Control of a Modular Fish Robot

Raphaël Cherney Frédéric Wilhelm

May 14, 2012

1 Introduction

For this exercise, we learned how to program and control a modular, robotic fish. The boxfish-like robot is built using 3 modules from the *Salamandra robotica II* platform (Figure 1): a passive head, a midsection with pectoral fins, and a tail. Each module is completely independent with its own motor, microcontroller, and battery. They can be assembled into many possible configurations, a subset of which are shown in Figure 2.

Figure 1: *Salamandra robotica II* modular robot (Crespi 2011)

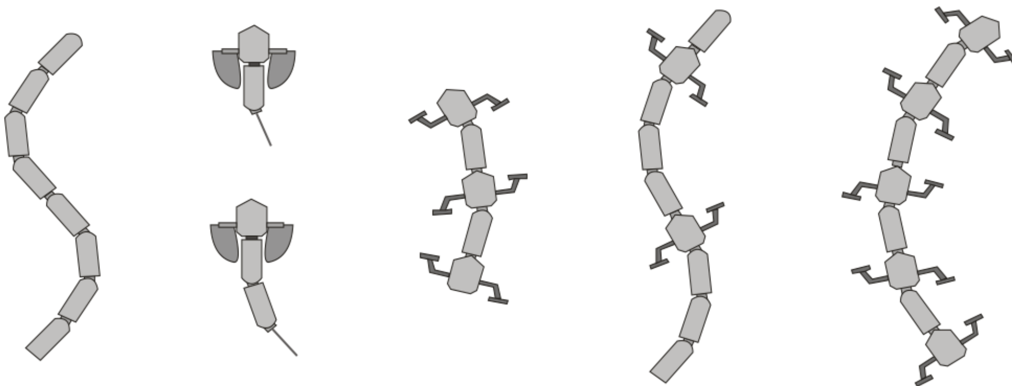

Figure 2: Possible configurations using modules (Crespi 2007)

The configuration of our fish robot with 4 degrees of freedom is shown in Figure 3. The head module is not actuated, but contains a wireless radio link to an off-board computer. We can wireless reprogram the head module and can communicate in realtime with the robot by writing to registers in the head module. More specifics on the hardware can be found in [1, 2]. This report contains a detailed description of the code we used to get started with the robot and a series of experiments taken while swimming (jump to section 9 on page 11 for swimming experiments).

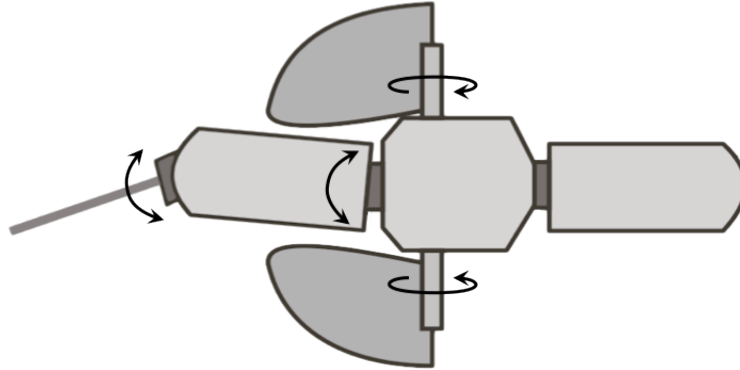

Figure 3: Configuration of fish robot (adapted from Crespi)

2 Programming the robot

In order to get started with the platform, we began with a simple program that cycles the color of the LED on the head module. The code is shown below:

```
1  int main(void)
2  {
3      int8_t i;
4
5      hardware_init();
6      reg32_table[REG32_LED] = LED_MANUAL;
7
8      // manual LED control
9      while (1) {
10         for (i = 0; i < 127; i++) {
11             set_rgb(i, 0, 0);
12             pause(TEN_MS);
13         }
14         for (i = 0; i < 127; i++) {
15             set_rgb(127, i, 0);
16             pause(TEN_MS);
17         }
18         for (i = 0; i < 127; i++) {
19             set_rgb(127, 127, i);
20             pause(TEN_MS);
21         }
22         for (i = 127; i >= 0; i--) {
23             set_rgb(i, 127, 127);
24             pause(TEN_MS);
25         }
26         for (i = 127; i >= 0; i--) {
27             set_rgb(0, i, 127);
28             pause(TEN_MS);
29         }
30         for (i = 127; i >= 0; i--) {
31             set_rgb(0, 0, i);
32             pause(TEN_MS);
33         }
```

```
34      }
35      return 0;
36  }
```

This program causes the light to progressively get more green, shift to yellow, white, cyan, blue, and then turn off before repeating the cycle. A ten millisecond pause in each for loop slows the transitions to make them visible (each transition then takes approximately $0.01 * 127 = 1.27$ seconds). Using this understanding of how to control the LED, we made the following, simple program to blink the LED green at 1 Hz:

```
1   int main(void)
2   {
3       int8_t i;
4
5       hardware_init();
6       reg32_table[REG32_LED] = LED_MANUAL;
7
8       while (1) {
9           // Blink green at 1 Hz
10          set_rgb(0, 127, 0);
11          pause(ONE_SEC/2);
12          set_rgb(0, 0, 0);
13          pause(ONE_SEC/2);
14      }
15      return 0;
16  }
```

3 Sending and receiving data

In order to communicate with the robot, the computer wirelessly reads and writes various registers in the head module's microcontroller. The robot then handles associated commands in an interrupt service routine (ISR). We use a special register callback function to handle our custom commands. We were given the following example register handler function with several specialized commands:

```
1   // Initialize variables
2   static uint32_t datavar = 0;
3   static uint8_t last_mb_size = 0;
4   static uint8_t counter = 0;
5   static uint8_t mb_buffer[MAX_MB_SIZE];
6
7   // Register callback function, handles some new registers on the radio.
8   static int8_t register_handler(uint8_t operation, uint8_t address, RadioData* radio_data)
9   {
10      uint8_t i;
11
12      switch (operation)
13      {
14          case ROP_READ_8:
15              if (address == 6) {
16                  radio_data->byte = counter;
17                  counter = 0;
18                  return TRUE;
19              } else if (address == 21) {
20                  counter++;
21                  radio_data->byte = 42;
22                  return TRUE;
23              }
24              break;
25          case ROP_READ_32:
26              if (address == 2) {
27                  radio_data->dword = datavar;
28                  return TRUE;
29              }
30              break;
31          case ROP_READ_MB:
32              if (address == 2) {
33                  radio_data->multibyte.size = last_mb_size;
34                  for (i = 0; i < last_mb_size; i++) {
35                      radio_data->multibyte.data[i] = mb_buffer[i];
```

```
36                  }
37                  return TRUE;
38              }
39              break;
40          case ROP_WRITE_8:
41              if (address >= 2 && address <= 4) {
42                  mb_buffer[address - 2] = radio_data->byte;
43                  return TRUE;
44              }
45          case ROP_WRITE_16:
46              if (address == 7) {
47                  datavar = (datavar * 2) + radio_data->word;
48                  return TRUE;
49              }
50              break;
51          case ROP_WRITE_MB:
52              if (address == 2) {
53                  last_mb_size = radio_data->multibyte.size;
54                  for (i = 0; i < last_mb_size; i++) {
55                      mb_buffer[i] = radio_data->multibyte.data[i] + 1;
56                  }
57                  return TRUE;
58              }
59              break;
60      }
61      return FALSE;
62  }
```

What the handler function will do is dependent on the register type (8-bit byte, 16-bit word, 32-bit dword, or multibyte), address (between 0 and 255 for each type), and operation (read or write). Since all of the commands are initiated by the computer, the result depends on the program being run on the PC. Our communication demo connects to the robot through the wireless radio and then runs the following function:

```
1   void read_registers(CRemoteRegs& regs)
2   {
3       cout << "get_reg_b(6)  = " << (int) regs.get_reg_b(6) << endl;
4       cout << "get_reg_b(21) = " << (int) regs.get_reg_b(21) << endl;
5       cout << "get_reg_b(21) = " << (int) regs.get_reg_b(21) << endl;
6       cout << "get_reg_b(6)  = " << (int) regs.get_reg_b(6) << endl;
7       cout << "get_reg_b(6)  = " << (int) regs.get_reg_b(6) << endl;
8
9       cout << "get_reg_mb(2) = ";
10      display_multibyte_register(regs, 2);
11
12      uint8_t buffer[8];
13      for (int i(0); i < 8; i++) {
14          buffer[i] = 100 + i;
15      }
16      regs.set_reg_mb(2, buffer, sizeof(buffer));
17
18      cout << "get_reg_mb(2) = ";
19      display_multibyte_register(regs, 2);
20
21      regs.set_reg_b(2, 111);
22      regs.set_reg_b(3, 222);
23
24      cout << "get_reg_mb(2) = ";
25      display_multibyte_register(regs, 2);
26
27      regs.set_reg_w(7, 3447);
28      cout << "get_reg_dw(2) = " << regs.get_reg_dw(2) << endl;
29      regs.set_reg_w(7, 1234);
30      cout << "get_reg_dw(2) = " << regs.get_reg_dw(2) << endl;
31  }
```

After understanding the code, we ran the program with the following result:

```
1   get_reg_b(6)  = 0
2   get_reg_b(21) = 42
3   get_reg_b(21) = 42
4   get_reg_b(6)  = 2
5   get_reg_b(6)  = 0
6   get_reg_mb(2) = 0 bytes:
7   get_reg_mb(2) = 8 bytes: 101, 102, 103, 104, 105, 106, 107, 108
8   get_reg_mb(2) = 8 bytes: 111, 222, 103, 104, 105, 106, 107, 108
9   get_reg_dw(2) = 3447
10  get_reg_dw(2) = 8128
```

Using this, we can verify our understanding. The computer begins by requesting the byte at address 6. This causes the robot to send back the value of counter which had previously been set to zero and assigns counter = 0. Next, the computer requests the byte at address 21. This increments the counter and returns the value 42. The computer then requests byte 21 again, causing the counter increment again. This time, when the computer sends the "read byte register 6 operation" it returns the updated counter value of two and resets the value to zero. The following read command just shows that the counter had been reset to zero.

Next we try reading the multibyte register 2. This returns value of last_mb_size (which corresponds to the the number of bytes to consider in mb_buffer) and returns the data from mb_buffer. In this case, last_mb_size has been initialized to zero, so it simply returns this value with no more data. Next the computer creates the array buffer[8] = {100, 101, 102, 103, 104, 105, 106, 107} and sends a multibyte write command with this data (address = 2). Logically, we would expect the the data from buffer[8] on the computer to be written to mb_buffer on the microcontroller, but careful inspection of our handler function shows that we actually write each data element + 1 to mb_buffer. This causes the values 101 though 108 to be returned when we send the multibyte read 2 command. Next the computer sends the write byte command with an address of 2 and value of 111. The handler function when address = 2 writes the incoming data to mb_buffer[address - 2] (i.e. mb_buffer[0] = 0). Similarly, the computer then writes 222 to mb_buffer[1] (since address = 3 in that case). Next we read the multibyte register (which now has a length of 8) and see that the first two components have changed, but the rest remains the same.

Finally, we try sending a word (16-bits) to the robot with an address of 7 and value of 3447. The address tells the robot to make datavar = (datavar * 2) + new value, which in this case, makes datavar = (0 * 2) + 3447 = 3447. We can tell that the value has changed by sending the read dword command with address 2, which returns the value of datavar. When we do the same thing again with the new value of 1234, we get datavar = (2 * 3447) + 1234 = 8128 (which we again read out with the read dword command). We got the results that we were expecting (though the multibyte writing command with +1 is rather illogical), showing that we understood what the code was doing. From there, we could create our own commands in the register read/write handler.

4 Communication between elements

So far, we have only sent commands to the head module (where the radio is). In order to communicate with the other elements of our modular robot, we use a 1 Mbps CAN bus. Using a custom protocol similar to the wireless connection, the head can read and write registers in the body elements. As an example, we programmed the robot to read the position from one of the body segments and adjust the color of the LED on the head according to the measurement. The following code snippet shows the main loop:

```
while (1) {
    pos = bus_get(MOTOR_ADDR, MREG_POSITION);
    if (pos > 0) {
        set_rgb(pos, 32, 0);
    } else {
        pos = -pos;
        set_rgb(0, 32, pos);
    }
}
```

When the element is at the zero position, the LED is green. The color will shift to yellow as the measurement increases and turns more cyan as the angle measurement decreases. With this ability to communicate with other segments, we implemented a program to read back the positions of each degree of freedom and display it on the computer. The robot gets the motor positions continuously and sends the values when requested by the PC. The robot code is as follows:

```
#include "hardware.h"
#include "module.h"
#include "robot.h"
#include "registers.h"
#include <string.h>

#define NUM_MOTORS 4
const uint8_t MOTOR_ADDR[NUM_MOTORS] = {72, 73, 74, 21};

// Motor positions
uint8_t pos[NUM_MOTORS] = {0, 0, 0, 0};

static int8_t register_handler(uint8_t operation, uint8_t address, RadioData* radio_data)
{
    switch (operation)
    {
        case ROP_READ_MB:
            if (address == 0) {
                radio_data->multibyte.size = NUM_MOTORS;
                memcpy(radio_data->multibyte.data, pos, NUM_MOTORS);
                return TRUE;
            }
        break;
    }
    return FALSE;
}

int main(void)
{
    uint8_t i;

    hardware_init();

    // Changes the color of the led (red) to show the boot
    set_color_i(4, 0);

    // Registers the register handler callback function
    radio_add_reg_callback(register_handler);

    // Initialises the body module with the specified address (but do not start the PD
       controller)
    init_body_module(MOTOR_ADDR[0]);
    init_body_module(MOTOR_ADDR[1]);
    init_body_module(MOTOR_ADDR[2]);
    init_body_module(MOTOR_ADDR[3]);

    // Keeps doing the following
    while (1) {
        // Retrive motor positions
        for (i = 0 ; i < NUM_MOTORS ; i++) pos[i] = bus_get(MOTOR_ADDR[i], MREG_POSITION);
    }
    return 0;
}
```

On the PC end, we simply request the motor positions and display them. In our case, after initialization, we call the following function twice a second:

```
void read_registers(CRemoteRegs& regs)
{
    int8_t buffer[4];
    uint8_t length;
```

```
5
6     // Get positions from robot
7     regs.get_reg_mb(0, (uint8_t*) buffer, length);
8
9     // Print the data
10    cout << "Motor 72  = " << (int) buffer[0] << endl;
11    cout << "Motor 73  = " << (int) buffer[1] << endl;
12    cout << "Motor 74  = " << (int) buffer[2] << endl;
13    cout << "Motor 21  = " << (int) buffer[3] << endl;
14    cout << "length  = " << (int) length << endl;
15 }
```

5 Position control

Now that we are able to read out the position of each module in our robot, we want to control them all. On the PC, we must tell the robot to enter a setpoints following mode and then generate and send the desired motor positions. In this case, we have the motor follow a sine wave trajectory ($\pm 40°$) for 5 seconds. Here is a code snippet from the PC program:

```
1  // Change the mode from idle to setpoint following
2  regs.set_reg_b(0, 2);
3
4  // Generate a sine wave at 1 Hz for 5 sec
5  while (time <= 5*1000)
6  {
7      int8_t angle = round(40 * sin(((double) time/1000)*2*M_PI));
8
9      // Send the value to the robot
10     regs.set_reg_b(1, (uint8_t) angle);
11
12     Sleep(10);
13     time += 10;
14 }
15
16 // Change the mode back to idle
17 regs.set_reg_b(0, 0);
```

Our updated register_handler function changes the mode (idle or setpoint following) and updates the desired position based on signals from the PC:

```
1  static int8_t register_handler(uint8_t operation, uint8_t address, RadioData* radio_data)
2  {
3      switch (operation)
4      {
5          case ROP_WRITE_8:
6              if (address == 0) {
7                  // Set the mode
8                  reg8_table[REG8_MODE] = radio_data->byte;
9                  return TRUE;
10             }
11             if (address == 1) {
12                 // Set the desired position
13                 pos = (int8_t) radio_data->byte;
14                 return TRUE;
15             }
16             break;
17         default:
18             break;
19     }
20     return FALSE;
21 }
```

Finally, the following function tells a body module to move to the desired position and is called when the mode is changed to IMODE_MOTOR_SETPOINT (2 in our implementation):

```
1  // Setpoint mode: The robot gets a angular value and moves the motor accordingly
2  void motor_setpoint_mode()
```

```
3  {
4      // Starts the module
5      init_body_module(MOTOR_ADDR);
6      start_pid(MOTOR_ADDR);
7      set_color(4);
8  
9      // Sends the desired positions to the module
10     while (reg8_table[REG8_MODE] == IMODE_MOTOR_SETPOINT) {
11         bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(pos));
12     }
13 
14     // Resets the position when the mode is changed
15     bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
16     pause(ONE_SEC);
17     bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);
18     set_color(2);
19 }
```

While it may not appear to do much in the while loop, the value of the global variable pos is updated in an ISR, meaning that it will actually be sending the most recent desired position. While this implementation is functional, there is still some lag and the large amount of constant communication is far from ideal.

6 Onboard trajectory generation

In order to make the control more efficient and robust, we shifted the trajectory generation onto the modules. The user supplies the frequency, amplitude, and phase shift of their desired sine wave trajectory, and the modules follow this pattern until they are sent a new signal. In our case, we want a sine wave trajectory at 1 Hz with an amplitude of $\pm 40°$:

```
1  // Set frequency and amplitude
2  #define ENCODE_PARAM_8(p,pmin,pmax) ((uint8_t) ((p - pmin) / (float) (pmax-pmin) * 255.0))
3  regs.set_reg_b(1, ENCODE_PARAM_8(1, 0, 1));      // frequency
4  regs.set_reg_b(2, ENCODE_PARAM_8(40, 0, 60));    // amplitude
5  
6  // Start
7  regs.set_reg_b(0, 2);
8  
9  Sleep(5000);
10 
11 // Stop
12 regs.set_reg_b(0, 0);
```

On the robot side, we get the values from the robot through the register_handler function (writing them to global variables):

```
1  static int8_t register_handler(uint8_t operation, uint8_t address, RadioData* radio_data)
2  {
3      switch (operation)
4      {
5          case ROP_WRITE_8:
6              /* Mode register */
7              if (address == 0) {
8                  reg8_table[REG8_MODE] = radio_data->byte;
9                  return TRUE;
10             }
11             /* Frequency register */
12             if (address == 1) {
13                 frequency = DECODE_PARAM_8((uint8_t) radio_data->byte, 0, 1);
14                 return TRUE;
15             }
16             /* Amplitude register */
17             if (address == 2) {
18                 amplitude = DECODE_PARAM_8((uint8_t) radio_data->byte, 0, 32);
19                 return TRUE;
20             }
21             /* Phase lag register */
22             if (address == 3) {
```

```
23                  phase_lag = DECODE_PARAM_8((uint8_t) radio_data->byte, 0, 2*3.14);
24                  return TRUE;
25              }
26              /* Offset register (for turning) */
27              if (address == 4) {
28                  body_offset = DECODE_PARAM_8((uint8_t) radio_data->byte, -32, 32);
29                  return TRUE;
30              }
31              break;
32          default:
33              break;
34      }
35      return FALSE;
36  }
```

The robot then sets the motor position in the following function:

```
1   const uint8_t MOTOR_ADDR = 21;
2
3   void sine_demo_mode()
4   {
5       init_body_module(MOTOR_ADDR);
6       start_pid(MOTOR_ADDR);
7
8       uint32_t dt, cycletimer;
9       float my_time, delta_t, l;
10      int8_t l_rounded;
11
12      cycletimer = getSysTICs();
13      my_time = 0;
14      do {
15          // Calculates the delta_t in seconds and adds it to the current time
16          dt = getElapsedSysTICs(cycletimer);
17          cycletimer = getSysTICs();
18          delta_t = (float) dt / sysTICSperSEC;
19          my_time += delta_t;
20
21          // Calculates the sine wave
22          l = amplitude * sin(M_TWOPI * frequency * my_time);
23          l_rounded = (int8_t) l;
24
25          // Outputs to motor
26          bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(l_rounded));
27
28          // Make sure there is some delay, so that the timer output is not zero
29          pause(ONE_MS);
30      } while (reg8_table[REG8_MODE] == IMODE_SINE_DEMO);
31
32      // Reset position when done
33      bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
34      pause(ONE_SEC);
35      bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);
36  }
```

The result is a higher quality, smoother sine wave than we observed before. This is due to the fact that the setpoint can be updated at a higher frequency because the onboard timer operations are extremely fast and do not have the inconsistent delays associated with wirelessly passing the setpoint information. The system is also more robust to losses in wireless connectivity. Overall, this solution is better and more efficient and is therefore the method we use for our experiments.

7 Modulating trajectory parameters

Because of the way that we structured our code, it is relatively easy to control multiple modules. Using the same register_handler as shown before, we can set the multiple sine wave trajectories with the following function (called by setting the mode to IMODE_SINE):

```
#define NUM_MOTORS 4
const uint8_t MOTOR_ADDR[NUM_MOTORS] = {72, 73, 74, 21};

// Global sine wave parameters
float frequency;
float amplitude;
float phase_lag;
float body_offset;

void sine_mode()
{
    int i;

    // Enable motors
    for (i=0 ; i<NUM_MOTORS ; i++)
    {
        init_body_module(MOTOR_ADDR[i]);
        start_pid(MOTOR_ADDR[i]);
    }

    // For timer
    uint32_t dt, cycletimer;
    float my_time, delta_t;
    int8_t l_rounded;

    // Desired angles
    float theta[NUM_MOTORS]={0,0,0,0};

    cycletimer = getSysTICs();
    my_time = 0;
    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
        theta[0] = amplitude * sin(M_TWOPI * frequency * my_time) + body_offset;
        theta[3] = amplitude * sin(M_TWOPI * frequency * my_time - phase_lag) + body_offset;

        // Outputs to motor
        for (i=0 ; i<NUM_MOTORS ; i++)
        {
            bus_set(MOTOR_ADDR[i], MREG_SETPOINT, DEG_TO_OUTPUT_BODY(theta[i]));
        }

        // Make sure there is some delay, so that the timer output is not zero
        pause(ONE_MS);
    } while (reg8_table[REG8_MODE] == IMODE_SINE);

    // Reset position when done
    for (i=0 ; i<NUM_MOTORS ; i++)
    {
        bus_set(MOTOR_ADDR[i], MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
    }
    pause(ONE_SEC);
    bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);
}
```

The computer simply updates the global parameter values (amplitude between 0° and 60° and frequency ≤ 1 Hz) Note that we also included a body_offset parameter to allow for turning and a phase lag between the tail segments to see if we could increase our swimming efficiency at a given frequency. The following PC code snippet allows the user to easily update the parameter values in real time:

```
/* Set mode to sine mode */
regs.set_reg_b(0, 2);

/* Initialize parameters */
regs.set_reg_b(1, 0);
regs.set_reg_b(2, 0);
regs.set_reg_b(3, 0);

while (true)
{
    /* Interface */
    float param;
    char c;
    cout << "> ";
    cin >> c;
    if (c == 'f') /* Frequency */
```

```
17        {
18            cin >> param;
19            uint8_t param8 = ENCODE_PARAM_8(param, 0, 1);
20            regs.set_reg_b(1, param8);
21        }
22        else if (c == 'a')  /* Amplitude */
23        {
24            cin >> param;
25            uint8_t param8 = ENCODE_PARAM_8(param, 0, 32);
26            regs.set_reg_b(2, param8);
27        }
28        else if (c == 'l')  /* Phase lag */
29        {
30            cin >> param;
31            uint8_t param8 = ENCODE_PARAM_8(param, 0, 360);
32            regs.set_reg_b(3, param8);
33        }
34        else if (c == '0')  /* Stop the robot */
35            regs.set_reg_b(0, 0);
36    }
```

8 Using the tracking system

In order to test our robot, we use a specialized tracking system found in the Biorobotics Laboratory. The system tracks LEDs in a 6×1.5 m pool using two cameras. In order to test the system with our robot, we implemented a simple program which changes the color of the LED according to the position in the aquarium. The red component of the LED depends on the x coordinate, while the blue component depends on the y coordinate. The green component is kept at 128. We checked that the colors gradually transitioned through the course and verified the colors at the corners of the aquarium: *green* at $(0,0)$, *yellow* (red+green) at $(6,0)$, *cyan* (green+blue) at $(0,1.5)$, and *white* (red+green+blue) at $(6,1.5)$.

9 Swimming trajectory generator

For simplicity, we decided to only focus on the body degrees of freedom and kept the pectoral fins still. We also imposed the same amplitude and offset for the two body elements.

Many vertebrates, such as the lamprey, swim in the water by propagating a sinusoidal wave along their body. We chose to do the same with our two degrees of freedom: we introduced a phase lag, ϕ, between the central DoF and the caudal fin DoF.

This gives us four parameters to control the motion of the robot:

- Amplitude R (common to the two DoFs)
- Frequency f
- Offset X (common to the two DoFs)
- Phase lag ϕ

The angular position of the central DoF and caudal fin DoF, represented by θ_1 and θ_2 respectively, are given by equation 1.

$$\begin{aligned} \theta_1 &= R\cos\left(2*\pi*f*t\right) + X \\ \theta_2 &= R\cos\left(2*\pi*f*t-\phi\right) + X \end{aligned} \tag{1}$$

Intuitively, the offset parameter X can be used to steer the robot (see [3]). We use it to stabilize the trajectory of the robot in the aquarium using a proportional (P) controller.

10 Proportional controller

The aquarium has a size of approximately 6.0×1.5 m. We therefore typically want the robot to swim in the x direction. Unfortunately, due to open-loop control, the robot often drifts and hits the side walls, making good velocity measurements difficult. In order to alleviate this problem, we decided to implement a closed-loop control to keep the robot along the middle of the pool ($y_0 = 0.75$ m) using the tracking system. We do this using a simple proportional controller that adjusts the offset (steers) based on the position of the robot relative to the center:

$$X = K(y - y_0) \tag{2}$$

where K is the proportional gain. Figure 4 show how K influences the trajectory, for $K = 0°$, $K = 30°$ and $K = 60°$.

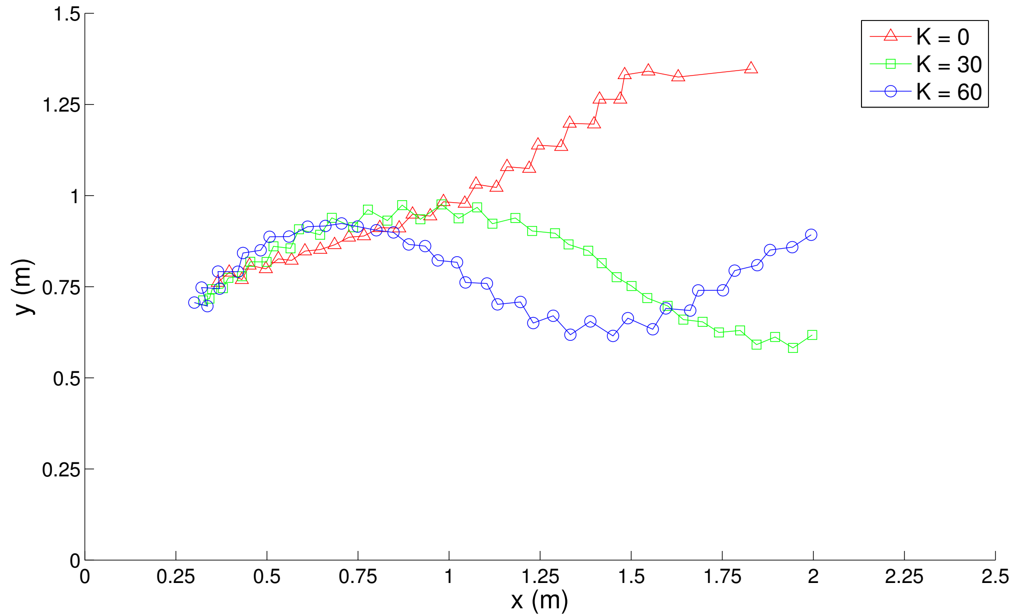

Figure 4: Influence of K on the trajectory.

For $K = 0$, the controller is deactivated, and the error increases over time. We see that for $K > 0$, the absence of a derivative term induces overshooting which is why the robot oscillations around the desired y-position. The oscillation is almost two times faster for $K = 60$ than for $K = 30$, which is coherent. While P controller is far from perfect, it at least allows the robot to do swim across the entire aquarium without hitting the side walls.

11 Experiments

In order to better understand how our control parameters (in particular the frequency and the phase lag ϕ) affect the swimming ability of the robot, we performed a few quick experiments. First, we examined how the oscillation frequency influences the velocity of our robot. We fixed an amplitude of $R = 32°$ and a phase lag of $\phi = 90°$, and measure the total displacement over a 5 second period[1]. For each frequency, we repeated the experiments 5 times. The results are presented in Figure 5. As expected, the speed of the robot increases, almost linearly, with the frequency. For $f = 1$ Hz, we obtain an average speed of 9.3 cm/s.

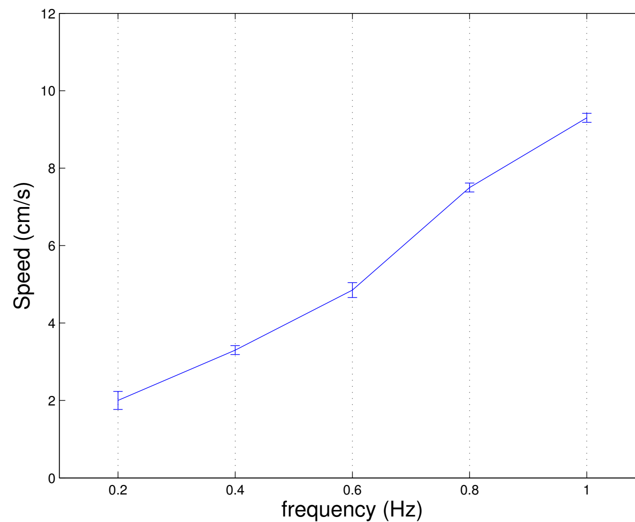

Figure 5: Influence of f on the speed of the robot.

Then, we fixed the frequency to 1 Hz, and measured the velocity for different phase lags, from 0° to 180°. The results are reported in Figure 6. For phase lags lower than 90°, the wave is propagating from the body to the caudal fin, which is more biologically relevant. We observe that the velocity is much higher in this range than for $\phi \in [90, 180]$, where

[1]We begin the measurement only when the speed and trajectory are stabilized.

the wave is propagating from the caudal fin to the body. The speed first increases, then decreases dramatically with phase lag. The best result we saw was with $\phi = 60°$ (and not 90°, as we might expect). This is probably due to the shortness and limited DoF of our robot.

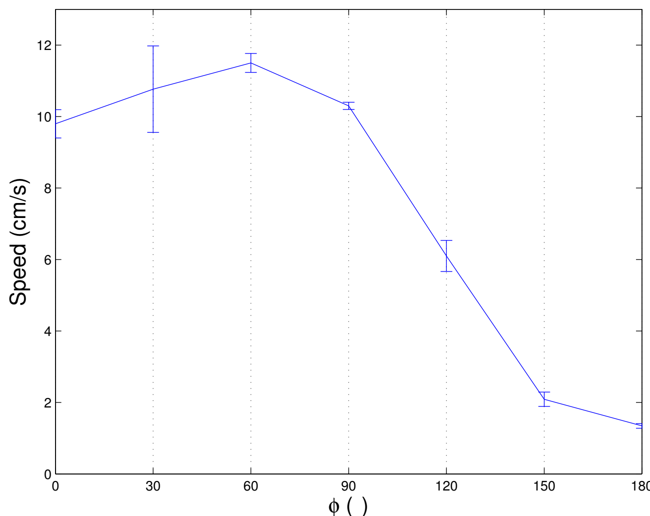

Figure 6: Influence of the phase lag ϕ on the speed of the robot.

12 Conclusion

The simple controller we wrote allows us to generate a good locomotion pattern. This sine-based controller, which represents a propagating wave from the body to the caudal fin, is also the locomotion used by a lot of vertebrates, like the lamprey [4] and the salamander [5]. However, our controller is still very simple, and, in real applications, may not be very robust against perturbations. A more robust and biologically relevant implementation would use Central Pattern Generators (CPGs) [6], which are neural circuits generating rythmic output patterns. This approach is very convenient to control modular robots, even when they are not biologically inspired [7].

References

[1] Crespi, A., Karakasiliotis, K., Knuesel, J., & Ijspeert, A. J. (2011). *Characterization of a salamander-like robot* (pp. 1-3). International Workshop on Bio-Inspired Robots.

[2] Crespi, A. (2007). *Design and Control of Amphibious Robots with Multiple Degrees of Freedom.* Thesis, École Polytechnique Fédérale de Lausanne.

[3] Lachat, D. (2005). *BoxyBot, the fish robot: Design and realization*. Semester project, École Polytechnique Fédérale de Lausanne.

[4] A. Crespi, A. Badertscher, A. Guignard, and A.J. Ijspeert. *Swimming and Crawling with an Amphibious Snake Robot*. In Proceedings of the 2005 IEEE International Conference on Robotics and Automation (ICRA 2005), pages 3035–3039, 2005.

[5] Auke Jan Ijspeert, Alessandro Crespi, Dimitri Ryczko, and Jean-Marie Cabelguen. *From swimming to walking with a salamander robot driven by a spinal cord model*. Science, 315(5817):1416–1420, 2007.

[6] Auke Jan Ijspeert. *Central pattern generators for locomotion control in animals and robots: a review*. Neural Networks, 21(4):642–653, 2008.

[7] Soha Pouya, Jesse van den Kieboom, Alexander Spröwitz, and Auke Ijspeert. *Automatic Gait Generation in Modular Robots: to Oscillate or to Rotate? that is the question*. In Proceedings of IEEE/RSJ IROS 2010, IEEE International Conference on Intelligent Robots and Systems, pages 514–520. Ieee Service Center, 445 Hoes Lane, Po Box 1331, Piscataway, Nj 08855- 1331 USA, 2010.