TP8: A MODULAR FISH ROBOT

FINAL REPORT

Jiangfan Li, Chengkun Li, Zakarya Souid

EPFL

1 Exercise1: First program

Source code in robot/ex1/main.c will control the LED manually, displaying different colors with a period of 10 milliseconds. The experimental phenomena is as expected, the LED shines in a continuous manner, from red to orange to white to cyan and to blue.

To make the green light blink in 1 Hz, we modify the code in the following way: light on the green LED and pause for half a second, and then turn off the LED for half a second.

```
// Blink in green in 1 Hz
set_rgb(0,127,0);
pause(ONE_SEC/2);
set_rgb(0,0,0);
pause(ONE_SEC/2);
```

2 Exercise2: Registers

The program generated by robot/ex2/main.c will blink the led (first red then green) at first showing initialization has been done, and then keeps blinking in green to indicate that it's working. At the meantime, it will monitor and handle the messages transported from the radio. Specifically, the registers' functions are showed in Table 1. The unshowed register operation will be treated as an error.

Register	R/W	Address	Function
8-bit	R	6	read and reset the variable counter
8-bit	R	21	counter++, return 0x42
8-bit	W	2/3/4	assign the [address-2]th byte of mb_buffer the input byte
16-bit	W	7	assign 3*datavar + input to datavar
32-bit	R	2	read datavar
multibyte	R	2	read mb_buffer
multibyte	W	2	assign input + 4 to mb_buffer byte and byte

Table 1: Register Map for Exercise 2

For the PC, the program generated by pc/ex2/ex.cc will initialize the radio interface first, then send the reboot command, and execute several register operations. At last, it will shut down the remote connection with the robot.

The register operations are, indicated by Table 1, as following.

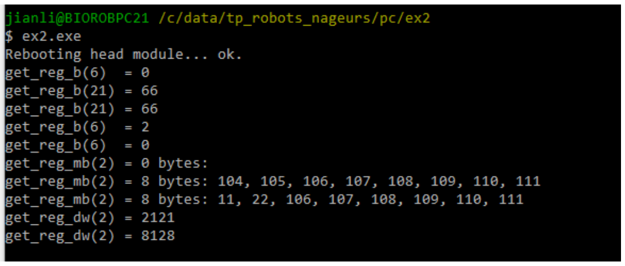
1. read (which should be 0) and reset the variable counter
2. increase the variable counter by 1, and the data received will be 0x42
3. increase the variable counter by 1, and the data received will be 0x42
4. read counter (which should be 2) and reset the variable counter

5. read counter (which should be 0) and reset the variable counter
6. read mb_buffer (which should be empty and '0'-byte long[1]) the variable mb_buffer and display it
7. assign [100+4, 101+4, 102+4, 103+4, 104+4, 105+4, 106+4, 107+4] to the variable mb_buffer
8. read (which should be [104, 105, 106, 107, 108, 109, 110, 111] now) the variable mb_buffer and display it
9. assign 11 to mb_buffer[0]
10. assign 22 to mb_buffer[1]
11. read (which should be [11, 22, 106, 107, 108, 109, 110, 111] now) the variable mb_buffer and display it
12. update datavar by 3*datavar + 2121
13. read variable datavar, which should be 2121
14. update datavar by 3*datavar + 1765
15. read variable datavar, which should be 8128[2]

So the output on the screen will be:

```
get_reg_b(6)  = 0
get_reg_b(21) = 66
get_reg_b(21) = 66
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 verified output from the terminal is shown in Fig. 1



```
jianli@BIOROBPC21 /c/data/tp_robots_nageurs/pc/ex2
$ ex2.exe
Rebooting head module... ok.
get_reg_b(6)  = 0
get_reg_b(21) = 66
get_reg_b(21) = 66
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
```

Figure 1: Output of Exercise 2.

3 Exercise3: Communication

The program generated by robot/ex3/main.c will initialize the motor with address 21, and query the position of the motor. It will show yellow signal if the position is positive, and cyan if negative. The intensity of the light is correlated to the absolute value of the position.

We first save the addresses of all four motors in the code of robot side:

```c
// the addresses of all the motors
const uint8_t MOTOR_ADDRS[4] = {21, 72, 73, 74};
// the positions of all the motors
int8_t pos[4];
```

[1]It's actually 29-byte long, but will be operated as last_mb_size-byte long
[2]datavar is of uint32_t type, so it will not overflow.

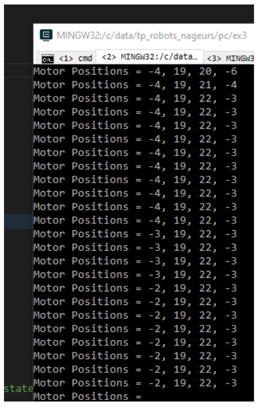
Then, we read the data from pos on pc side via radio by modifying the `register_handler` as follows:

```
static int8_t register_handler(uint8_t operation, uint8_t address, RadioData* radio_data)
{
  uint8_t i;
  switch (operation)
  {
    case ROP_READ_MB:
      if (address == 2) {
        radio_data->multibyte.size = 4;
        for (i = 0; i < 4; i++) {
          radio_data->multibyte.data[i] = pos[i];
        }
        return TRUE;
      }
      break;
  }
  return FALSE;
}
```

Finally in the main function we repeatedly print the motors' DOF by using the `mb_buffer`:

```
// Displays the contents of a multibyte register as a list of bytes
void display_multibyte_register(CRemoteRegs& regs, const uint8_t addr)
{
  uint8_t data_buffer[32], len;
  if (regs.get_reg_mb(addr, data_buffer, len)) {
    for (unsigned int i(0); i < len; i++) {
      if (i > 0) cout << ", ";
      cout << (int)(int8_t) data_buffer[i]; // uint8 -> int8(char) -> int
    }
    cout << endl;
  } else {
    cerr << "Unable to read multibyte register." << endl;
  }
}

int main()
{
  CRemoteRegs regs;

  if (!init_radio_interface(INTERFACE, RADIO_CHANNEL, regs)) {
    return 1;
  }

  // Reboots the head microcontroller to make sure it is always in the same state
  reboot_head(regs);

  while(!kbhit()){
    cout << "Motor Positions = ";
    display_multibyte_register(regs, 2);
  }
  regs.close();
  return 0;
}
```

What's worth mentioning here is that the data from data_buffer is in the data type unint8_t and we need to cast in into char first and then cast it into int to display properly in the terminal as well as allowing it to be negative. The final output is shown in Fig. 2



MINGW32:/c/data/tp_robots_nageurs/pc/ex3

```
Motor Positions = -4, 19, 20, -6
Motor Positions = -4, 19, 21, -4
Motor Positions = -4, 19, 22, -3
Motor Positions = -4, 19, 22, -3
Motor Positions = -4, 19, 22, -3
Motor Positions = -4, 19, 22, -3
Motor Positions = -4, 19, 22, -3
Motor Positions = -4, 19, 22, -3
Motor Positions = -4, 19, 22, -3
Motor Positions = -4, 19, 22, -3
Motor Positions = -4, 19, 22, -3
Motor Positions = -4, 19, 22, -3
Motor Positions = -3, 19, 22, -3
Motor Positions = -3, 19, 22, -3
Motor Positions = -3, 19, 22, -3
Motor Positions = -3, 19, 22, -3
Motor Positions = -2, 19, 22, -3
Motor Positions = -2, 19, 22, -3
Motor Positions = -2, 19, 22, -3
Motor Positions = -2, 19, 22, -3
Motor Positions = -2, 19, 22, -3
Motor Positions = -2, 19, 22, -3
Motor Positions = -2, 19, 22, -3
Motor Positions = -2, 19, 22, -3
Motor Positions =
```

Figure 2: Output of Example 3.

4 Exercise4: Position Control

The verification is done by following the instruction in the manual. The address of the last chain in our model is 21, so we set `MOTOR_ADDR = 21;` To generate sine waves at 1 Hz, we made the following modifications:

First, in the robot part, we created a variable to save the setpoint degree on the robot side:

```
int8_t mysetpoint = 0;
```

The value comes from our pc side and is stored in this variable by using the register handler:

```
// listen to the setpoint command
static int8_t register_handler(uint8_t operation, uint8_t address, RadioData* radio_data)
{
  switch (operation)
  {
    case ROP_WRITE_8:
      if (address == REG8_SETPOINT) {
        mysetpoint = (int8_t) radio_data->byte;
        return TRUE;
      }
      break;
  }
  return FALSE;
}
```

And then we call this degree in the actuating part

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

And for the pc side, the computer should send the command of sine wave to the radio with frequency of 1 Hz. To do so, we made the following modifications in the main loop:

```
double starter = time_d();
double cnt = starter;
while(!kbhit()){//quit when key hit
  if (time_d() - cnt > 1.0){//send command at frequency of 1 Hz (expected)
      regs.set_reg_b(REG8_SETPOINT, (int) 40*sin((time_d() - starter)*2*M_PI*1));
      cnt = time_d();
      }
  }
```

5 Exercise5: Trajectory

5.1 Simple on-board trajectory generation

If the robot is placed near the radio transmitter, the sine wave obtained by sending commands from the robot side is similar to the sine wave obtained by using the controller on the PC. However, if the robot is too far away from the radio transmitter in the PC control scheme, the motion becomes laggy and not smooth.

5.2 Modulating trajectory parameters

In this part, we use `ENCODE_PARAM_8` to encode the user input amplitude as well as frequency (floating point) as bytes on the PC side and decode them on the robot side.

The modifications on pc side are shown below:

```
float f_amp = -1, f_freq = -1;
while(f_amp<0 || f_amp>60){
cout << "Amp(0-60):";
cin >> f_amp;
}
while(f_freq<0 || f_freq>2){
cout << "Freq(0-2):";
cin >> f_freq;
}

// turn floats to bytes
uint8_t amp = ENCODE_PARAM_8(f_amp, 0, 60);
uint8_t freq = ENCODE_PARAM_8(f_freq, 0, 2);

// send the command
regs.set_reg_b(REG8_MODE, 2);
regs.set_reg_b(REG8_AMP, amp);
regs.set_reg_b(REG8_FREQ, freq);
```

The modifications on robot side are shown below:

```
// Modify the register handler
static int8_t register_handler(uint8_t operation, uint8_t address, RadioData* radio_data)
{
  switch (operation)
  {
    case ROP_WRITE_8:
      if (address == REG8_AMP) {
        amp = DECODE_PARAM_8(radio_data->byte, 0, 60);
        return TRUE;
      }
      else if (address == REG8_FREQ)
      {
        freq = DECODE_PARAM_8(radio_data->byte, 0, 2);
      }
      break;
  }
  return FALSE;
}
```

```
// modify sine_demo_mode
l = amp * sin(M_TWOPI * freq * my_time);
```

6 Exercise6: LED Tracking System

6.1 Combining with the radio

First, to achieve the effect that LED's color changes with the location of the robot, we add the following code to the pc side:

```
uint8_t r = ENCODE_PARAM_8(x, 0, 6);
uint8_t g = 64;
uint8_t b = ENCODE_PARAM_8(y, 0, 2);
uint32_t rgb = ((uint32_t) r << 16) | ((uint32_t) g << 8) | b;
regs.set_reg_dw(REG32_LED, rgb);
```

And the verification results we got are shown in Fig. 3, the colors are green (only g), red (r & g) and purple (r & g & b) respectively. For the tracking system, we save the `x, y, t` which are x, y, and time to the output file and use them for

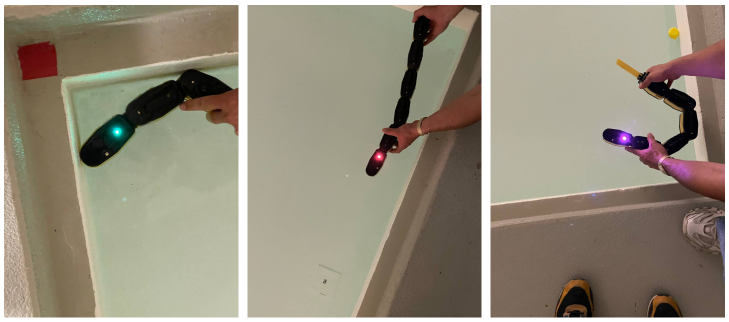


Figure 3: The color of LED at $(0, 0)$, $(x_{max}, 0)$, (x_{max}, y_{max}) respectively.

plotting in the later sections.

7 Exercise7: Swimming

7.1 Trajectory

For the lamprey robot, we used a travelling wave Eq. (1) from the head to the tail (i starts from 0 from the head).

$$\theta_i = Asin(2\pi(\nu t + \frac{(N-i)\phi}{N})) + \text{Turn constant} \tag{1}$$

We have 5 motor segments in the robot, and their addresses are:

```
#define MOTOR_NUM 5
const uint8_t MOTOR_ADDRS[MOTOR_NUM] = {25, 22, 24, 26, 23};
```

Similar to exercise 5, we send the parameters, A, ν, ϕ to the robot with the radio. And the robot was able to swim forward successfully as shown in Fig. 4, Here we report two types of metrics: Average speed and Velocity (Instantaneous

Figure 4: Lamprey swimming forward.

speed), they are defined as:

$$\text{Average speed} = \frac{1}{t}\int ds, \tag{2}$$

and

$$\text{Velocity} = \frac{ds}{dt} \tag{3}$$

An example of captured trajectory is shown in Fig. 5, where the maximum instantaneous speed is around 7 cm/s and the average speed is around 1.5 cm/s. More visualizations can be found in Section 9.

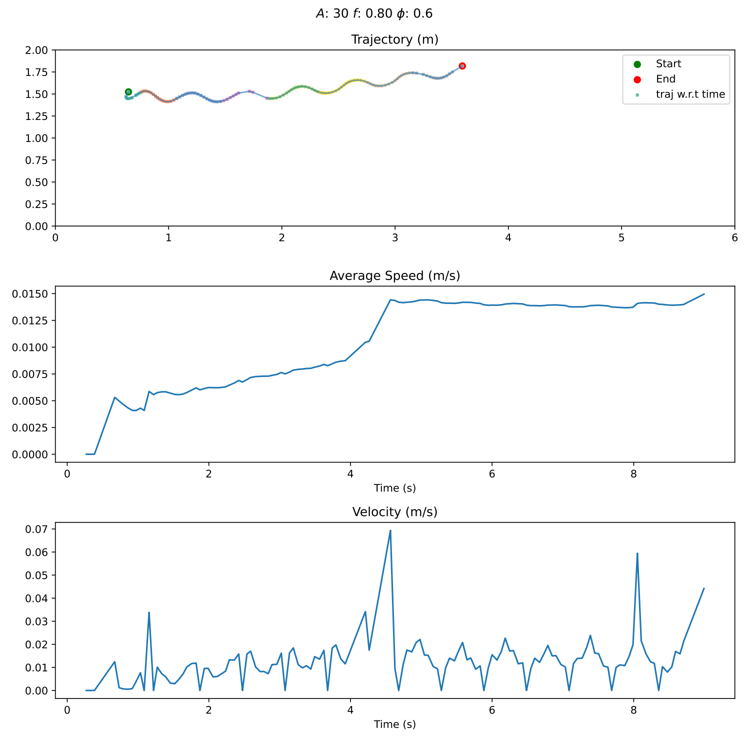


Figure 5: Swimming forward trajectory

7.2 Experiment

In this section, we try to explore the relationship between *Average speed* and *Total phase lag* ϕ in the Lamprey swimming motion setting.

In terms of experimental design, we first try various parameter combinations to find a parameter pair with acceptable performance. And then we fix the amplitude A as well as the frequency f and change only the total phase lag ϕ. In the experiments, we performed four trials for each parameter and recorded the trajectory of each trial as well as the maximum average speed of all trials.

To gather more robust data, we choose 4 different levels of phase lag and ran the experiments multiple times. We ran each phase lag level 4 times, obviously while trying to keep every other variable as close as possible. We then used a Python script to analyse the coordinates and infer instantaneous speed and average speed on the run. We then did an analysis to regroup each runs with the same parameters to try to regress speed on phase lag.

The relationship between maximum average speed and total phase lag is shown in Fig. 6. In the figure 6, the green triangle is the mean value, and std is shown on the side of each box. We can see that the parameter set with phase lag of 0.5 achieves the best maximum average speed, total phase lag of 0.6 and 0.5 achieves similar yet lower maximum average speed, and with bigger total phase lag like 1.0 the maximum average speed decreases significantly, becoming almost zero. As we can see in figure 8, for a phase lag of 1 the robot is almost stationary, taking twenty seconds to do a

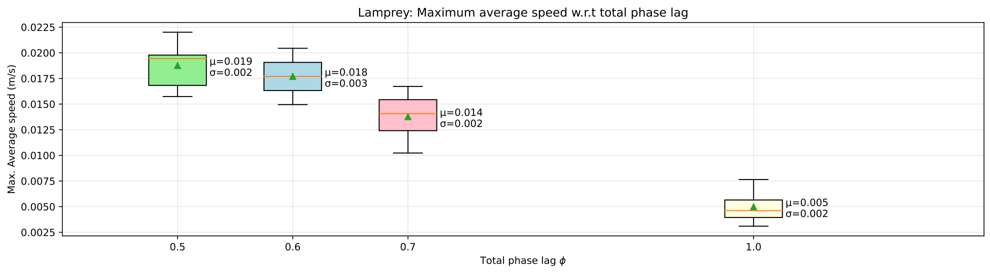


Figure 6: Relationship between maximum average speed and total phase lag

distance achieved in 3 with the phase lag of 0.5. We think an even lower phase lag would lead to slower speeds, as when phase lag goes to zero the robot starts to "wing its tail" like a dog.

Consistency within runs was fairly high as attested by the low standard deviation and the graphs 8, and one thing that's worth mentioning is that we dropped 2 trials with human intervention to compare them fairly.

8 Exercise8: More Experiments

In this part we carried out two additional experiments, namely, we

- Implemented steering control.
- Implemented tracking system as feedback to make lamprey swim back and forth.

We will briefly explain what we did to achieve these two parts in the following sections.

8.1 Steering Control

The steering control is implemented by using an extra register to store the **turn constant** in Eq. (1). In our case, we set the turn constant to 15/-15 to achieve a left/right turn. And from our observation, the turns look very natural to the eye.

The detailed implementation is shown below:

```
// pc side
uint8_t turn = ENCODE_PARAM_8(f_turn, -30, 30);
regs.set_reg_b(REG8_TURN, turn);

// robot side
static int8_t register_handler(uint8_t operation, uint8_t address, RadioData* radio_data)
{
  switch (operation)
  {
    ...
      else if (address == REG8_TURN)
      {
        turn = DECODE_PARAM_8(radio_data->byte, -30, 30);
      }
      break;
    ...
  }
}

// Output sine wave to motors
for(int i = 0; i<MOTOR_NUM; i++){
  // Calculates the sine wave
  l = amp * sin(M_TWOPI * (freq * my_time + (MOTOR_NUM - (float)i) *phi /MOTOR_NUM )) + turn ;
  l_rounded = (int8_t) l;

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

8.2 Back and Forth Swimming

In this part, we want to achieve a back and forth swimming pattern with the help of tracking system. We set the travel range of the robot as $[1.0, 4.0]m$ on the x direction of the pool. And the goal in our case was achieved with the following logic:

Program Logic

1. if get out of the range $\rightarrow$ turn for maximum 4 seconds.
2. else swim straight.

The reason for this design is that the lamprey may overturn and not being able to get inside the range if we do not add timer constraint to the control. The final implementation on pc side is shown below:

```cpp
// Only the turning logic part of the code is displayed
bool done = false;
bool initialized = false;
double start_x = 0.f, start_y = 0.f;
double cur_distance = 0.f;
double last_x=0, last_y=0;

clock_t start, now;
clock_t turn_start = 0;
bool turned = false;
start = clock();
while (!kbhit() && !done) {
    uint32_t frame_time;
    // Gets the current position
    if (!trk.update(frame_time)) {
      return 1;
    }
    // Gets the ID of the first spot (the tracking system supports multiple ones)
    int id = trk.get_first_id();
    double x, y; // current position
    // Reads its coordinates (if (id == -1), then no spot is detected)
    if (id != -1 && trk.get_pos(id, x, y)) {
      now = clock();
      // Get initial position of robot
      if (!initialized) {
        last_x = start_x = x;
        last_y = start_y = y;
        initialized = true;
      }
      // if it's approaching the boundary
      if(x < 1.0 || x > 4.0 ){
        if(!turned){
          // turn left for max. 4s
          regs.set_reg_b(REG8_TURN, turn);
          turn_start = now;
          turned = true;
        }
      }
      else {
        regs.set_reg_b(REG8_TURN, turn_zeros);
        turned = false;
      }
      if((float)(now - turn_start )/ CLOCKS_PER_SEC > 4){
        regs.set_reg_b(REG8_TURN, turn_zeros);
      }
    } else {
      cout << "(not detected)" << '\r';
    }
    // accumulate the distance
    cur_distance += sqrt(pow(x - last_x, 2) + pow(y - last_y, 2));
    if(cur_distance >= distance) {
      done = true;
      break;
    }
    last_x = x;
    last_y = y;
...
```

The final performance of lamprey can be found from our YouTube link[3], and the trajectory plot is shown in Fig. 7.

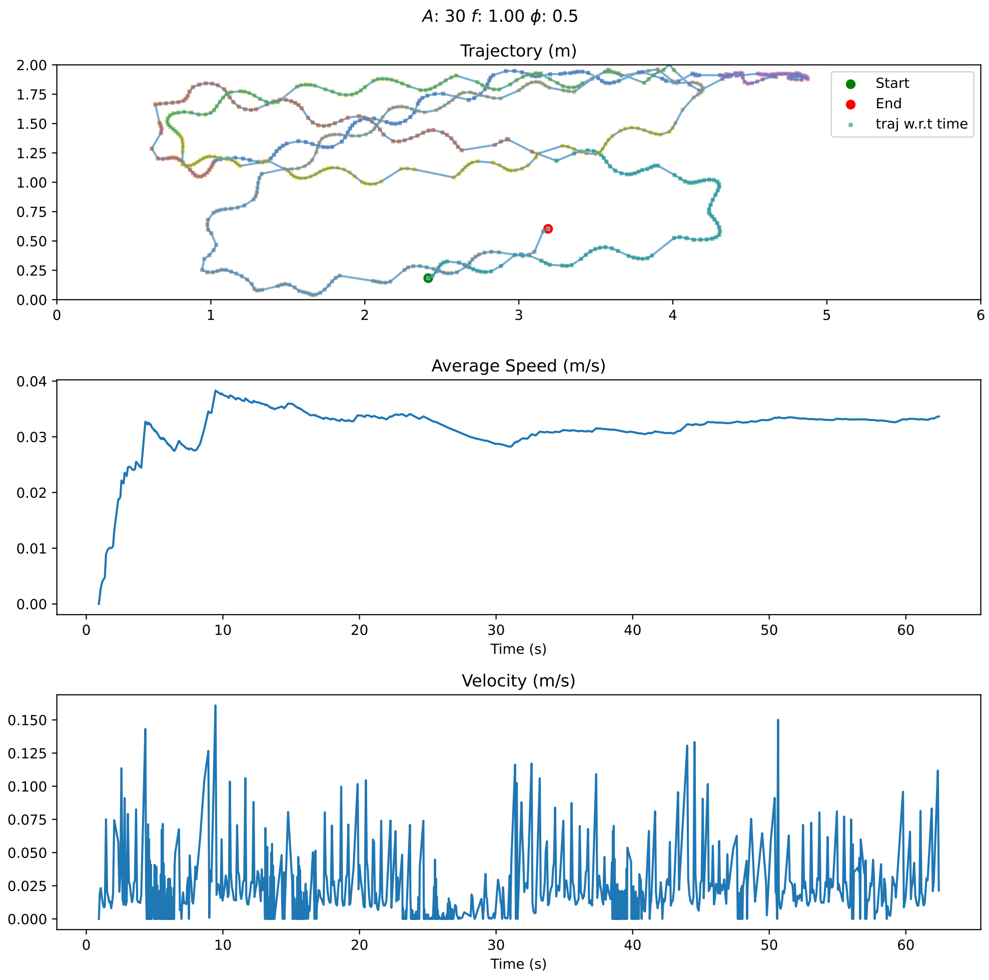


Figure 7: Trajectory plot of turning

From the trajectory, we can see that most of the trajectory is within the predetermined range (or at least not too far from it), however, the lamprey is stuck on the turning point at around 30s, so it is slightly outside the predetermined range at that period of time. The maximum instantaneous velocity we achieved in this trajectory is around 15 cm/s and the average speed of the entire trajectory is around 3 cm/s.

9 Appendix: Some Trajectories Visualization

In this part, we show some trajectories we acquired from the experiments.

[3]Video link: `https://www.youtube.com/watch?v=9Rig8uKHdts`

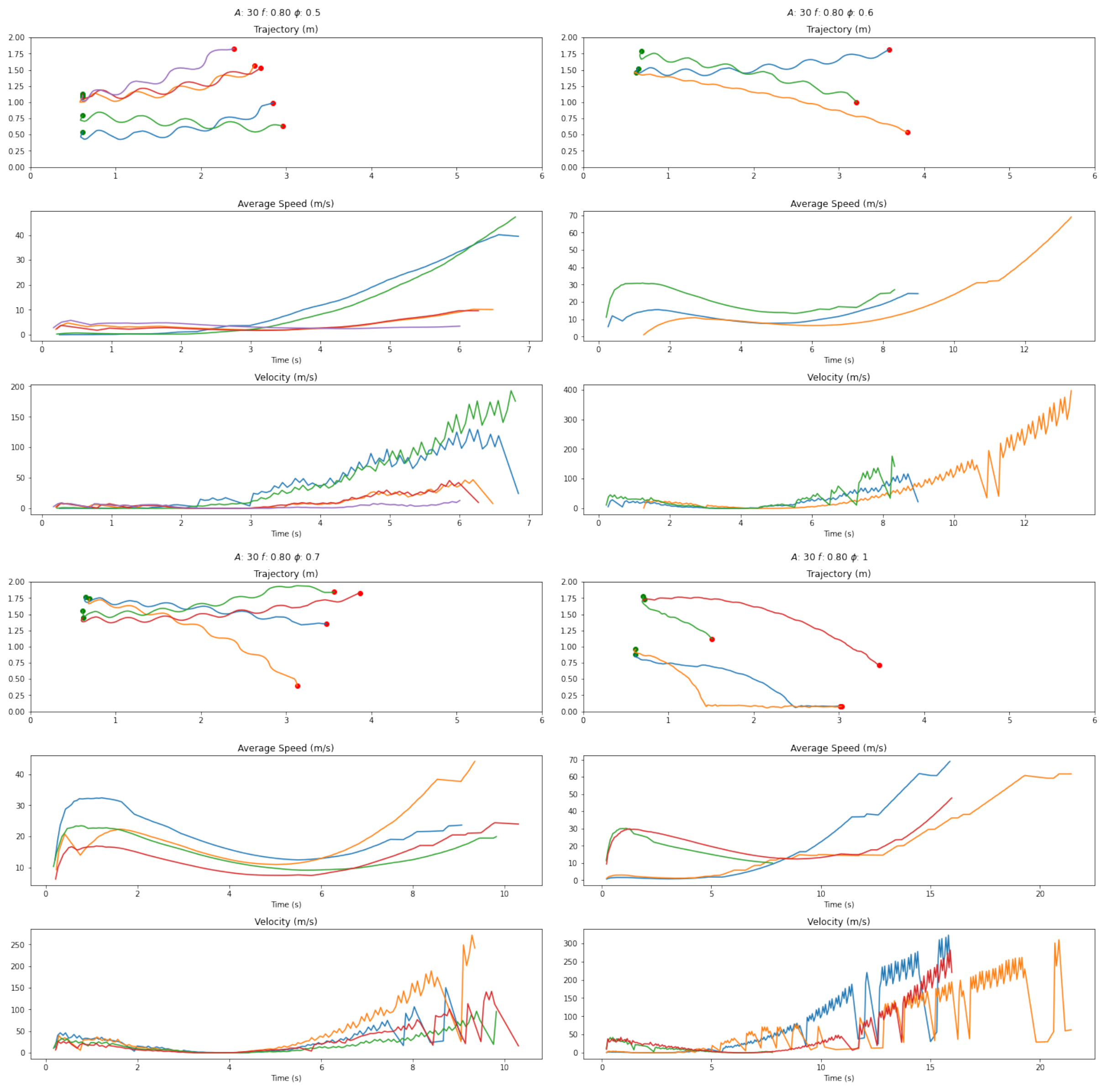


Figure 8: All trajectories of all sets of parameters used in 7.2

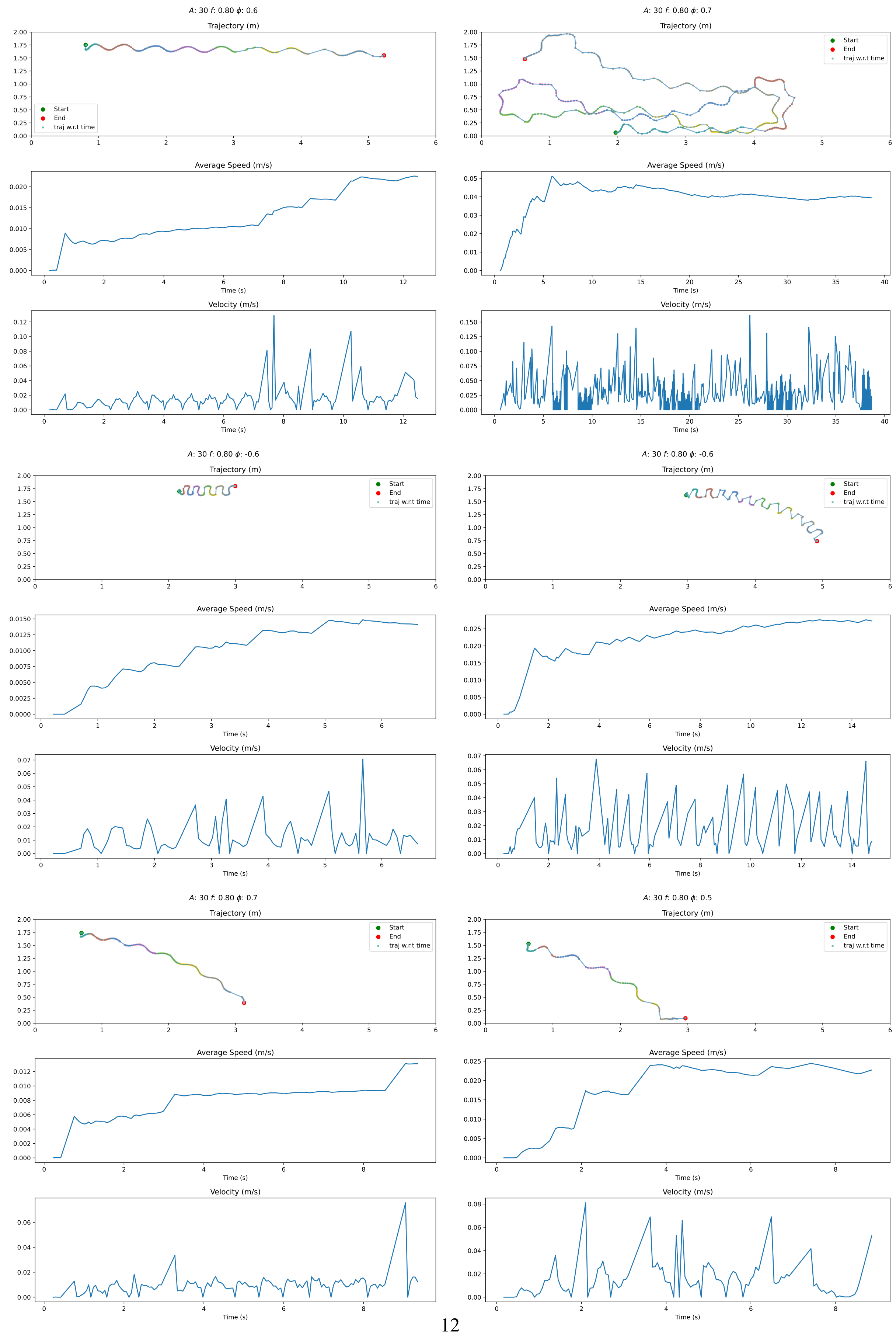


Figure 9: Some interesting trajectories