Yozh Robot

Alexander Kirillov

Aug 22, 2023

TABLE OF CONTENTS

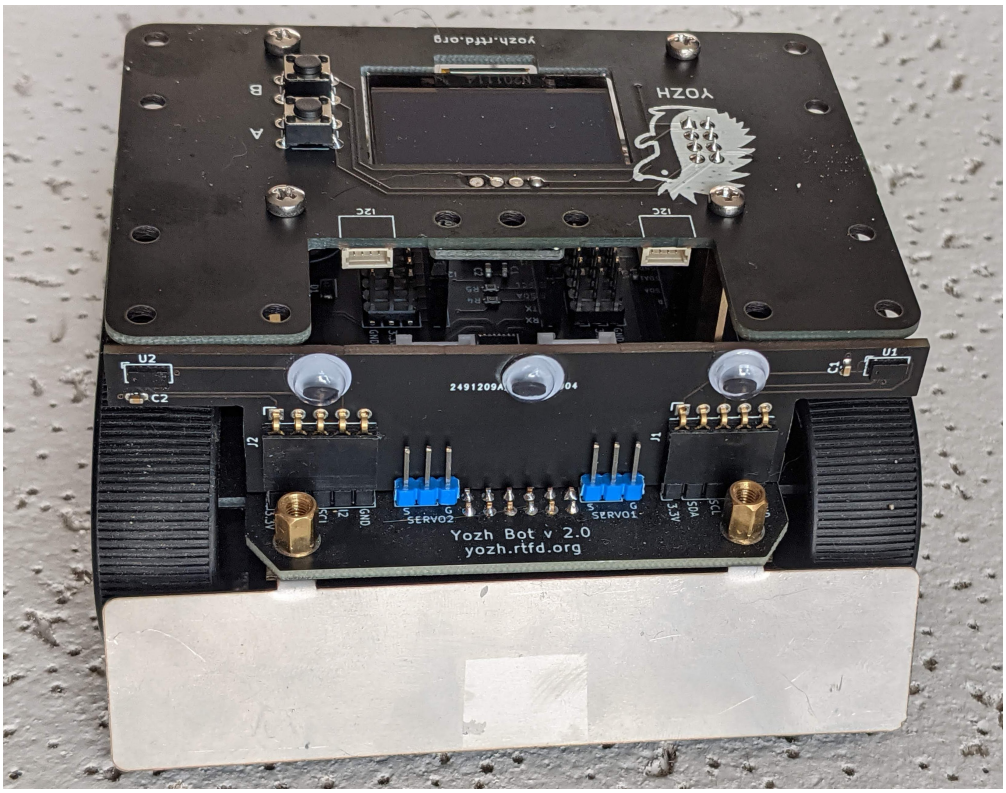
1 Quick start guide 3

2 Python Primer 9

3 Projects 13

4 Yozh Library Reference 21

Index 29



Yozh is a small (under 10cm*10cm) robot, based on Pololu's Zumo chassis. It was created by shurik179 for a robotics class at SigmaCamp. Below are the key features of this robot.

The robot consists of the following components:

- Zumo chassis by Pololu
- Power source: 4 AA batteries (NiMH rechargeable batteries recommended)
- Two micro metal gearmotors by Pololu (6V, HP, 75 gear ratio), with motor encoders
- Custom electronics board, containing a slave 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
- ItsyBitsy RP2040 by Adafruit, which serves as robot brain. It plugs into the main board and is programmed by the user in CircuitPython, using a provided CircuitPython library to communicate with the slave MCU over I2C. This library provides high-level commands such as *move forward by 30cm*
- Included sensors and electronics:
  - Top plate with 128*64 **OLED display** and 2 **buttons** for user interaction
  - Bottom-facing **reflectance array** with 8 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
  - Two ports for connecting **servos**

- There are plenty of pins available for connection additional electronics. We also provide several standard connectors for users convenience (Qwiic/Stemma QT connector for I2C devices, Grove connectors)
- 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.

CHAPTER

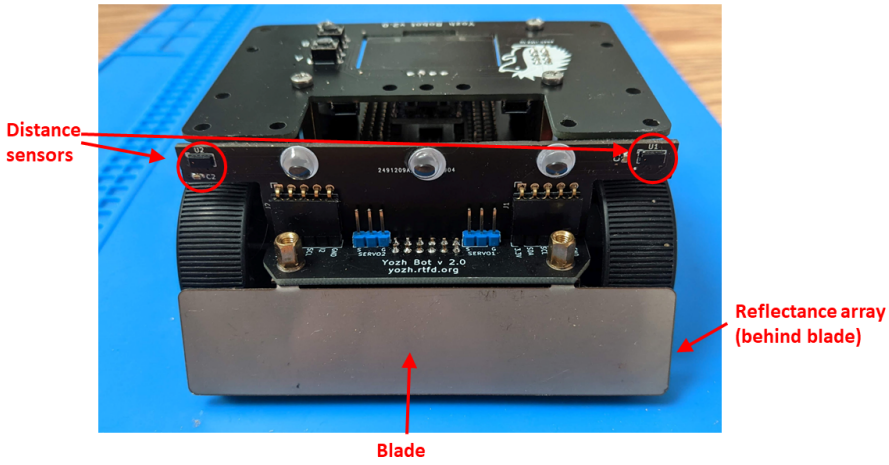
ONE

QUICK START GUIDE

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:

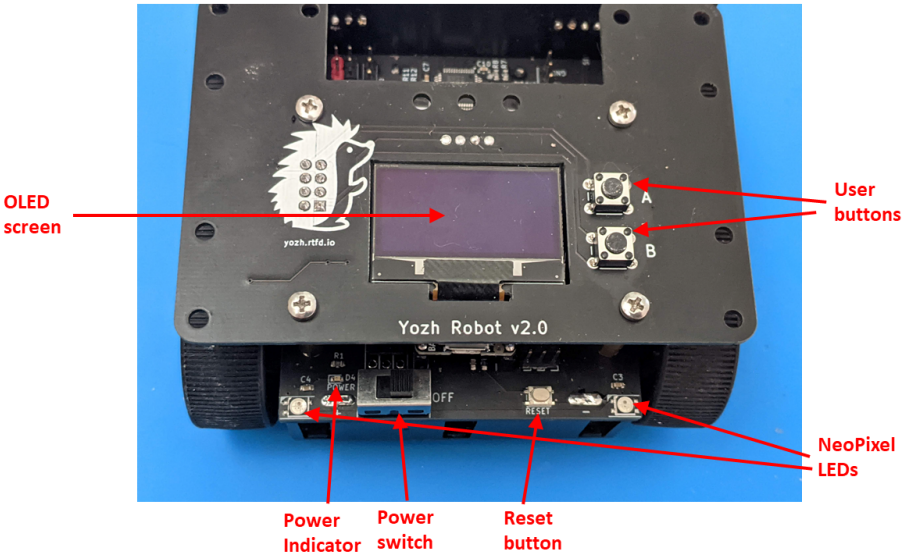


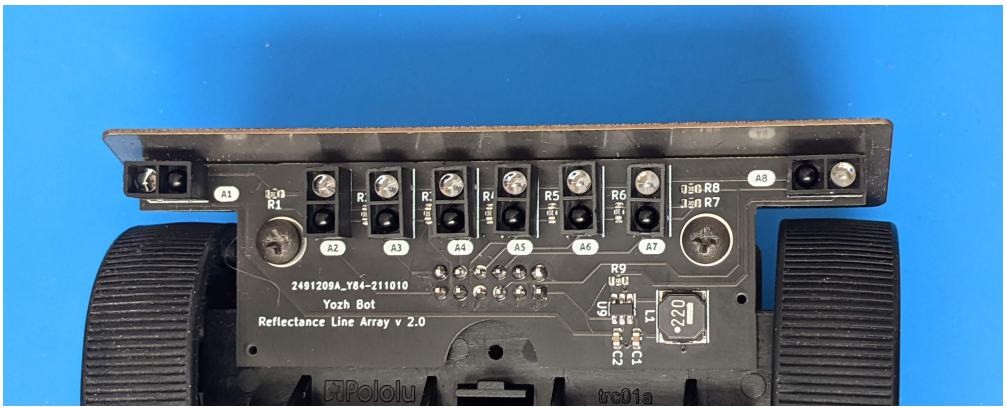
1.2 Micro Python library installation

Yozh is intended to be programmed in MicroPython - an implementation of Python programming language for micro-controllers. For general background on MicroPython, please visit MicroPython website .

Before you can start programming Yozh, you need to do the following:

1. Install MicroPython firmware on ItsyBitsy board, as described here (you will need to remove the top plate to access the BOOTSEL button of ItsyBitsy).
2. Use Thonny editor (see next section) to copy file *yozh.py* (found in this repository) to ItsyBitsy board. This file contains Yozh MicroPython library.
3. Again using Thonny, cerate folder *lib* on the ItsyBitsy board and copy there files *ssd1306.py* and *vl53l0x.py*

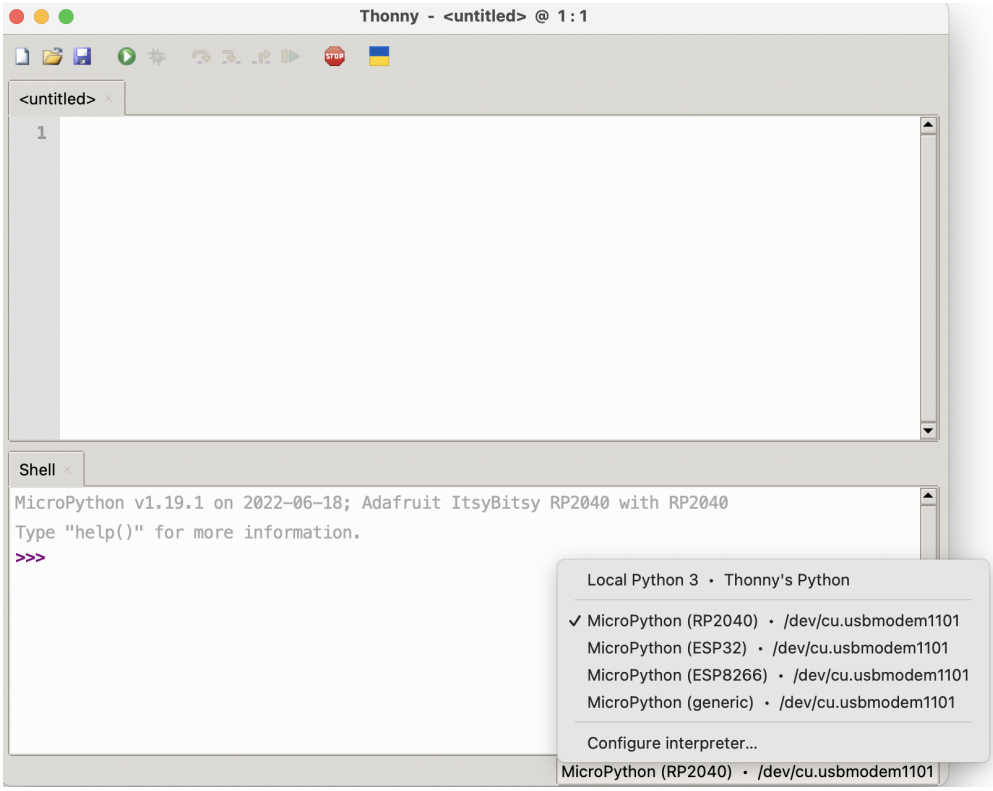

OLED
screen
User
buttons
A
B
yozh.rtfd.io
Yozh Robot v2.0
OFF
RESET
NeoPixel
LEDs
Power
Indicator
Power
switch
Reset
button


2491209A_Y84-211010
Yozh Bot
Reflectance Line Array v 2.0
Pololu

1.3 Thonny editor

We suggest using Thonny editor for creating and editing programs for your robot. Please follow the instructions on their website to install Thonny editor on your computer.

o verify your installation, connect Yozh to the computer using a USB cable (using the USB connector of ItsyBitsy board) and then start Thonny editor. Select *MicroPython (RP2040)* in the lower right corner of the screen. Tab *Shell* should show version of MicroPython installed on the ItsyBitsy board, similar to what is shown below.



1.4 First program

To create your first program, start Thonny. Create a new file (using *File->New* menu item) and copy-paste in the file the following lines:

```
from yozh import Yozh
GREEN=[0,255,0]
bot = Yozh()

bot.begin()

# buzz at frequency 440Hz for 1 sec
bot.buzz(440,1.0)
# set leds to green
bot.set_leds(GREEN)
```

Now, use *File->save as* menu item to save this file. When prompted, choose *RP2040 device* and enter file name *main.py* (this is important!)

Click green arrow at the top of Thonny window to run this file now. You should see the following message in Thonny window:

and the robot OLED display should show battery voltage and firmware version.

You can disconnect the robot from the computer and restart it by turning it off and on. Upon restart, it will automatically run the code in *main.py*.

You can now experiment with the program, modifying it any way you like. Here are some useful tips:

- You can have many programs uploaded to the robot, but by default, upon restart it will execute file with name *main.py*. Thus, it makes sense to always use this file for you program.
- If your program is running, or if you disconnected and reconnected your robot to the computer, you need to hit *STOP* icon to stop the program before you can save a new version of the program
- when copying and pasting, make sure that the indentation is correct!
- When ending your work session, it is highly recommended to save a copy of the program to the computer!

1.5 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 (Shell), so you can see these messages.

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/adafruit-itsybitsy-rp2040/the-repl for more info.

1.6 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 July 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
- *servo_test.py* - testing servos (if you have any attached).
- *drive_test.py* - testing higher-level drive commands, such as *go forward for 10cm* or *turn 90 degrees*
- *distance_test.py* - testing operation of front-facing distance sensors
- *linearray_test.py* - testing reflectance sensor array

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

1.7 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 shurik179@gmail.com.

CHAPTER

TWO

PYTHON PRIMER

These commands are not directly related to Yozh, but can serve as useful cheat sheet for people new to MicroPython.

For more information about MicroPython on RP2040 microcontrollers, see MicroPython website.

Remember that in Python, indentation is important!

2.1 Comments

Single line comments are indicated by #: anything after # till the end of the line is ignored by the computer. Multiline comments are indicated by three double quote symbols:

```
"""
This is a
multiline comment
"""
```

2.2 Variables

In Python, you do not need to declare a variable; you can just start using it:

```
x = 20.5 #float (decimal) number

address = 0x29 #hexadecimal number

text = "Welcome" #string
```

As any computer language, Python also has logic (boolean) variables. It has two predefined logic constants: *True*, *False*

You can use common logic operations: *and*, *or*, *not*. Note that unlike many other languages, Python doesn't use && and || for logic operations

Comparison: to compare if two values are equal, use == operator:

```
if x==5 and y>0:
    ...
```

Warning: Writing *if x=5* would give a very different result (single equality is assignment, not comparison) - this is one of the most common beginner mistakes in any programming language

2.3 Lists

Lists in Python are analogs of arrays in other languages. To define a list, use

```
list = ["A", "B", "C"]
```

To access i-th item in the list, use *list[i]*. Note that in a list of N elements, index ranges from 0 to N-1.

To find current length of a list, use function *len(list)*.

To append one item the list at the end, use *append()* function: *list.append("D")*

To join (concatenate) two lists, you can use +:

```
list=["A", "B", "C"]+["X", "Y", "Z"]
```

There are other list-related functions - check Python docs.

2.4 Python control structures

2.4.1 Conditional

```
if condition:
   some operators
else:
   other operators
```

else: part is optional. Note that there is no need to enclose condition in parentheses (but no harm if you do it anyway). If you need more options, use the form below; *elif* is short for *else if*

```
if condition:
   some operators
elif:
   some more operators
else:
   other operators
```

2.4.2 Loops

- Common while loop:

```
while condition:
   operators
```

- For loop: repeat for every value of *i* from the list.

```
for i in list:
  operators
  ...
  operators
```

A list can be defined explicitly, e.g. list = ["A", "B", "C"]. More commonly, if you want the loop to be repeated N times, for all values of i from 0 to N-1, you use *for i in range(N)*

2.5 Functions

You can define your own function and call it later:

```
def factorial(n):
    result = 1
    for i in range (n):
        result = result * (i+1)
    return(result)

print("20!={}".format(factorial(20)))
```

Note: a function must be defined **before** it is called. Also, please note that a function can not access variables defined outside of the function; if you need this, read about global variables in Python docs.

2.6 Printing

To print a message to standard output (for programs running on the robot, it would be Shell tab of Thonny editor), use *print()* function:

```
print("Hello, world!")
```

The argument can be a string, a variable, or any other expression. You can also provide several arguments separated by commas: *print(x,y,z)*.

By default, every print command will also print a newline at the end, moving to the next line in the output. To suppress it, use *end* parameter: *print("Hello, world!", end = "")*

(in this case, end parameter is the empty string)

To print a message containing some numerical values (or other variable types), insert in your message placeholders {} in the places where numerical values would go, and then use *format* function as follows:

```
message = "Acceleration: x={} y={} z={}"
print(message.format(a_x, a_y, a_z))
```

It is also possible to format the numbers, specifying how many decimal places you want printed; refer to Python documentation for details.

2.7 Time control

The commands below are defined in *time* module. Thus, to use them you must put *include time* in your Python file.

To pause the execution of the program for given time, use

```
time.sleep(time_in_seconds)
```

To time various events, you can use the *time.ticks_ms()* milliseond counter (this is specific to RP2040 microcontoller):

```
t0 = time.ticks_ms()
...
t1=time.ticks_ms()
time_interval = t1-t0 #duration in milliseconds
```

2.8 Miscellaneous

Python has a special name for non-existing (undefined) values, *None*. Thus, to test if a variable has been defined, you can use

```
if x is None:
   ...
```

(for technical reasons, you can't use if *x*==*None*).

Note that it is different from value 0 or empty string. *None* means that the variable has not been defined yet, which is different from being defined and given 0 value.

Also, Python has a special function that does nothing, named *pass*:

```
while (bot.sensor_on_white(bot.A1)):
   pass
```

This is commonly used as a placeholder to be replaced later by actual commands.

CHAPTER

THREE

PROJECTS

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

3.1 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)
```

3.2 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?

Here is a corrected version:

```
Kp = 9
error = 0
while error is not None:
    bot.set_motors(50+Kp*error,50-Kp*error)
    error = bot.line_position_white()
bot.stop_motors()
```

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

3.3 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.

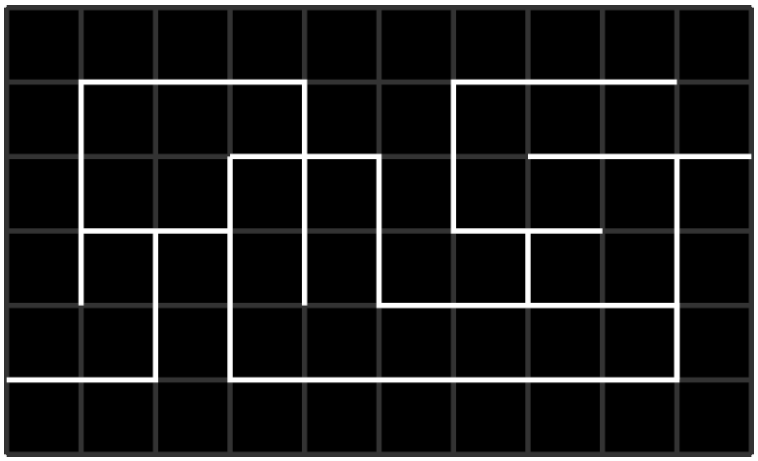
Start following passages, and whenever you reach a junction always follow the leftmost open passage. This is equivalent to a human walking in the a maze by putting his hand on the left wall and keeping it on the wall as he walks through.

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()
```

(continues on next page)



(continued from previous page)

```
    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 (A8) 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 A8 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(bot.A8):
            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(bot.A8):
            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)
```

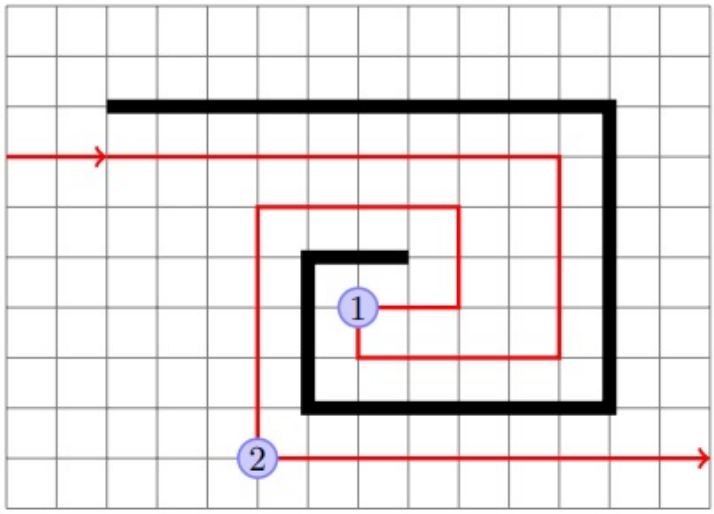
3.4 Maze runner: pledge algorithm

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.

CHAPTER

FOUR

YOZH LIBRARY REFERENCE

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

This document describes version 2.0 of the library. It is intended to be used with MicroPython 1.20 or later.

4.1 Initialization and general functions

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

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

This creates 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()`.

By default, creating bot object also initializes the OLED display; it will produce errors if the OLED display is not found. If for some reason you are not using OLED, you can initialize the robot using this form of initialization command:

```
from yozh import Yozh
bot = Yozh(oled = None)
```

Here are some basic functions:

begin()

Shows basic info (firmware version, battery voltage) on OLED screen

fw_version()

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

battery()

Returns battery voltage, in volts. For normal operation it should be at least 4.5 V.

4.2 Display, buttons, LEDs

Yozh contains a buzzer, two NeoPixel LEDs in the back and an 128x64 OLED screen and two buttons on the top plate, for interaction with the user. To control them, use the functions below.

4.2.1 LEDs

set_led_L(color)

set_led_R(color)

These commands set the left (respectively, right) LED to given color. Color must be a list 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. You can also define named colors for easier use, e.g.

```
BLUE=[0,0,255]

bot.set_led_L(BLUE)
```

set_leds(color_l, color_r)

Set colors of both LEDs at the same time. As before, each color is a list of three values. Parameter `color_r` is optional; if omitted, both LEDs will be set to the same color.

set_led_brightness(value)

Set the maximal brightness of both LEDs to a given value (ranging 0-255). Default value is 64 (i.e., 1/4 of maximal brightness), and it is more than adequate for most purposes, so there is rarely a need to change it. Setting brightness to 255 would produce light bright enough to hurt your eyes (and drain the batteries rather quickly)

4.2.2 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.

4.2.3 Buttons

wait_for(button)

Waits until the user presses the given button. There are two possible pre-defined buttons: `bot.button_A` and `bot.button_B`

is_pressed(button)

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

choose_button()

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

```
bot.set_text("Press any button", 0)
#wait until user presses one of buttons
if (bot.choose_button()=="A"):
```

(continues on next page)

(continued from previous page)

```
    # do something
else:
    # button B was pressed
    # do something else
```

4.2.4 OLED

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

clear_display()

Clears all text and graphics from display

set_text(text, line_number)

Prints a line of text on OLED display, on line number *line_number*, erasing all previous contents of that line. Note that line numbers start with 0, not 1! If the text to print includes line break character *n*, the line is broken and continues on next line, e.g.

```
bot.set_text("Initialized", 0)
bot.set_text("Press A to \n continue", 1)
```

4.3 Motor control

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

4.3.1 Basic control

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 fix that, use PID control as described below.

stop_motors()

Stop both motors.

4.3.2 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 both encoders

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

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

get_distance()

Returns distance (in cm) travelled by the robot since the last encoder reset.

4.3.3 PID

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.

4.3.4 Drive control

Yozh python library also provides higher level commands for controlling the robot.

go_forward (distance, speed=50)

go_backward(distance, speed=50)

Move forward/backward by given distance (in centimeters). Parameter `speed` is optional; if not given, default speed of 50 (i.e. half of maximal) is used.

Note that distance and speed should always be positive, even when moving backward.

turn(angle, speed=50)

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 all of these commands use encoder readings to determine how far to drive or turn. Of course, to do this one needs to know how to convert from centimeters or degrees to encoder ticks. This information is stored in properties `bot.CM_TO_TICKS` and `bot.DEG_TO_TICKS`. By default, Yozh library uses `CM_TO_TICKS = 150`, `DEG_TO_TICKS=14`,

which should be correct for 75:1 motors. If you find that the robot consistently turns too much (or too little), you can change these values, e.g.

```
bot.DEG_TO_TICKS=15
bot.turn(90)
```

4.4 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.

4.5 Reflectance sensor array

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

4.5.1 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`. One can use either indices 0…7 or (preferred) named values `bot.A1 = 0` … `bot.A8 = 7`. A1 is the rightmost sensor, and A8, leftmost.

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 50-80, and on black painted plywood, 950. Note that black surfaces can be unexpectedly reflective; on some materials which look black to human eye, the reading can be as low as 600.

4.5.2 Calibration

Process of calibration refers to learning the values corresponding to black and white areas of the field and then using these values to rescale the raw readings.

`calibrate()`

Calibrates the sensors, recording the lowest and highest values. This command should be called when some of the sensors are on the white area and some, on black.

`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_black()`

`all_on_white()`

Returns `True` if all 8 sensors are on black (resepctively, on white) and `False` otherwise.

4.5.3 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 -5 (line far to the left of the robot) to 5 (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 6 sensors; rightmost and leftmost sensor (A1 and A8) are not used.

If there is no line under these sensors, the function returns *None* value. Thus, before using the retunred value in any computations, check that it is not *None*.

`line_position_black()`

Same as above, but assuming black line on white background.

4.6 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.

`ping_L()`

`ping_R()`

Distance reading of left (respectively, right) sensor, in mm.

4.7 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 LSM6DSL chip from ST Microelectronics. This chip combines a 3-axis accelerometer and a 3-axis gyro sensor, which provide information about acceleration and rotational speed. The sensor is placed on the back side of the top plate. Yozh firmware combines 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.

4.7.1 Initialization

By default, the IMU is inactive. To start/stop it, use the functions below.

void **`IMU_start()`**

Activate IMU

void **`IMU_stop()`**

Stop the IMU

bool **`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

4.7.2 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:

void **`IMU_calibrate()`**

This function will determine and apply the corrections; it will also save these corrections in the flash storage of the Yozh slave 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.

4.7.3 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), relative to 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

INDEX

I

IMU_calibrate (*C function*), 27
IMU_get_accel (*C function*), 28
IMU_get_gyro (*C function*), 28
IMU_pitch (*C function*), 28
IMU_roll (*C function*), 28
IMU_start (*C function*), 27
IMU_status (*C function*), 27
IMU_stop (*C function*), 27
IMU_yaw (*C function*), 28