Series 2 – Braitenberg controllers & FSM

Introduction

Dans cette série, il s'agit de développer des contrôleurs capables produire des comportements dépendant de données d'entrée en temps réel. Cet objectif est atteint en implémentant une boucle de rétroaction où chaque itération récupère les données d'entrée et adapte le comportement du robot en sortie, ce qui influencera les prochaines données d'entrée.

Pour effectuer ces exercices, seuls les capteurs de proximité (au nombre de huit) seront utilisés pour fournir des données d'entrée à interpréter. Le comportement en sortie s'effectue par l'ajustement des vitesses des deux moteurs et par le changement d'état des LEDs.

Développement

Initialement, chaque contrôleur utilise un certain nombre de fonctions de la bibliothèque C de Webots afin de pouvoir notamment initialiser les composants du robot, récupérer ses données d'entrées et ajuster ses comportements.

Afin d'éviter une redondance de code pour effectuer ces mêmes actions dans chaque contrôleur, ainsi que pour ajouter une couche d'abstraction sur l'interface du robot, une petite collection de fonctions utilitaires a été développée dans différents fichiers sources nommés d'après les composants concernés. L'ensemble de ces fichiers se trouve dans un dossier `util`. De cette manière, il est possible d'inclure ces fonctions dans la portée de n'importe quel contrôleur :

```
#include "../util/motors.h"
#include "../util/leds.h"
// etc.
```

Pour pouvoir utiliser ces fonctions utilitaires, le *Makefile* de chaque contrôleur a dû être modifié afin que tous les fichiers sources nécessaires soient compilés. Il est d'ailleurs nécessaire d'avoir un compilateur supportant le standard C99.

Pour le moment, seul un nombre minimal de fonctions ont été développées car elles sont suffisantes pour réaliser les exercices de cette série. Pour les séries suivantes, de nouvelles fonctions pourront certainement apparaitre afin de résoudre de nouveaux problèmes.

1 Proximity Sensor Values

Le contrôleur correspondant à cet exercice est `S02_Distance_Measurements`. Le code source de celui-là se trouve dans le fichier `S02_Distance_Measurements.c`.

1.1 Mise en place

L'objectif de cet exercice est de produire une représentation visuelle des valeurs des capteurs de proximité en fonction de la distance du robot par rapport à un mur. Puisque le robot ne peut pas directement mesurer et fournir des données relatives à sa position, on se contentera d'utiliser des valeurs de temps : celles-ci étant linéairement dépendantes à la distance parcourue, l'évolution des valeurs des capteurs sera donc identique.

L'enregistrement des valeurs ne se fera que sur les deux capteurs avants, définis dans le tableau `USED_SENSORS` :

```
#define USED_SENSORS_COUNT 2
const size_t USED_SENSORS[USED_SENSORS_COUNT] = {0, 7};
```

Le déroulement de l'expérience se fait en trois phases : calibration des capteurs, placement contre un mur et mise en mouvement.

1.1.1 Calibration des capteurs

Dans cette phase, le robot va recueillir et mémoriser les valeurs des capteurs alors qu'il est éloigné de tout obstacle. D'après la documentation, le robot détecte des obstacles à une distance maximale de 4 cm, valeur à laquelle il faudra, à l'exécution, éloigner les potentiels obstacles qui pourraient engendrer une mauvaise calibration.

Concrètement, le robot patiente 5 *steps*, enregistre les valeurs des capteurs pendant les 50 *steps* suivants, puis stocke la moyenne de chaque capteur dans un tableau contenant les valeurs de correction.

Cette calibration s'effectue lors de l'appel à la fonction `sensors_init()`. L'obtention de la valeur d'un capteur se fait grâce à la fonctoin `sensors_get_value(index, correct)`, où *index* est le numéro du capteur (entre 0 et 7) et *correct* une valeur booléenne qui définit si la valeur doit être corrigée ou non par la calibration. Ces deux fonction sont déclarées dans `util/sensors.h`.

Durant cette phase de calibration, toutes les LEDs clignotent. Lorsqu'elles cessent de clignoter, cela signifie que la calibration est terminée.

1.1.2 Placement contre un mur

Ici, il s'agit simplement de placer manuellement le robot contre le mur. La difficulté se trouve dans le fait que la calibration doit être déjà effectuée : il ne faut donc pas que le robot se mette en mouvement directement après la fin de la calibration.

Pour résoudre ce problème, la fonction `void wait_for_wall()` a été développée dans le but de mettre le robot en pause tant qu'il ne capte aucun obstacle. La constante `WAIT_DURATION` est définie à 3 secondes et représente le temps durant lequel le robot doit capter un mur avant de s'activer. Cela empêche le robot de se mettre instantanément en mouvement immédiatement après une détection de mur. La constante `WAIT_FOR_DISTNACE` a été définie à 2000 et représente la valeur minimale que les capteurs doivent mesurer pour que le robot soit considéré comme collé au mur.

```
void wait_for_wall()
{
    printf("Please place the robot close to a wall\n");

    const static unsigned init_countdown = WAIT_DURATION * 1000
       / TIME_STEP;
    unsigned countdown = init_countdown;

    leds_set(true);

    while(wb_robot_step(TIME_STEP) != -1)
    {
        for(size_t i = 0; i < USED_SENSORS_COUNT; ++i)
            if(sensors_get_value(USED_SENSORS[i], true) <
               WAIT_FOR_DISTANCE)
                countdown = init_countdown;

        if(--countdown == 0)
```

```
            break;
    }

    leds_set(false);
}
```

Durant cette phase, les LEDs sont toutes allumées. Elles s'éteignent lorsque le robot détecte le mur et se met en mouvement.

1.1.3 Mise en mouvement

Le robot commence à s'éloigner du mur à vitesse constante. À chaque itération, il enregistre dans un fichier CSV les valeurs des capteurs avant corrigées par la calibration, et dans un second fichier CSV les valeurs non corrigées. Le contrôleur termine son exécution lorsqu'un certain temps s'est écoulé après la mise en mouvement. Cette durée est définie par la constante `MOVE_DURATION` et vaut 5 secondes. La vitesse des roues est fixée à -2 rad/s.

```
FILE * calib = csv_create("calib.csv");
FILE * uncalib = csv_create("uncalib.csv");

if(calib == NULL || uncalib == NULL)
    return EXIT_FAILURE;

const double start_time = wb_robot_get_time();

while(wb_robot_step(TIME_STEP) != -1)
{
    motors_set_speed(SPEED, SPEED);

    double current_time = wb_robot_get_time() - start_time;

    csv_add_row(calib, current_time, true);
    csv_add_row(uncalib, current_time, false);

    if(current_time >= MOVE_DURATION)
        break;
}

fclose(calib);
fclose(uncalib);
```

Dans cet extrait de code, la fonction `csv_create(filename)` renvoie un pointeur vers un nouveau fichier CSV contenant déjà un entête. La fonction `csv_add_row(file, time, correct)` ajoute une ligne dans le fichier *file* et, si *correct* vaut `true`, corrige la valeur des capteurs par la calibration.

1.2 Exécution

L'expérience s'est déroulée avec un vrai robot (modèle n° 3480) en laboratoire. Comme on peut le voir dans la figure 1, les valeurs non calibrées sont en réalité trop hautes (entre 60 et 80 lorsque le robot n'est proche d'aucun obstacle).

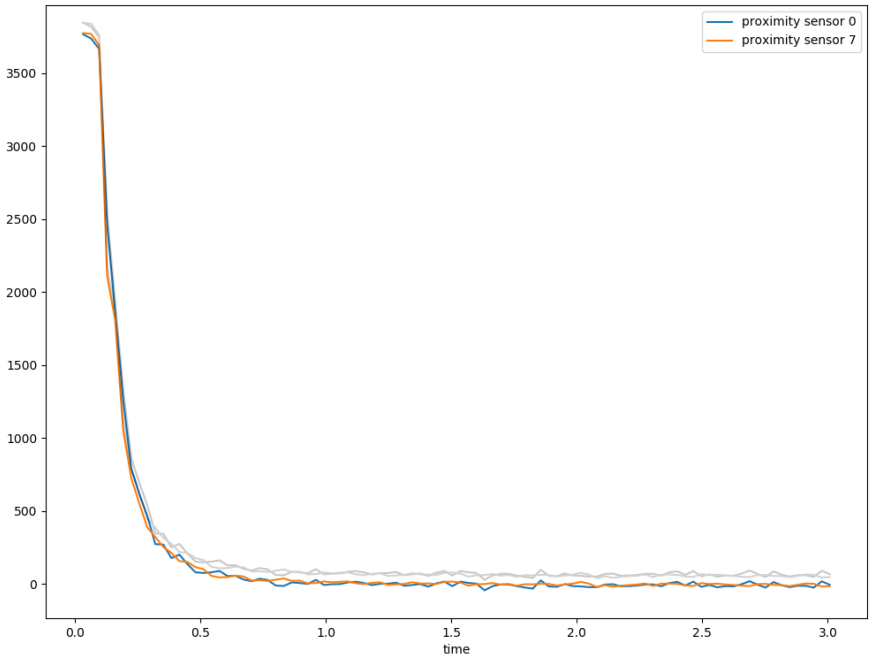


FIGURE 1 – Valeurs des capteurs calibrées (en gris : valeurs non calibrées)

On remarque également que la plupart des valeurs sont atteintes dans un intervalle de temps très court (environ 0,3 secondes), ce qui signifie que les capteurs sont principalement sensibles lorsque le robot est très près d'un obstacle.

Si l'on observe le graphe de réponse infrarouge fourni dans la documentation, l'évolution est assez semblable au graphe de la figure 1. On voit effectivement que la plupart des valeurs sont atteintes lorsque l'obstacle est très proche car les capteurs sont très sensibles à moins de 2 cm. Quant au bruit, le graphe de la figure 1 semble assez fidèle, bien que le bruit semble plus prononcé lorsqu'aucun obstacle n'est proche.

2 Explorer & Advanced Lover

Les contrôleurs correspondant à cet exercice sont `S02_Explorer` et `S02_Lover`. Le code source de ceux-là se trouve dans les fichiers `S02_Explorer.c` et `S02_Lover.c`.

2.1 Mise en place

La phase de calibration est pareille à celle de l'exercice précédent. Une fois cette phase terminée, le robot devient parfaitement indépendant et se déplace suivant qu'il soit un « explorer » ou un « lover ».

2.2 Comportement « explorer »

Les différents poids de chaque capteur ont été fixés plus ou moins arbitrairement, mais de sorte à ce que les capteurs avant soient les plus influent : en effet, si le robot vient à rencontrer un obstacle se trouvant juste en face de lui, il doit tourner et fuir bien plus vite que si l'obstacle se trouve sur le côté ou derrière lui.

La vitesse initiale des roues est fixée à 4 rad/s et est définie par la constante `SPEED`. Le seuil de détection des capteurs est est fixé à 400 et est défini par la constante `THRESHOLD`.

```
static const double total_weight = 10;
static const double weights[SENSORS_COUNT] =
        {4, 3, 2, 1, 1, 2, 3, 4};

while(wb_robot_step(TIME_STEP) != -1)
{
    double prox[2] = {0, 0};
    for(size_t i = 0; i < SENSORS_COUNT; ++i)
    {
        size_t index = i * 2 / SENSORS_COUNT;
        prox[index] += weights[i] * sensors_get_value(i, true) /
            total_weight;
    }

    double speed_right = SPEED - prox[0] / THRESHOLD * SPEED;
    double speed_left = SPEED - prox[1] / THRESHOLD * SPEED;

    motors_set_speed(
            clamp(speed_right, -MAX_SPEED, MAX_SPEED),
            clamp(speed_left, -MAX_SPEED, MAX_SPEED));
}
```

2.3 Comportement « lover »

Le code pour ce comportement est essentiellement identique à celui pour le comportement « explorer ». La différence se trouve notamment dans les poids où le raisonnement a été inversé : cette fois, si l'obstacle ne se trouve pas en face du robot, il doit tourner et s'y diriger bien plus vite que si l'obstacle se trouve déjà devant.

```
static const double weights[SENSORS_COUNT] =
        {4, 3, 2, 1, 1, 2, 3, 4};
```

Pour finir, les vitesses de roues ont simplement été échangées afin d'attirer le robot plutôt que de le faire fuir.

```
motors_set_speed(
        clamp(speed_right, -MAX_SPEED, MAX_SPEED),
        clamp(speed_left, -MAX_SPEED, MAX_SPEED));
```

3 Exploring Lover

Le contrôleur correspondant à cet exercice est `S02_State_Machine`. Le code source de celui-là se trouve dans le fichier `S02_Distance_Measurements.c`.

3.1 Mise en place

La phase de calibration est pareille à celle de l'exercice précédent. Une fois cette phase terminée, le robot devient parfaitement indépendant et se déplace suivant en alternant le comportement « explorer » et « lover ».

Cet exercice consiste essentiellement à combiner les deux comportements réalisés dans l'exercice précédent. Un nouvel élément entre en jeu : une variable d'état qui peut changer durant l'exécution, définissant le comportement actuel du robot : « explorer » ou « lover ».

Le robot démarre dans l'état « lover » et s'approche des obstacles. Lorsqu'il est suffisamment proche, son comportement devient « explorer » et il s'en éloigne. Une fois éloigné, il redevient un « lover ».

3.2 Problèmes

Cependant, une implémentation naïve risque bien de ne pas bien fonctionner. En effet, si le robot passe du comportement « lover » à « explorer », il risque de détecter à nouveau immédiatement l'obstacle qu'il vient de quitter, et donc de rester bloquer dans une zone limitée en alternant rapidement d'un état à l'autre.

Pour éviter ce problème, une seconde variable est utilisée : un compteur qui s'assure que le robot ne change pas trop rapidement d'état. Ainsi, lorsque le robot passe du comportement « lover » à « explorer », il doit patienter au moins 2,5 secondes avant de pouvoir redevenir un « lover ». En pratique, cela a pour effet de permettre au robot d'être moins « attaché » à un obstacle et de se déplacer plus librement.

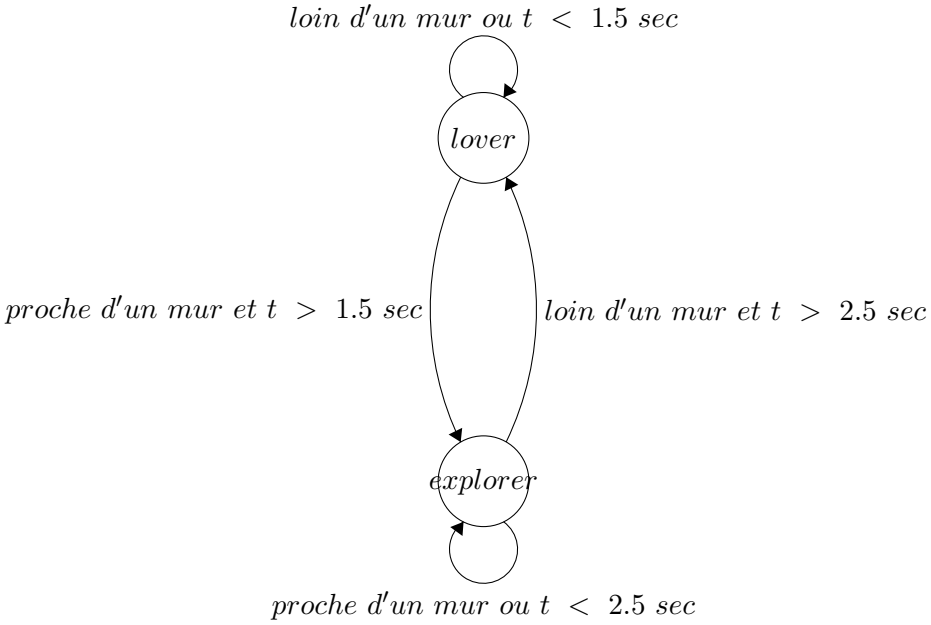
Un second problème est alors apparu : le robot réel avance plus lentement dans le temps que la simulation. Il est donc nécessaire d'introduire un facteur de temps en fonction du mode du robot (simulation ou réel).

3.3 Implémentation

```
static const double lover_total_weight = 11;
static const double lover_weights[SENSORS_COUNT] =
        {2, 2, 3, 4, 4, 3, 2, 2};

static const double explorer_total_weight = 10;
static const double explorer_weights[SENSORS_COUNT] =
        {4, 3, 2, 1, 1, 2, 3, 4};

const double time_factor = wb_robot_get_mode() == 0 ? 1 : 0.4;
printf("time factor = %f\n", time_factor);

const unsigned max_counter = MIN_STATE_DURATION * 1000 /
   TIME_STEP * time_factor;
unsigned counter = max_counter;

enum { LOVER, EXPLORER } state = LOVER;

while(wb_robot_step(TIME_STEP) != -1)
{
    double distance =
            (sensors_get_value(0, true) + sensors_get_value(7,
               true)) / 2;

    if(++counter >= max_counter)
    {
```

```
        if(state == LOVER && distance > LOVER_DISTANCE_THRESHOLD
           )
        {
            state = EXPLORER;
            counter = 0;
            printf("switching to explorer\n");
        }

        else if(state == EXPLORER && distance <
           EXPLORER_DISTANCE_THRESHOLD)
        {
            state = LOVER;
            counter = max_counter * 3 / 4; // shorter timer
            printf("switching to lover\n");
        }
    }

    double prox[2] = {0, 0};
    for(size_t i = 0; i < SENSORS_COUNT; ++i)
    {
        double weight = state == LOVER ? lover_weights[i] :
           explorer_weights[i];
        double total_weight = state == LOVER ?
           lover_total_weight : explorer_total_weight;
        prox[i * 2 / SENSORS_COUNT] +=
                weight * sensors_get_value(i, true) /
                   total_weight;
    }

    double speed_right = SPEED - prox[0] / THRESHOLD * SPEED;
    double speed_left = SPEED - prox[1] / THRESHOLD * SPEED;

    motors_set_speed(
            clamp(state == LOVER ? speed_left : speed_right, -
               MAX_SPEED, MAX_SPEED),
            clamp(state == LOVER ? speed_right : speed_left, -
               MAX_SPEED, MAX_SPEED));

    leds_set(state == LOVER);
}

leds_set(false);
```

3.4 Schéma de l'automate fini



3.5 Vidéo

https://youtu.be/v0Y7L28oFAo