Ecole Polytechnique
Fédérale de Lausanne

MICRO-453: Robotics Practicals

REPORT

Programming and characterization of a modular fish robot

Francesco Maglie - 378056
Andrea Grillo - 371099

Spring 2024

Contents

1 Introduction

This report presents the outcomes of a practical exercise focused on programming and characterizing a swimming fish robot. Leveraging modules of the Salamandra robotica II and AmphiBot III robots, we embarked on implementing basic programs on the onboard microcontroller, followed by the creation of a trajectory generator for swimming motions. The subsequent phase involved assessing the robot's performance and its correlation with trajectory parameters through the utilization of a video tracking system. This practical provided us with hands-on experience in robotics programming and characterization, thereby enhancing our understanding of robotic locomotion principles.

2 Exercise 1

In the first exercise of the Practicals we had to understand a basic code dealing with the setting of the colour of the LED which is embedded in the robot module.

2.1 Used functions

The led is set by means of the function *set_rgb* which is part of the given *hardware.h* header file. The function takes 3 *uint8_t* values as input, which represent respectively the red, green and blue intensities of the LED.
The other function which is used is *pause*, which is defined in the *sysTime.h* header and takes as input a *uint32_t* of the numbers of CPU TICs to be waited. This is a blocking call, meaning that it will not return until the time to wait is finished.

2.2 Given Code

The given code has an endless loop in which it starts from a LED turned off, then increases gradually and sequentially the 3 channels of the LED, and then decreases it until shutting off the LED. Then it starts again the loop.

2.3 Modified Code

For this exercise, we had to modify the code to make the LED blink at *1Hz* in green. To achieve this, we turned on the green channel of the LED at maximum intensity, then waited for half a second, and then turned it off and waited again.
Below is the code we have used:

```
while (1) {
      set_rgb(0,255,0);
      pause(ONE_SEC / 2);
      set_rgb(0,0,0);
      pause(ONE_SEC / 2);
}
```

3 Exercise 2

For the exercise number 2, it was required to understand thoroughly the communication system between the PC and the robot.
The communication system is based on registers of different sizes. The PC is the master component of the communication, meaning that all operations are initiated from it. The robot handles the operations in a callback function, that is called leveraging the interrupt system of the microcontroller.
For this exercise, the task was to read the given programs and understand them, and try to predict the console output of the PC program.

3.1 PC side

On the PC side, the program is just setting the value of some registers and getting back the values by means of the methods of the class *CRemoteRegs*.
The *display_multibyte_register* helper function is used to display the contents of a multibyte register.
Then the *read_registers* function is used to do all the operations.

3.2 Robot side

In the robot-side code, some global variables are used as register memory to store data, and everything is handled by the *register_handler* callback function.
The callback function was quite tricky to analyze, as it does not just store the data received from the radio interface to the respective registers, but does some operations with the data that make the analysis harder.
After a thorough investigation of the function and the sequential operations, we came out with the following prediction of the output:

```
get_reg_b(6)  = 0
get_reg_b(21)  = 42
get_reg_b(21)  = 42
get_reg_b(6)  = 2
get_reg_b(6)  = 0
get_reg_mb(2) = 0 bytes:
get_reg_mb(2) = 8 bytes: 104 105 106 107 108 109 110 111
get_reg_mb(2) = 8 bytes: 11 22 106 107 108 109 110 111
get_reg_dw(2) = 2121
get_reg_dw(2) = 8128
```

The prediction was almost right, the only error that we had was in the second and third line, as the actual value is 0x42 (hexadecimal), therefore it is printed as a 66 in decimal representation.

4 Exercise 3

The purpose of this exercise is to understand how the communication between the different modules works, using the CANBUS serial communication. The CAN is a serial protocol based on a differential pair of signals, which runs across the whole robot allowing for communication on a shared bus.

4.1 Used functions

The main 2 functions to understand the code that was given are the *init_body_module* and *bus_get* functions. They are both provided by the *robot.h* header. The former takes as input the address of the module to setup and does not have any return value. The latter has as parameters the address of the module and the register from which to retrieve data and returns a *uint8_t* variable with the result of the communication.

4.2 Given code

The given code first initializes a body module, then it starts an endless loop in which the color of the LED is set depending on the position of the module that is read continuously. If the position is positive, the colour will be red, otherwise it will be blue, with an intensity proportional to the magnitude of the position. There is a "base" value that is set constant for the green channel of the LED at 32.

4.3 Modified Code

We had to modify the code to achieve the following task: read the position of each DOF of the robot and print it continuously on the PC console.

For this task, we had to integrate the radio communication seen in the previous exercise and the communication with other modules of the robot.
The main loop of the program just keeps reading the positions using the *bus_get* function and updates a global array of positions. Then there is a callback function *register_handler* which handles the sending of the position array through a multi-byte radio communication.
The main two parts of the code written are shown below:

```
// Callback function
static int8_t register_handler(uint8_t operation,
                               uint8_t address,
                               RadioData* radio_data)
{
  if(operation == ROP_READ_MB && address == 2)
  {
    radio_data->multibyte.size = NUM_POS;
    for (uint8_t i = 0; i < NUM_POS; i++)
      radio_data->multibyte.data[i] = positions[i];
    return TRUE;
  }
  return FALSE;
}

// Main loop of the program
while (1)
  {
    // Get the position of each joint
    for(uint8_t i = 0; i < NUM_POS; i++)
      positions[i] = bus_get(addresses[i], MREG_POSITION);
  }
```

5 Exercise 4

In exercise 4 the task was to implement the control of a joint of the robot. On the computer the reference signal is generated, which is then sent by means of the radio to the robot, which then is sent to the last joint of the robot using the CAN line.

5.1 Given code

An example code was given to show how to implement the remote control. The PC sends a command to set the mode of the robot and then to stop it.

5.1.1 PC side

The function used to send the control command is the method *set_reg_b*, provided by the class *CRemoteRegs* which is defined in the header *remregs.h*.
It is used to set the *REG8_MODE* register, which contains the working modality of the robot.
Below an excerpt of the code:

```
  regs.set_reg_b(REG8_MODE, 1);
  Sleep(10000);
  regs.set_reg_b(REG8_MODE, 0);
```

5.1.2 Robot side

The working modes of the robots are defined in the *modes.c* file, in which there is a function defined for each working modality.
The one given in the example code is listed below, which makes the joint move between $\pm 21^{\circ}$.

```
void motor_demo_mode()
{
  init_body_module(MOTOR_ADDR);
  start_pid(MOTOR_ADDR);
  set_color(4);
  while (reg8_table[REG8_MODE] == IMODE_MOTOR_DEMO) {
    bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(21.0));
    pause(ONE_SEC);
    bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(-21.0));
    pause(ONE_SEC);
  }
  bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
  pause(ONE_SEC);
  bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);
  set_color(2);
}
```

5.2 Modified code

For this exercise, we had again to integrate the radio communication and the communication with other modules.

5.2.1 PC side

In the code to be run on the PC, the generation of the sinusoidal reference signal has been implemented, which is then sent to the robot using the radio interface:

```
while(!kbhit()) {
    while(time_d() - actTime < DT);
    actTime = time_d();

    regs.set_reg_b(10, ENCODE_PARAM_8(40 * sin(2 * M_PI * actTime)
        ,-40,40));
}
regs.set_reg_b(REG8_MODE, 0);
```

5.2.2 Robot side

A callback has been defined to listen to changes in the setpoint coming from the radio interface:

```
static int8_t register_handler(uint8_t operation,
                               uint8_t address,
                               RadioData* radio_data)
{
  if(operation == ROP_WRITE_8 && address == 10) {
    setpoint = DECODE_PARAM_8(radio_data->byte,(-40),(40));
    return TRUE;
  }
  return FALSE;
}
```

Then the movement modality which initializes the module and the PID controller, sets continuously the setpoint of the controller and stops the PID controller at the end.

```
void motor_demo_mode()
{
  // Registers the register handler callback function
  radio_add_reg_callback(register_handler);

  init_body_module(MOTOR_ADDR);
  start_pid(MOTOR_ADDR);
  set_color(4);
  while (reg8_table[REG8_MODE] == IMODE_MOTOR_DEMO) {
    bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(setpoint))
       ;
    pause(TEN_MS);
  }
  bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
  pause(ONE_SEC);
  bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);
  set_color(2);
}
```

6 Exercise 5

For exercise 5, the objective is the same as the one in exercise 4. Make a joint move using a sinusoidal reference signal.
While in exercise 4 the reference signal was generated in the computer and sent to the robot using the radio interface, in this case, the whole calculation of the sinusoidal function is made on-board.

6.1 Given code

The given code does the calculation of the sinus functions and uses the output as the colour to be written to the RGB onboard LED.

```
do {
  // Calculates the delta_t in seconds and adds it to the current
     time
  dt = getElapsedSysTICs(cycletimer);
  cycletimer = getSysTICs();
  delta_t = (float) dt / sysTICSperSEC;
  my_time += delta_t;

  // Calculates the sine wave
  l = 32 * sin(M_TWOPI * FREQ * my_time);
  l_rounded = (int8_t) l;

  // Outputs the sine wave to the LED
  if (l >= 0) {
    set_rgb(0, l, 32);
  } else {
    set_rgb(-l, 0, 32);
  }

  // Make sure there is some delay, so that the timer output is not
     zero
```

```
    pause(ONE_MS);

  } while (reg8_table[REG8_MODE] == IMODE_SINE_DEMO);
```

6.2 Modified code

6.2.1 Exercise 5.1

For exercise 5.1 we just had to modify what is done with the sinus output, and write it to a motor controller instead of the RGB led. The main reason to do this is that the radio communication adds some latency and some packets may even be lost, so if the trajectory generation is done on the PC, the movement is not smooth. Moving the trajectory generation on the robot means that the latency of the radio communication is not present anymore, the only latency is due to the CAN communication, but it is negligible for our application.
The following is the code we have used to achieve this result:

```
bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(l_rounded));
```

6.2.2 Exercise 5.2

For exercise 5.2 the objective was to be able to set the parameters of the reference signal generator from the computer.
To achieve this, the parameters have to be sent using the radio interface, both frequency and amplitude of the sine wave. These parameters have therefore been defined as global variables, which are updated by means of a radio callback function. We defined the global variables as *volatile* to make sure that the compile does not optimize away anything from the code that manages those variables.
The callback function is shown below:

```
static int8_t register_handler(uint8_t operation, uint8_t address,
    RadioData* radio_data)
{
  if(operation == ROP_WRITE_8)
  {
    if (address == 10)
    {
      freq = DECODE_PARAM_8(radio_data->byte,(0),(2));

      return TRUE;
    }
    else if (address == 11)
    {
      ampl = DECODE_PARAM_8(radio_data->byte,(-60),(60));

      return TRUE;
    }
  }
  return FALSE;
}
```

A basic code for the PC side has been written to take as inputs from the user the parameters to be sent to the robot and then activate it until a key is pressed on the keyboard:

```
cout << "Insert␣frequency:␣";
cin >> freq;
cout << "Insert␣amplitude:␣";
cin >> ampl;
```

```
cout << freq << "␣" << ampl << endl;

regs.set_reg_b(REG8_MODE, 2);

regs.set_reg_b(10, ENCODE_PARAM_8(freq,(0),(2)));
regs.set_reg_b(11, ENCODE_PARAM_8(ampl,(-60),(60)));

ext_key();

regs.set_reg_b(REG8_MODE, 0);
```

7 Exercise 6

During the second session of the practical, we used the LED tracking system.
The first task has been to turn on the system, and then to calibrate it.

7.1 Calibration

To calibrate the system means to define a matrix of transformation from the points given in the image to the actual spatial coordinates.
This is done by positioning an LED in known positions in the pool and running a script which generates the transformation matrix.

7.2 Given code

Running the given code, the user can retrieve information about the position of the robot in the aquarium in real time. After the connection to the video tracking system, the program shows the current coordinates of the robot on screen using the function *get_pos()* defined in the *trkcli* library.

7.3 Modified code

In exercise 6.1, we modified the code to let the LED on the head module change according to the current position of the robot. The LED is set sending to the designed register a 32-bit value. To control each RGB channel's intensity, we set respectively the third, second and first (starting from the right) half-word of this 32-bit register. Here the shift operator (*«*) comes in handy to properly set the register.

```
uint32_t rgb = ((uint32_t) r << 16) | ((uint32_t) g << 8) | b;
regs.set_reg_dw(0, rgb);
```

In the main loop, after updating the position of the robot, we compute the correct intensity to be assigned to the red and blue values of the RGB LED. Then we send it to the robot using the function *set_reg_dw()*

```
if (id != -1 && trk.get_pos(id, x, y)) {
  cout << "(" << fixed << x << ",␣" << y << ")" << "␣m␣␣␣␣␣␣\r";
} else {
  cout << "(not␣detected)" << '\r';
}

r = x / X_MAX * 255;
b = y / Y_MAX * 255;

rgb = ((uint32_t) r << 16) | ((uint32_t) g << 8) | b;
regs.set_reg_dw(0, rgb);
```

It's also important to remember to deactivate all the other LEDs on each motor module, in order to have a correct reading from the tracking system. This is done in the main body of the robot script, just after the hardware initialization.

8 Exercise 7

In this exercise, we are required to develop a trajectory generator for the lamprey robot. To do that, we need to generate a *traveling wave*.
We therefore send to each motor module the value of the sine wave as defined in Equation 8.0.1, introducing an increasing phase lag and decreasing the amplitude of the oscillation starting from the tail towards the head.

$$\theta_i = A(1 - i\psi)\sin\left(2\pi\left(\nu t + \frac{i\phi}{N}\right)\right) \tag{8.0.1}$$

where A is the maximum oscillation half-amplitude, ψ is the amplitude decrease rate, ν the oscillation frequency, t the time, i the element number (starting at 0 from the tail), ϕ the total phase lag, and N the total number of active elements in the robot.

8.1 Code

8.1.1 PC side

After initializing the connection with the tracking system and the robot, we let the user insert the desired frequency, amplitude and phase lag that will be sent to the robot. This is done by parsing the program parameters from the command line and not using the *standard input (STDIN)* interface.

```
float freq = strtod(argv[1], 0),
      ampl = strtod(argv[2], 0),
      phase= strtod(argv[3], 0);

[...]

regs.set_reg_b(10, ENCODE_PARAM_8(freq,(0),(2)));
regs.set_reg_b(11, ENCODE_PARAM_8(ampl,(0),(60)));
regs.set_reg_b(12, ENCODE_PARAM_8(phase,(0.5),(1.5)));
```

Subsequently, we collect data on the position of the robot and save it into a CSV file to perform offline analysis.

```
double startingTime = time_d();
double currentTime  = time_d();

while(currentTime - startingTime < TEST_DURATION)
{
  currentTime = time_d();

  // Gets the current position
  if (!trk.update(frame_time)) {
    cerr << "Could not get position from tracking." << endl;
  } else {

    // Gets the ID of the first spot (the tracking system supports
       multiple ones)
    int id = trk.get_first_id();

    // Reads its coordinates (if (id == -1), then no spot is detected)
```

```
      if (id != -1 && trk.get_pos(id, x, y)) {
          x_hist.push_back(x);
          y_hist.push_back(y);
          t_hist.push_back(currentTime);
          cout.precision(8);
          cout << "x: " << x << " y: " << y << " t: " << fixed <<
              currentTime << endl;

      } else {
          cerr << "Could not get position from tracking." << endl;
      }
    }
  }
```

8.1.2 Robot side

In the robot script, we first defined the register callback to properly initialize the sinusoidal values

```
static int8_t register_handler(uint8_t operation,
                               uint8_t address,
                               RadioData* radio_data)
{
  switch (operation)
  {
    case ROP_WRITE_8:
      if (address == 10) {
        freq = DECODE_PARAM_8(radio_data->byte,(0),(2));
        return TRUE;
      } else if (address == 11) {
        ampl_set = DECODE_PARAM_8(radio_data->byte,(0),(60));
        return TRUE;
      } else if (address == 12) {
        phase = DECODE_PARAM_8(radio_data->byte,(0.5),(1.5));
        return TRUE;
      }
  }
  return FALSE;
}
```

Then, in the main loop, for each time step, we compute the correct value of the sinusoidal wave to be set for each motor module.

```
    do {
    // Calculates the delta_t in seconds and adds it to the current time
    dt = getElapsedSysTICs(cycletimer);
    cycletimer = getSysTICs();
    delta_t = (float) dt / sysTICSperSEC;

    my_time += delta_t;

    for (uint8_t i = 0; i < NUM_POS; i++) {
      ampl = (1 - i * AMPLITUDE_DECREASE_RATE) > 0 ? ampl_set * (1 - i *
          AMPLITUDE_DECREASE_RATE > 0) : 0;

      l = ampl * sin(M_TWOPI * (freq * my_time + i * phase / NUM_POS));
      l_rounded = (int8_t) l;
```

```
        bus_set(addresses[i], MREG_SETPOINT, DEG_TO_OUTPUT_BODY(l_rounded)
            );
    }


    // Make sure there is some delay, so that the timer output is not
        zero
    pause(ONE_MS);

} while (reg8_table[REG8_MODE] == IMODE_TRAVELING_WAVE_DEMO);
```

8.2 Testing phase

During the testing phase, we collected data on different configurations of amplitude, frequency and phase lag, and for each, we computed (offline) the average speed and its standard deviation. All the results are summarized in Table 1.

Amplitude [°] - Frequency [Hz] - Phase lag	Average speed	Speed Standard Deviation
35 - 1 - 0.5	0.6012	0.0419
35 - 1 - 0.8	0.3809	0.0020
35 - 1 - 1.2	0.1806	0.0147
35 - 1.4 - 0.5	0.5702	0.0051
35 - 1.4 - 0.8	0.4837	0.0271
35 - 1.4 - 1.2	0.2419	0.0049
20 - 1 - 0.5	0.3747	0.0221
20 - 1 - 0.8	0.2212	0.0318
20 - 1 - 1.2	0.0379	0.0442

Table 1: Experiments results

From the data, it's visible that, when considering the same amplitude and frequency, the speed performance drops as soon as the phase lag is increased.
In the picture below you can see the evolution of the speed as a function of the phase lag, for the 3 different configurations of the wave that we have tested.

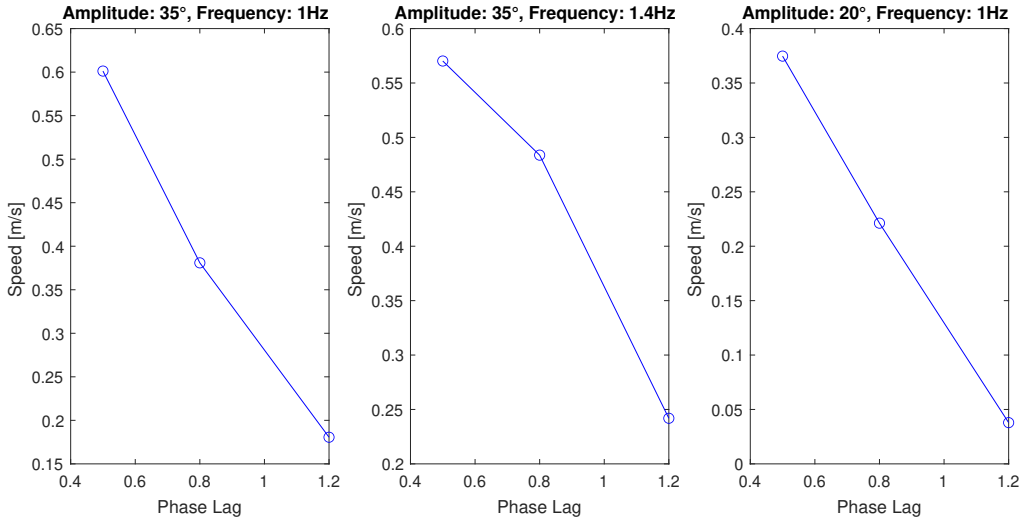

Figure 1: Speed as a function of the phase lag for different wave configurations.

Figure 2 shows some trajectories for different wave configurations.

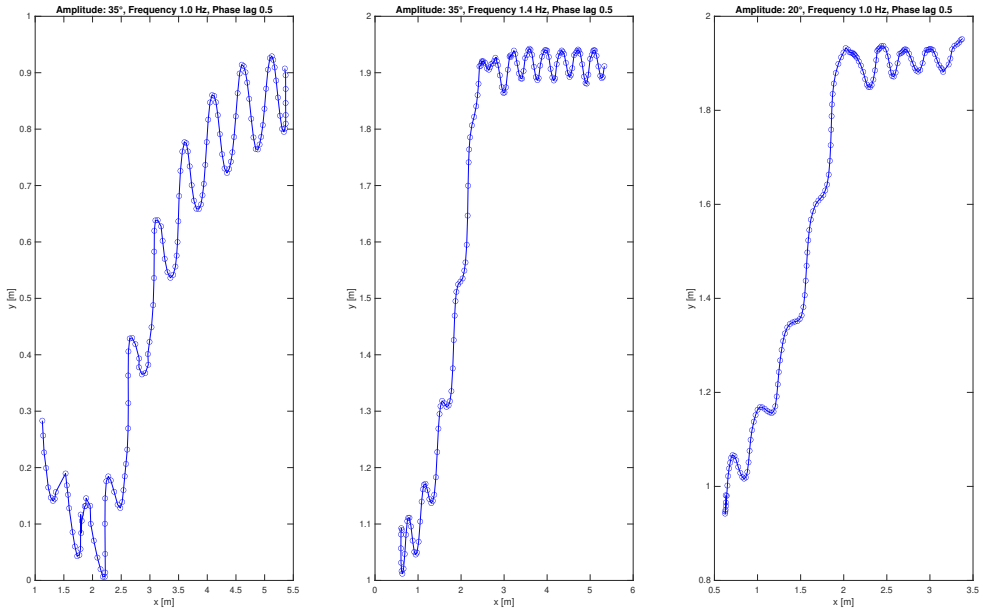

Figure 2: Trajectories for different wave configurations.

9 Exercise 8

As a bonus exercise, we decided to implement a program to let the user control the lamprey robot using the PC keyboard.

9.1 Code

9.1.1 PC side

Firstly, as we did in Exercise 7, we let the user set a desired configuration of amplitude, frequency and phase lag. In addition, the user can decide how aggressive the steering should be, defining a value in the range $[-30\,\text{deg}; +30\,\text{deg}]$. Then, in the main loop, we collect the user input through the *Standard Input* interface. The user can choose among three available control inputs

- 'w', the robot will move straightforward;
- 'a', the robot will turn left;
- 'd', the robot will turn right;

After updating the control input, we send to the robot the corresponding shift in amplitude.

```
while(currentTime - startingTime < TEST_DURATION)
  {
    currentTime = time_d();

    uint16_t inp = ext_key();
    char a = inp & 0xFF;
    if(a == 'a') steering = 20;
    else if(a == 'w') steering = 0;
    else if(a == 'd') steering = -20;
    cout << a << endl;
```

```
        regs.set_reg_b(13, ENCODE_PARAM_8(steering,(-30),(+30)));
    }
```

In addition, we have implemented a tracking program that is run in parallel of the main program, and just reads the position of the robot and saves the data to a CSV file, as done in the previous exercise.

9.2 Robot side

After the usual initialization, in the main loop, we compute the wave needed by the robot to move, taking into account the steering command. To do so, we add an offset to the computed wave value; if the offset is greater than zero, then the robot will turn left, otherwise, it will turn right.

```
for (uint8_t i = 0; i < NUM_POS; i++) {
  ampl = (1 - i * AMPLITUDE_DECREASE_RATE) > 0 ? (ampl_set) * (1 - i
      * AMPLITUDE_DECREASE_RATE > 0) : 0;

  l = ampl * sin(M_TWOPI * (freq * my_time + i * phase / NUM_POS)) +
      steering;

  if (l < -60) l = -60;
  else if (l > 60) l = 60;

  l_rounded = (int8_t) l;

  bus_set(addresses[i], MREG_SETPOINT, DEG_TO_OUTPUT_BODY(l_rounded)
      );
}
```

Figure 3 shows the trajectories we recorded while the robot was controlled manually using the keyboard.

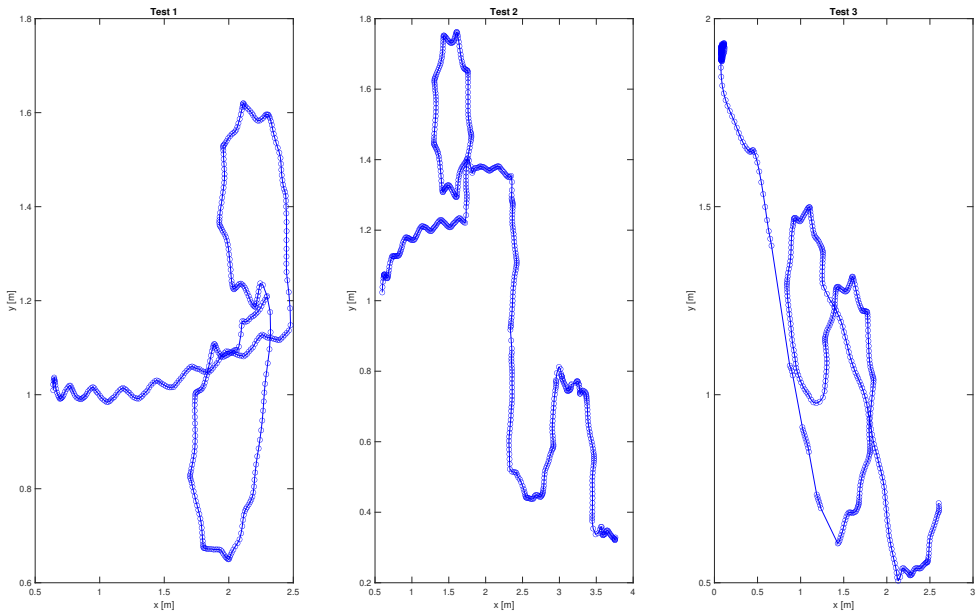

Figure 3: Trajectories during manual control experiments.

10 Conclusion

In conclusion, the practical exercise detailed in this report has provided invaluable insights into the programming and characterization of a swimming fish robot. Through the integration of modules from established robotic platforms, we were able to develop and implement basic programs both on the microcontroller and the computer, going from basic exercises to understanding the working principles of the system to the generation of a trajectory generator and steering algorithm. The characterization of the robot behaviour, by means of the analysis of trajectory using a video tracking system, underscored the relationship between trajectory parameters and locomotion performances. Overall, this practical not only enhanced our understanding of robotics but also equipped us with essential hands-on experience, paving the way for further advancements in the field of aquatic robotics.