, turn left
    otherwise, if there is a passage forward, go forward
    otherwise, turn right
```

Function *go_to_intersection()* should follow the line until we reach an intersection (that is, until the reflectance sensors at the front of the robot are above an intersection). This function is very similar to line follower algorithm from the previous project, with added checks: it should stop when reflectance sensor A1 (rightmost) or A8 (leftmost) sees white.

Function *check_intersection()* should do three things:

1. Slowly advance forward until the center (not front!) of the robot is above the intersection.
2. While doing this, keep checking whether there is a passage to the left and record it somehow; same for passage to the right
3. once we advanced so that the center of the robot is above the intersection, also check if there is a passage forward.

We can achieve this by asking the robot to start moving forward until we have travelled 5 cm; while doing this, we will be checking the line sensors. If the leftmost line sensor (number 6) sees white, it means that there is a passage to the left. To record it, we can create boolean variable *path_left* and set it to *True* once the sensor 6 sees white (Also, we should remember to set it to *False* initially):

```
def check_intersection():
    # go forward while checking for intersection lines
    bot.reset_encoders()
    path_left = False

    bot.set_motors(30,30) #start moving forward slowly
    while bot.get_distance()<5:
        if bot.sensor_on_white(6):
            path_left = True
    bot.stop_motors()
```

We should also add similar code for determining whether there is a path to the right (left to the reader as an exercise).

Next, once we advanced, we need to check if there is a passage ahead. This is easy using *all_on_black()* function (if there is no passage forward, all sensors will be on black).

Finally, we need somehow to return this information to whatever place in our program called this function. If we needed to return one value, we could just say *return(path_left)*, but here we need to return 3 boolean values: *path_left*, *path_forward*, *path_right*. One way to do that is to put them in a list and return the list. This gives the following code:

```
def check_intersection():
    # go forward while checking for intersection lines
    bot.reset_encoders()
    path_left = False
    path_forward = False
    path_right  = False

    bot.set_motors(30,30) #start moving forward slowly
    while bot.get_distance()<5:
        if bot.sensor_on_white(6):
            path_left = True
        ....
    bot.stop_motors()
    if not bot.all_on_black():
        path.forward = True
    # now, let us return the found values
    return([path_left, path_forward, path_right])
```

Now we can write the main program:

```
while True:
    go_to_intersection()
    paths = check_intersection()
    if paths[0]:
       # path to the left is open
       bot.turn(-90)
   elif paths[1]:
       # path forward is open - do nothing, no need to turn
       pass
   elif paths[2]:
       bot.turn(90)
```

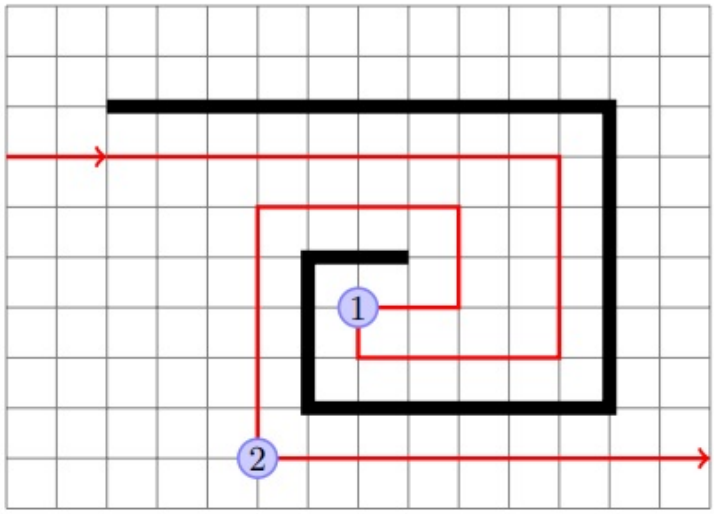
4.4 Maze runner: pledge algorithm

This is a modified version of wall following that's able to jump between islands, to solve mazes that wall following cannot. It's a guaranteed way to reach an exit on the outer edge of any 2D maze from any point in the middle. However, it is not guaranteed to visit every passage inside the maze, so this algorithm will not help you if you are looking for a hidden treasure inside the maze.

Start by picking a direction, and always move in that direction when possible. When you hit a wall, start wall following, using the left hand rule. When wall following, count the number of turns you make, a left turn is -1 and a right turn is 1. Continue wall following until your chosen direction is available again and the total number of turns you've made is 0; then stop following the wall and go in the chosen direction until you hit a wall. Repeat until you find an exit.

Note: if your chosen direction is available but the total number of turns is not zero (i.e. if you've turned around 360 degrees or more), keep wall following until you untwist yourself. Note that Pledge algorithm may make you visit a passage or the start more than once, although subsequent times will always be with different turn totals.

In the figure above, thick black lines show the walls of the maze; the red line shows the path of the robot. At point 1, the robot turns so that it is again heading the same direction as in the beginning; however, the number of turns at



this point is not zero, so the robot continues following the wall. At point 2, the robot is again heading in the original direction, and the number of turns is zero, so it stops following the wall. Had the robot left the wall at point 1, it would be running in circles.

To program the Pledge algorithm, we need to keep track of robot direction and number of turns. In fact, just the number of turns is sufficient: if we know the number of turns, we can determine the direction. Thus, we introduce a global variable `numTurns`. Every time we turn 90 degrees clockwise, `numTurns` is increased by 1; every time we turn 90 degrees counterclockwise, we decrease `numTurns` by 1.

Thus, the draft of the program would be

```
numTurns = 0
def loop():
    goToWall()
    followWall()
```

where

- Function `gotoWall()` goes forward along the line, through intersections, until the robot hits a wall
- Function `followWall()` follows the wall using left hand rule until we are again facing the same direction as before, with numTurns=0.

For each of these functions, we need to describe carefully what conditions the function expects at the start and in what condition it leaves the robot at the end (which way is it facing? is it at intersection?).

goToWall():

- Initial condition: robot is on the line (i.e., the line is under the center of the front sensor array; robot could be at intersection), `numTurns=0`
- Final state: robot is at an intersection, there is a wall ahead (i.e., no passage forward), and `numTurns=0`

followWall():

- Initial condition: robot is at an intersection, there is a wall ahead (i.e., no passage forward), and `numTurns=0`
- Final state: robot is on the line (i.e., the line is under the sensor of the front sensor array; robot could be at intersection), `numTurns=0`

When we think about implementing the algorithm, we see that in the very beginning of `followWall()`, the robot needs to turn so that the wall is on its left. Normally it would be just a 90 degree right turn; however, if we are at a dead end,

we need to turn 180 degrees. Thus, we need to know whether there is a passage to the right. Therefore, we add one more condition to the final state of `goToWall()`:

- Final state: robot is at the intersection, there is a wall ahead (i.e., no passage forward), `numTurns=0`, and global variable `passageRight` contains information about whether there is a passage to the right.

To implement these two functions, we will make use of the functions `goToIntersection()`, `checkIntersection()` which we used for the wall-following algorithm. Implementing `goToWall()` is trivial.

For followWall(), in the beginning we must put

```
if passageRight:
    turn(90)
    numTurns += 1
else:
    # no passage to the right - need to turn 180
    turn(180)
    numTurns += 2
```

After this, we do the regular line following algorithm: go to intersection, check intersection, turn as needed, except that we should exit the function if, after a "turn as needed", we have `numTurns=0`. We leave it to you to complete the algorithm.

INDEX

I