

SÉBASTIEN CABANA et YOHAN LEFEBVRE

Projet d'intégration en sciences informatiques et mathématiques

420-204-RE, groupe 00101

Rapport synthèse

Travail présenté à

M. Robert TURENNE

Département d'informatique

Cégep de Saint-Jérôme

Le 28 mai 2023

Table des matières

Phase préliminaire.....	3
Hacking de la trottinette Gotrax GLX V2	3
Étude du fonctionnement de la gâchette de vitesse	3
Reproduction du comportement de la gâchette.....	5
Contrôle du moteur avec un Arduino.....	5
Dérivation du circuit embarqué de la trottinette.....	8
Interface physique.....	8
Programmation des boutons.....	9
Montage temporaire des boutons sur la trottinette.....	10
Changement des boutons pour un joystick muni d'un bouton.....	12
Programmation et choix de l'écran	13
Voyants lumineux.....	14
Construction d'un premier prototype	14
Modifications permettant un branchement rapide	15
Développement des fonctions adaptatives et de ses composantes.....	15
Développement du speedomètre	15
Méthode choisie pour maintenir la vitesse définie.....	19
Développement du détecteur de distance.....	20
Ajustement de la vitesse en fonction des obstacles par notre système	22
Complexité du freinage automatisé.....	23
Alternative au freinage automatisé.....	23
Fin du projet	23
Évolution du programme.....	24
Version 1 : premier montage.....	25
Version 2 : joystick, écran LCD, LEDs et contrôle du moteur avec un Arduino	25
Version 3 : speedomètre et vitesse réelle au lieu d'une valeur analogue	25
Version 4 : allègement de la fonction loop()	25
Version 5 : simplification du code en abandonnant certaines conventions de la programmation orientée objet	25
Version 6 : implémentation du LiDAR	26
Version 7 : méthodes adaptatives ajustant la vitesse en fonction des obstacles.....	26

Phase préliminaire

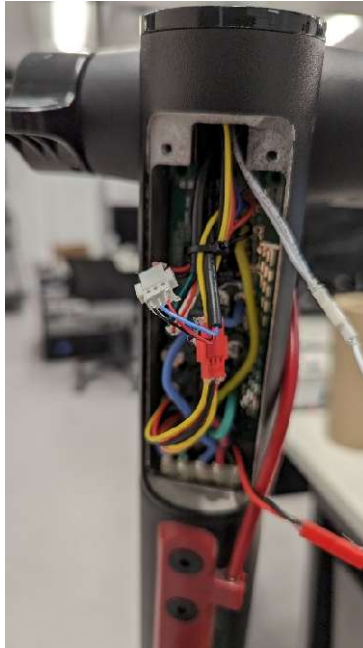
Dans le cadre de notre projet synthèse pour nos études en sciences informatiques et mathématiques, nous avons décidé de construire et d'installer un Cruise control adaptatif, un speedomètre utilisant un capteur à effet Hall ainsi qu'une interface physique constituée de boutons de contrôle et d'un écran LCD sur une trottinette électrique. Nous voulions initialement construire une trottinette électrique de A à Z, mais la contrainte de temps à laquelle nous faisons face nous a forcé à utiliser un modèle préfabriqué à la place. Nous avons choisi la Gotrax GXL V2 puisqu'un membre de l'équipe l'avait déjà en sa possession. La majeure partie de notre projet était d'analyser et de comprendre son fonctionnement afin de la contrôler avec notre propre système électronique. Il serait toutefois possible de reprendre notre idée initiale et d'adapter l'interface que nous avons construite spécifiquement pour ce modèle afin qu'il contrôle une trottinette faite maison ou un autre modèle de trottinette électrique.

Hacking de la trottinette Gotrax GLX V2

Afin de mener notre projet à bien, nous devons avant tout comprendre avec détail le fonctionnement de la trottinette et comment le signal envoyé par la gâchette de vitesse contrôlait la puissance du moteur. Notre but était d'introduire au système embarqué de la Gotrax GLX V2 un microcontrôleur permettant de dériver le signal habituel et de le contrôler à notre guise. Il deviendrait alors possible de réguler la vitesse en programmant ce microcontrôleur pour qu'il ajuste la puissance du moteur en fonction de la vitesse de la trottinette et, éventuellement, des obstacles.

Étude du fonctionnement de la gâchette de vitesse

Tout d'abord, nous avons ouvert le système embarqué de la trottinette pour y découvrir et analyser son système de filage. Nous voulions à ce moment comprendre comment la gâchette de vitesse communiquait avec le moteur.



Nous avons d'abord mesuré, à l'aide d'un multimètre, la résistance électrique entre les fils de ce que nous pensions être la gâchette et le contrôleur. Nous recommandons à quiconque ayant un projet dont l'étude d'un circuit embarquée est de mise de vérifier que les fils utilisés lors des mesures sont les bons. Nous avons en effet utilisé ceux associés aux freins plutôt qu'à la gâchette. Cette erreur nous a coûté beaucoup de temps. Comme nos résultats étaient peu concluants, nous avons conclu que la gâchette n'était pas un potentiomètre et que le signal n'était pas une variation de résistance. À l'aide d'un multimètre muni d'une fonction de logique, nous avons aussi cherché à déterminer si le signal de la gâchette était digital ou analogue : sans succès. Nous avons donc démonté la gâchette afin de connaître sa nature et avons déterminé qu'il s'agit d'un capteur à effet Hall muni de deux aimants. Ce type de dispositif permet de mesurer la présence et l'intensité d'un champ magnétique. Il utilise l'effet Hall, qui est une propriété physique de certains matériaux conducteurs qui se produisent lorsqu'ils sont exposés à un champ magnétique. Le capteur est généralement constitué d'une fine plaque de matériau semi-conducteur, telle que le silicium, qui est placée entre deux bornes électriques. Lorsque le matériau semi-conducteur est soumis à un champ magnétique, les électrons qui se déplacent dans le matériau sont déviés de leur trajectoire initiale par la force magnétique. En conséquence, une tension électrique est créée perpendiculairement au courant et au champ magnétique. Cette tension, appelée tension de Hall, est proportionnelle à l'intensité du champ magnétique et à l'intensité du courant qui traverse le capteur. Le capteur à effet Hall convertit ensuite cette tension en une sortie numérique ou

analogique. À la suite de ces recherches, nous avons donc pu confirmer que la gâchette n'était effectivement pas un potentiomètre, bien que cette hypothèse eût d'abord été émise parce que nous nous étions trompés de fils.

Reproduction du comportement de la gâchette

En connaissant la nature de la gâchette, nous avons maintenant une meilleure idée de comment l'utiliser. Nous avons donc passé à l'étape suivante, qui était de reproduire son comportement pour contrôler la trottinette sans avoir à utiliser la gâchette. À l'aide d'un microcontrôleur Arduino, nous avons tenté de lire la valeur analogue en entrée en branchant la gâchette directement dedans. Les valeurs des premières lectures ne faisaient aucun sens : nous avons des écarts beaucoup trop petits entre les valeurs minimales et maximales lues, ce qui ne permettrait pas de contrôler efficacement la puissance du moteur (la trottinette n'aurait eu que deux vitesses possibles : nulle ou maximale). Nous avons alors émis l'hypothèse qu'une résistance était intégré au circuit embarqué de la trottinette afin d'avoir une plus grande variation entre les valeurs analogues produites par la gâchette. Nous avons donc ajouté une résistance entre la mise à la terre de la gâchette et son signal afin de voir quel effet cela aurait sur les valeurs analogues minimales et maximales. Comme il était impossible de savoir la valeur de la résistance à utiliser, nous en avons testé plusieurs. Voici un tableau présentant les résultats obtenus.

Tableau 1. Valeurs analogues de la gâchette selon différentes résistances

Résistance (Ω)	Valeur analogue minimale lue	Valeur analogue maximale lue
0	999	1000
300	165	790
1 000	354	787
10 000	806	818
100 000	1000	1000

À partir de ces données, nous avons déterminé qu'en ajoutant une résistance de 300Ω à notre circuit, ce problème d'écart trop petit serait réglé. Nous avons d'ailleurs choisi cette valeur parce qu'elle offre la plus grande variation et, par le fait même, le plus de différentes vitesses possibles.

Contrôle du moteur avec un Arduino

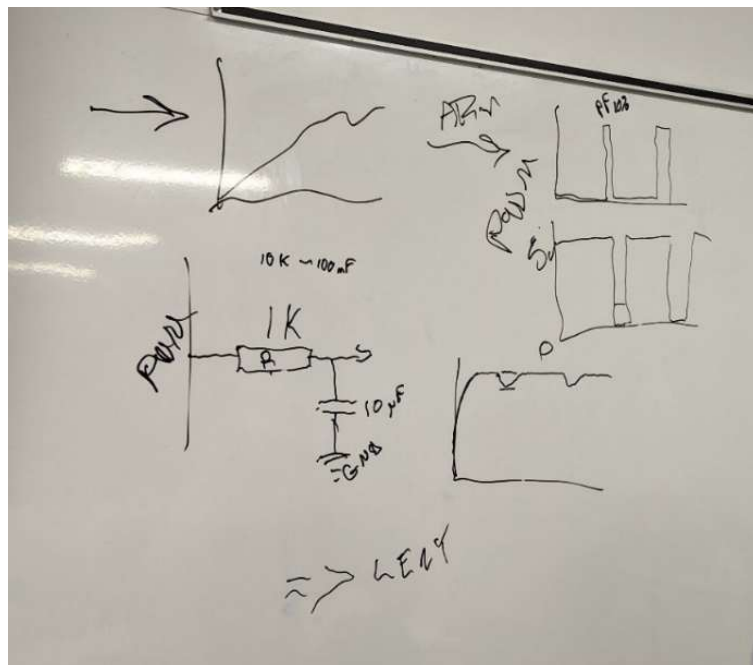
Puisqu'il nous était maintenant possible de lire la valeur d'entrée de la gâchette, il nous fallait alors traiter ces valeurs afin de les communiquer au moteur et ce, de la même manière que le

circuit embarqué le fait. Une fois cette étape surmontée, la coexistence de notre Cruise control avec le circuit embarqué de la trottinette serait complétée. Il nous fallait d'abord trouver quelles étaient les valeurs acceptées par le circuit embarqué. En effet, le système de la trottinette détecte lorsque les données sont incohérentes et affiche un message d'erreur. À l'aide d'un code très simple incrémentant chaque seconde une valeur initialement de 0 envoyée au circuit embarqué, nous avons décelé plusieurs situations où l'erreur se produit. Voici le code utilisé.

```
void setup() {  
  Serial.begin(9600);  
  pinMode(2, INPUT); // button (pause count when pressed)  
  pinMode(9, OUTPUT); // pin PWM  
}  
int value = 0;  
void loop() {  
  if(digitalRead(2) != HIGH) {  
    analogWrite(9, value);  
    Serial.println(value);  
    delay(1000);  
    value++;  
  }  
}
```

À la suite de ce test, nous avons donné un nom à l'erreur la plus fréquente : le non-respect des bornes. Nous entendons par «bornes» les valeurs analogues minimales et maximales envoyées par notre microcontrôleur au moteur. Lorsque celui-ci était branché à un ordinateur, ces valeurs étaient de 9 et 168. Nous avons plus tard découvert que ces valeurs dépendent du voltage fourni au microcontrôleur : elles étaient différentes lorsque l'Arduino était alimenté par la batterie de la trottinette ou lorsque nous avons ajouté une pile de 9V à notre circuit. Nous avons donc ajouté à notre programme des constantes définissant ces bornes en fonction de l'alimentation. Nous avons déterminé les valeurs analogues des bornes par essai-erreur, en augmentant le minimum et en diminuant le maximum jusqu'à ce que le message d'erreur n'apparaisse plus, tout en s'assurant que la vitesse maximale de la trottinette reste la même. Nous avons aussi remarqué que le message d'erreur s'affiche si nous allumons la trottinette alors qu'une valeur valide mais supérieure à sa borne minimale est déjà envoyée au moteur. Par exemple, si le microcontrôleur est alimenté par un ordinateur et que l'on veut lui envoyer la valeur de 100 (qui est comprise entre 9 et 168, donc valide), il faut absolument lui envoyer la valeur de 9 en premier. Cela peut se

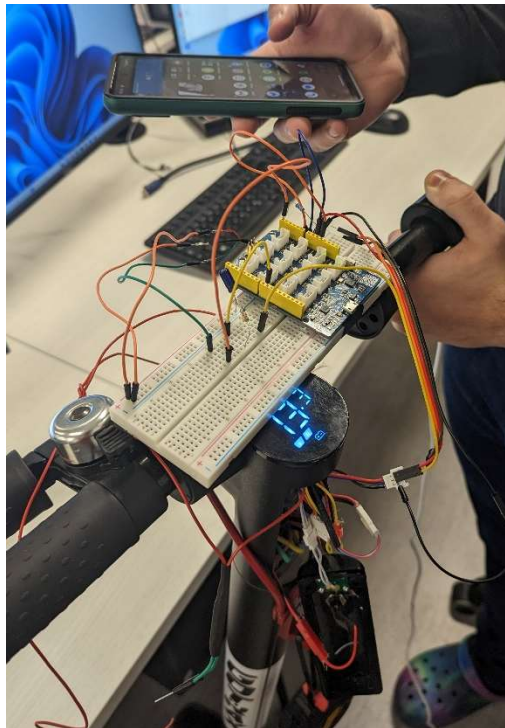
faire au démarrage et suffit à régler le problème. La dernière étape requise au contrôle du moteur fût d'intégrer à notre circuit un filtre RC¹ pour prévenir les coups donnés par le moteur. Ce problème semble être causé par l'utilisation d'une broche PWM pour le contrôle de la puissance du moteur. En effet, ce type de signal est constitué d'une série de créneaux (ou pulses) d'une certaine durée. La largeur de ces pulses, appelée cycle de travail, peut varier en fonction de la puissance requise pour le dispositif. Plus le cycle de travail est long, plus la puissance fournie est importante, tandis qu'un cycle de travail plus court fournit moins de puissance. Or, puisqu'un signal PWM n'est pas constant, le moteur s'activait momentanément avant de faire une pause, puis de s'activer momentanément à nouveau, et ce en boucle. Pour régler ce problème, il nous a suffi d'ajouter à notre montage un circuit RC entre notre microcontrôleur et le moteur de la trottinette. Son rôle est d'envoyer au reste du circuit un courant stable en emmagasinant d'abord le courant qui lui est fourni avant de se décharger graduellement. Il absorbe ainsi les fluctuations créées par la broche PWM et s'assure que le moteur reçoive un courant constant.



¹ M. Horowitz, J. Plummer, R. Howe. RC Filters.

Dérivation du circuit embarqué de la trottinette

Finalement, il ne restait plus qu'à combiner la lecture de la gâchette et le contrôle du moteur avec un Arduino pour compléter l'étape du hacking de la trottinette. À noter que puisque les valeurs d'une lecture analogue sont comprises entre 0 et 1023 et que les valeurs d'une écriture analogue sont comprises entre 0 et 255, il est primordial de convertir la donnée retournée par la gâchette avant de l'envoyer au moteur. Pour ce faire, nous avons utilisé la méthode `map()`². À ce point, nous avons réussi à insérer un microcontrôleur au circuit embarqué de la trottinette tout en la gardant opérationnelle. Cela peut sembler peu, mais une fois cette étape achevée, la partie la plus frustrante et dont nous avions le moins de connaissances était derrière nous. Il deviendra alors plus facile d'avancer et de suivre notre plan de projet initial.



Interface physique

Nous entendons par interface physique le boîtier de contrôle de notre régulateur de vitesse. Il est composé aujourd'hui d'un joystick, de voyants lumineux, d'un écran d'affichage, d'une pile de 9V

² Référence : <https://cdn.arduino.cc/reference/en/language/functions/math/map/>

et d'un bouton. Voici une description de l'évolution de ses composantes, des changements apportés à notre module et des problèmes encourus.

Programmation des boutons

Nous avons initialement décidé d'utiliser cinq boutons disposés en croix pour contrôler l'état de notre régulateur de vitesse. Nous nous sommes inspirés du Cruise control intégré à plusieurs voitures et voulions reproduire ce fonctionnement : l'utilisateur doit d'abord mettre le régulateur en marche en appuyant sur « ON » Il doit ensuite appuyer sur le bouton « SET » afin que la vitesse à laquelle la voiture roulait au moment où le bouton a été appuyé soit maintenue. Il est possible de reprendre le contrôle de la vitesse en appuyant sur l'accélérateur, le frein ou le bouton « CANCEL », et ce tout en gardant la vitesse réglée en mémoire. Le bouton « RESUME » permet de réactiver le Cruise control et de maintenir la vitesse définie auparavant. Lorsqu'il est en marche, il est aussi possible d'augmenter et de réduire la vitesse régler avec le bouton « + » et « - ». En appuyant sur le bouton « OFF », le régulateur se désactive et sa mémoire est effacée. Nous avons réussi à reproduire ce comportement en utilisant de nombreuses conditions imbriquées. Voici le premier code fonctionnel permettant de contrôler les différents états de notre Cruise control décrits ci-haut avec les boutons.

```
// Valeurs fictives
float vitesse_set = 0;
float vitesse_trottinette = 20;

void setup() {
  Serial.begin(9600);

  // Initialize the button inputs:
  pinMode(set_buttonPin, INPUT);
  pinMode(on_off_buttonPin, INPUT);
  pinMode(res_can_buttonPin, INPUT);
  pinMode(plus_buttonPin, INPUT);
  pinMode(minus_buttonPin, INPUT);
}

void loop() {
  // Read if on_off was pressed
  buttonState = digitalRead(on_off_buttonPin);
  if (buttonState == HIGH) {
    if (is_on) { // Turn off
      is_on = false;
      is_activated = false;
      vitesse_set = 0;
    }
    else { // Turn on
      is_on = true;
      is_activated = false;
      vitesse_set = 0;
    }
  }
}
```

```

    }

    // Read next buttons only if on
    if(is_on) {

        // Read if set was pressed
        buttonState = digitalRead(set_buttonPin);
        if (buttonState == HIGH) {
            vitesse_set = vitesse_trottinette;
        }

        // Read next buttons only if a speed was set
        if(vitesse_set > 0) {

            // Read if res_can was pressed
            buttonState = digitalRead(res_can_buttonPin);
            if (buttonState == HIGH) {
                if(is_activated) { // Turn off
                    is_activated = false;
                }
                else { // Turn on
                    is_activated = true;
                }
            }
        }

        // Read next buttons only if activated
        if(is_activated) {

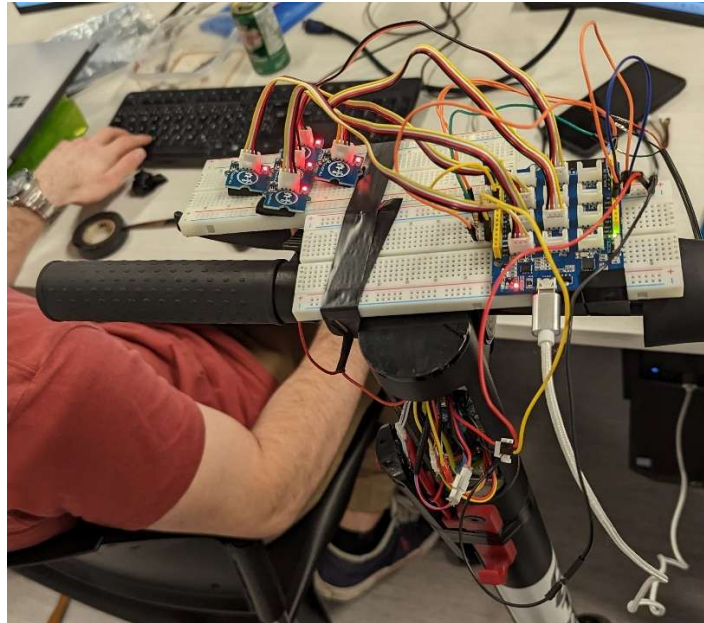
            // Read if + was pressed
            buttonState = digitalRead(plus_buttonPin);
            if (buttonState == HIGH) {
                ++vitesse_set;
            }

            // Read if - was pressed
            buttonState = digitalRead(minus_buttonPin);
            if (buttonState == HIGH) {
                --vitesse_set;
            }
        }
    }
}

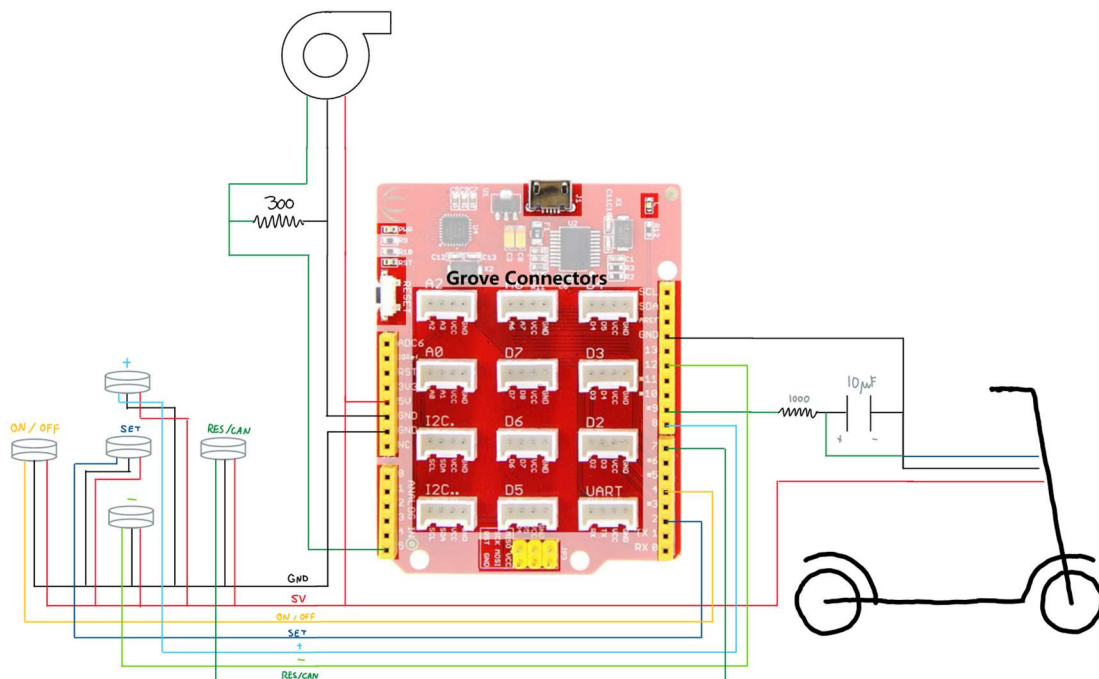
```

Montage temporaire des boutons sur la trottinette

Afin de voir si notre code et la logique de notre programme fonctionnait, nous avons construit un circuit temporaire réunissant les composantes requises au contrôle du moteur (gâchette de la trottinette, filtre RC, résistance de 300Ω, microcontrôleur) et les cinq boutons. Voici de quoi notre projet avait l'air à ce moment.



Puisque le tout fonctionnait bien, nous avons aussi fait le premier schéma résumant le branchement des composantes utilisées jusqu'à présent.



Le premier prototype de notre Cruise control était ainsi opérationnel. Toutefois, les variables associées aux vitesses dans notre programme étaient en réalité des valeurs analogues. Nous étions donc conscients qu'il faudrait éventuellement changer notre logique. Il était cependant

inutile de le faire tant que nous n'avions pas conçu notre speedomètre : il aurait alors été impossible d'effectuer nos tests.

Changement des boutons pour un joystick muni d'un bouton

À la suite d'une première promenade d'essai dans le corridor, nous avons réalisé que l'utilisation des boutons pendant que la trottinette est en mouvement était entravée par les nombreux fils et par les autres boutons. Notre professeur nous a donc recommandé une alternative : remplacer nos boutons par un joystick muni d'un bouton. Nous avons ainsi pu regrouper nos cinq options en n'utilisant qu'une seule composante. Les modifications au niveau de la programmation ont aussi été effectuées. Au lieu de vérifier si un bouton a été appuyé, il faut simplement vérifier si les valeurs des deux axes du joystick ont changé. Les changements apportés sont donc minimes. Il est cependant important de respecter un ordre logique dans les vérifications de l'état du joystick. En effet, contrairement aux boutons dont l'ordre des vérifications importe peu, il faut regarder si le bouton du joystick a été appuyé avant de voir s'il pointe vers une direction quelconque. Autrement, le bouton risque d'être ignoré par moments, ce qui nuit au bon fonctionnement du Cruise control. Voici le code conçu pour déterminer l'état du joystick.

```
// Définition des pins
const int button_Pin = 2;
const int x_Pin = A1;
const int y_Pin = A2;

// État du bouton et des deux axes
int buttonState = 0;
int xValue = 0;
int yValue = 0;

// Variable décrivant l'état actuel du joystick
String state = "ERROR";

void setup() {
  Serial.begin(9600);

  // Initialize the inputs:
  pinMode(button_Pin, INPUT_PULLUP);
  pinMode(x_Pin, INPUT);
  pinMode(y_Pin, INPUT);
}

void loop() {

  // Lecture des valeurs retournées par le joystick
  xValue = analogRead(x_Pin);
  yValue = analogRead(y_Pin);
  buttonState = digitalRead(button_Pin);

  // Vérification si le bouton a été appuyé avant tout
  if(buttonState != HIGH) {
```

```

    state = "PRESS";
}

// Si le bouton n'a pas été appuyé, vérification de la direction où le joystick
pointe
else {
    // Up
    if((xValue <= 1000 && xValue >= 500) && (yValue <= 500 && yValue >= 0)) {
        state = "UP";
    }
    // Down
    else if((xValue <= 1000 && xValue >= 500) && (yValue >= 1000)) {
        state = "DOWN";
    }
    // Right
    else if((xValue >= 1000) && (yValue <= 1000 && yValue >= 500)) {
        state = "RIGHT";
    }
    // Left
    else if((xValue <= 500 && xValue >= 0) && (yValue <= 1000 && yValue >= 500)) {
        state = "LEFT";
    }
    else {
        state = "x";
    }
}

if(state != "x")
Serial.println(state);
delay(100);
}

```

Programmation et choix de l'écran

Comme nous avons déjà un écran inclus avec le Grove Beginner Kit remis au début de la session, nous avons au départ prévu utiliser celui-ci pour notre interface physique. Puisque sa taille de 0,96" n'était pas assez grande à notre goût, nous pensions en utiliser plusieurs. Cette idée est vite tombée à l'eau puisque nous avons découvert, au fil de nos recherches sur le fonctionnement des écrans OLED SSD1315 à notre disposition, que les composantes utilisant un port i²c sont accessibles via leur adresse respective plutôt qu'à une broche assignée. Cette complexité accrue a finalement corroboré la pertinence d'utiliser un écran plus grand à la place. Une autre raison de ne pas utiliser ce modèle d'écran a été découverte après l'avoir ajouté à notre prototype : le rafraîchissement de l'écran nécessaire au bon fonctionnement de notre Cruise control (il faut pouvoir afficher la vitesse en temps réel, donc réduire la fréquence de rafraîchissement n'est pas envisageable) ralentissait énormément notre programme. Afin de régler tous ces problèmes en même temps, nous avons finalement opté pour un écran LCD muni de rétroéclairage RGB. Le seul problème rencontré lors de l'utilisation de ce type d'écran a été lorsque nous alimentions notre circuit électronique avec la batterie de la trottinette plutôt qu'avec un ordinateur : le courant

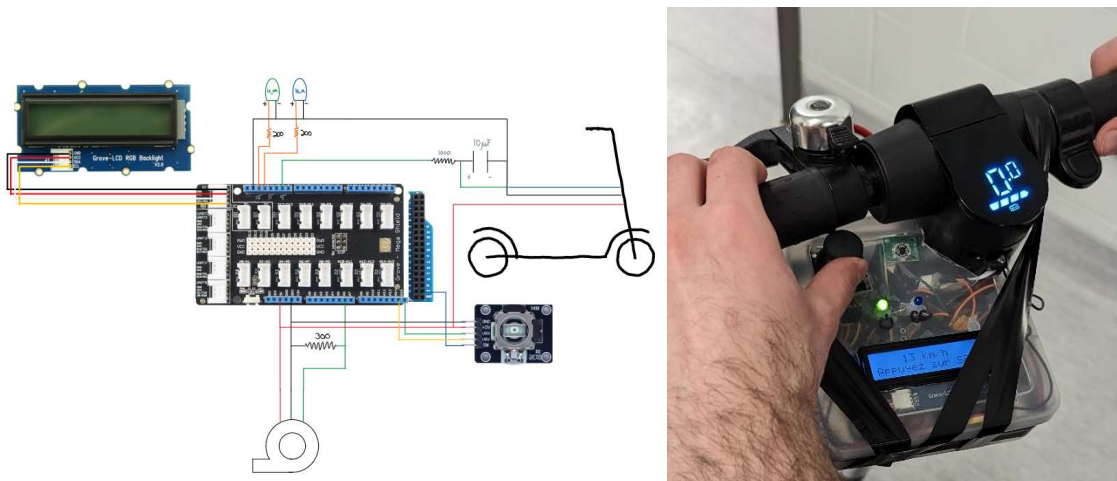
étant insuffisant, l'écriture normalement opaque devenait très pâle et impossible à lire si la trottinette était en mouvement. Pour régler ce problème, il nous a suffi d'ajouter une batterie de 9V pour alimenter notre microcontrôleur au lieu d'utiliser la batterie de la trottinette comme source d'énergie. À noter que les bornes définies dans la section « Hacking de la trottinette Gotrax GLX V2 » ont aussi été ajustées afin d'éviter les messages d'erreur.

Voyants lumineux

Comme notre écran n'a que deux lignes et que nous devons afficher la vitesse de la trottinette ainsi que la vitesse réglée, nous avons décidé d'utiliser de petits voyants lumineux pour montrer l'état du régulateur de vitesse plutôt que de l'afficher à l'écran. Ainsi, si l'ampoule LED verte est allumée, cela veut dire que le Cruise control est à ON. Sinon, il est OFF. Si l'ampoule LED bleue est allumée, cela veut dire que le Cruise control est présentement en marche. Sinon, il est en pause. À noter qu'une résistance de 220Ω doit être utilisée afin de protéger les ampoules.

Construction d'un premier prototype

Après avoir développé de manière indépendante toutes les composantes de l'interface physique, il était maintenant temps de les assembler ensemble. Nous avons ainsi découpé un boîtier en plastique de sorte à soutenir nos éléments tout en cachant le plus possible les nombreux fils. Nous avons d'ailleurs changé le microcontrôleur à ce moment, optant pour un Arduino Mega plutôt qu'un Arduino Uno afin de ne pas manquer de broches ou de mémoire. Un nouveau schéma tenant compte des modifications et des nouveaux ajouts à l'interface physique a aussi été réalisé à cette étape.



Modifications permettant un branchement rapide

Une fois l'interface physique terminée, un cours fût consacré à la rendre plus facilement démontable. En effet, avec le début du beau temps, le propriétaire de la trottinette électrique tenait à la récupérer pour s'en servir pour aller travailler. Le bouton d'alimentation de la trottinette fût donc retiré afin de permettre d'y passer les fils nécessaires à l'ajout de notre circuit électronique tout en protégeant son système embarqué.



Développement des fonctions adaptatives et de ses composantes

Notre Cruise control étant maintenant fonctionnel, il nous fallait implémenter à notre système les fonctions lui permettant de suivre un obstacle tout en ajustant sa vitesse pour maintenir une distance sécuritaire.

Développement du speedomètre

À ce point du projet, nous utilisons toujours la valeur analogue de la gâchette au moment où le bouton « SET » était appuyé plutôt qu'une vraie vitesse. Sur un terrain plat, cela ne pose pas de problème puisqu'en envoyant constamment la même donnée au moteur, la vitesse initiale est maintenue. Cependant, nous voulions que la vitesse s'adapte automatiquement à l'environnement, notamment que la trottinette accélère en montant une pente et décélère en la descendant. Pour se faire, il nous fallait ajouter un speedomètre afin de connaître à tout moment

la vitesse réelle de la trottinette. Nous avons donc ajouté sur la roue avant un capteur à effet Hall branché à notre Arduino et un petit aimant qui est détecté à chaque révolution de la roue. Rappelons qu'un capteur à effet Hall, la composante présente dans la gâchette de la trottinette, permet de détecter les changements de champ magnétique. Dans le cas de notre speedomètre, ce changement se produit lorsque l'aimant collé à la roue passe devant le capteur.



Notre microcontrôleur calcule ensuite le nombre de tours effectués en fonction du temps, ce qui peut par la suite être converti en vitesse en tenant compte de la circonférence de notre roue. En effet, en connaissant toutes ces données, on peut déduire la distance parcourue par la roue (pour chaque tour complet, la trottinette s'est déplacée d'une fois sa circonférence) en un certain laps de temps, ce qui nous donne une vitesse en l'intégrant. Voici la formule utilisée³ :

$$vitesse = \frac{\text{distance parcourue}}{\text{nombre de révolutions}} \times \frac{1000}{\text{temps écoulé}} \Leftrightarrow v = \frac{\Delta x}{n} \times \frac{1000}{\Delta t}$$

Il fallait toutefois trouver un moyen d'utiliser ces éléments théoriques relatifs à la physique mécanique dans notre programme. Nous avons donc élaboré l'ébauche suivante.

```
// Méthode indépendante de la fonction loop() permettant de compter le nombre de tours
// pendant l'exécution du code (évite les pertes de données)
void detecter_changement_etat() {
```

³ SÉGUIN, Marc. Physique XXI Tome A – Mécanique, ERPI, 2010, 588 p.


```

    if (digitalRead(hall_pin) == HIGH) { // Si l'état du capteur est HIGH (aimant détecté)
        compteur_revolutions++; // Incrémenter le compteur de révolutions
    }
}

void setup(){
    Serial.begin(9600);
    pinMode(hall_pin, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(hall_pin), detecter_changement_etat, CHANGE);
    // Attacher l'interruption pour détecter les changements d'état du capteur
}

void loop(){
    unsigned long temps_actuel = millis(); // Temps actuel en millisecondes
    unsigned long delai_ecoule = temps_actuel - temps_precedent; // Calcul du temps
    écoulé depuis la dernière mesure
    float distance_par_tour = DIAMETRE_ROUE * PI; // Calcul de la distance parcourue par
    tour en mètres
    float distance_totale = compteur_revolutions * distance_par_tour; // Calcul de la
    distance totale parcourue en mètres
    if (delai_ecoule >= DELAI_MESURE && compteur_revolutions != 0) { // Si le délai entre
    chaque mesure est atteint
        temps_precedent = temps_actuel; // Mettre à jour le temps précédent

        // Calcul de la vitesse en m/s
        vitesse = (distance_totale / compteur_revolutions) / (delai_ecoule / 1000.0);

        // Conversion en mph
        vitesse *= 2.23694;
    }

    compteur_revolutions = 0; // Réinitialiser le compteur de révolutions

    // Si aucune mesure depuis trop longtemps, la vitesse est nulle
    if (delai_ecoule >= DELAI_ZERO) {
        vitesse = 0;
    }
    Serial.println(vitesse);
}

```

Nous avons choisi d'utiliser une interruption afin de s'assurer que le décompte du nombre de révolution soit cohérent. En effet, si nous avions vérifié l'état du capteur à effet Hall directement dans notre fonction loop(), il aurait été possible que notre compteur ne soit parfois pas incrémenté en raison d'un délai dans l'exécution de notre programme, surtout à grande vitesse. Quelques détails par rapport à l'utilisation de la méthode attachInterrupt()⁴ doivent toutefois être considérés. Premièrement, il faut que la méthode exécutée lors de l'interruption soit très simple pour éviter de retarder l'exécution de la fonction loop(). Deuxièmement, le choix de la broche à utiliser ne peut pas être fait au hasard : seules quelques-unes d'entre elles permettent l'utilisation d'une interruption et varient selon le modèle du microcontrôleur. En voici quelques exemples.

⁴ Référence : <https://reference.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/>

attachInterrupt()

[External Interrupts]

Description

Digital Pins With Interrupts

The first parameter to `attachInterrupt()` is an interrupt number. Normally you should use `digitalPinToInterrupt(pin)` to translate the actual digital pin to the specific interrupt number. For example, if you connect to pin 3, use `digitalPinToInterrupt(3)` as the first parameter to `attachInterrupt()`.

BOARD	DIGITAL PINS USABLE FOR INTERRUPTS
Uno, Nano, Mini, other 328-based	2, 3
Uno WiFi Rev.2	all digital pins
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21
Micro, Leonardo, other 32u4-based	0, 1, 2, 3, 7
Zero	all digital pins, except 4
MKR Family boards	0, 1, 4, 5, 6, 7, 8, 9, A1, A2
Due	all digital pins
101	all digital pins (Only pins 2, 5, 7, 8, 10, 11, 12, 13 work with CHANGE)

En ce qui concerne le choix du capteur à effet Hall, n'importe quel modèle ferait l'affaire et ne devrait pas apporter de changement significatif au fonctionnement de notre système. Dans notre cas, nous avons utilisé le NJK-5002C pour son bas prix et sa simplicité. Voici les fiches techniques associées à ce modèle. À noter que le circuit présenté à la dernière page ne s'applique pas à notre projet : notre capteur est alimenté par la même source que notre Arduino, donc un diviseur de voltage n'est pas nécessaire.


HT

Handson Technology

Two Cables

NJK-5002C Hall Effect Proximity Sensor

The NJK-5002C is a magnetic Hall Effect proximity sensor which is used as an indicator of position or proximity of magnetic materials. When the NJK-5002C is close to magnetic object, its output sends a control signal in addition to having a status indicator LED which visually supports its in the detection. It can also be used as a speed counter.




SKU: SSR1047

Brief Data:

- Model: NJK-5002C
- Supply voltage: 5-30VDC
- Detection Distance: 18mm (effective detection distance 6-18mm)
- Load current: <150mA
- Output Form: NPN
- Output state: Normally Open (NO)
- Detection Object: Magnetic Material
- Output Indication: LED (red)
- Probe Dimension: Ø12x12mm
- Cable Length: 110cm
- Weight: ~40g

Mechanical Dimensions:


Unit: mm



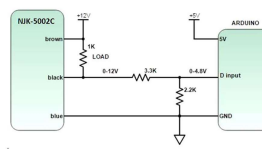
1

www.handsontec.com

Terminal Assignment:



How to Connect to Arduino Board ?



3

www.handsontec.com

Il est aussi important de s'assurer de la solidité du montage afin d'éviter de perdre l'aimant collé sur la roue ou d'endommager le capteur à effet Hall. Il faut aussi s'assurer que le fil reliant le capteur au microcontrôleur n'entrave pas les mouvements du guidon. Nous avons eu énormément de problèmes liés au placement de l'aimant sur la roue, qui n'était parfois plus détecté par le capteur. Nous l'avons changé à plusieurs reprises, dont pendant la présentation de notre projet lors de la colloque des sciences.

Méthode choisie pour maintenir la vitesse définie

Le premier modèle fonctionnel de notre Cruise control fonctionnait en mémorisant la valeur analogue de la gâchette au moment où le bouton SET était appuyé et en envoyant constamment la donnée convertie au moteur. Cette méthode fonctionnait bien sur un terrain plat, mais ne permettait pas de maintenir une vitesse constante sur différents dénivelés. En effet, la valeur envoyée au moteur n'était aucunement liée à la vitesse réelle de la trottinette, mais s'apparente plutôt à la puissance à fournir. Pour avoir un véritable Cruise control dont la vitesse est maintenue peu importe l'environnement, il fallait donc contrôler ce pourcentage de puissance moteur en fonction de la vitesse réelle de la trottinette. Pour ce faire, nous avons développé un algorithme simple qui augmente ou diminue graduellement la valeur analogue étant initialement égale à la valeur retournée par la gâchette au moment où le bouton SET est appuyé. Ainsi, si la vitesse réelle est trop élevée par rapport à la vitesse voulue, on diminue la puissance du moteur jusqu'à atteindre l'objectif, ou inversement en augmentant la puissance si la vitesse est trop basse. Si le moteur fournit déjà sa puissance maximale, un avertissement s'affiche à l'écran pour avertir l'utilisateur que la vitesse souhaitée ne pourra pas être atteinte. Pour que cette méthode fonctionne, il était toutefois nécessaire d'intégrer à notre algorithme un délai entre chaque incrémentation / décrémentation de la valeur à envoyer au moteur. En effet, l'exécution de notre programme était beaucoup plus rapide que l'ajustement de la vitesse, qui est contraint par les révolutions de la roue. Il était donc impossible d'avoir une vitesse stable parce que la puissance du moteur changeait trop vite par rapport à l'accélération ou le ralentissement de la roue. Voici un exemple pour mieux illustrer ce problème : si le moteur doit fournir 60% de sa puissance pour faire rouler la trottinette à 10 km/h et que la vitesse lue par notre speedomètre est de 14 km/h (la puissance fournit pour cette vitesse est égale à 80%), notre algorithme diminue ce pourcentage jusqu'à ce que notre speedomètre retourne une vitesse d'environ 10 km/h. La valeur de la puissance a donc le temps de descendre jusqu'à 30% avant que le nombre de tours effectués par la roue soient suffisants pour que notre speedomètre sache que la vitesse a assez diminuée et

que la valeur de la puissance arrête de diminuer. Il faudra finalement qu'elle augmente, puisqu'elle est trop descendue par rapport à la valeur attendue. L'ajout d'un compteur limitant l'exécution de notre algorithme pour éviter qu'il soit lancé à chaque itération de notre fonction `loop()` a suffi pour régler ce problème. Un autre problème encouru lié à notre algorithme est le suivant : le moteur roule à pleine puissance lorsque le bouton SET est appuyé, avant de très doucement réduire sa vitesse pour atteindre la vitesse souhaitée, qui sera alors maintenue. En observant la valeur analogue envoyée au moteur, qui devrait être égale à la lecture faite sur la gâchette à ce moment, nous avons noté qu'elle était complètement incohérente, atteignant ± 7500 au lieu d'être comprise entre ± 150 et ± 750 . Nous n'avons d'ailleurs pas trouvé la cause de ce problème. Pour le contourner, nous avons simplement ajouté à la méthode une vérification que la valeur se trouve entre les bornes définies. Si ce n'est pas le cas, une donnée prédéfinie qui correspond à une puissance moteur moyenne est alors utilisée comme valeur initiale, ce qui réduit le délai avant d'atteindre la vitesse souhaitée puisque la valeur s'approche déjà plus de la cible. À ce point, notre Cruise control, bien que très peu optimisé, était complètement opérationnel et nous avons enfin pu commencer à développer les fonctions adaptatives.

Développement du détecteur de distance

Le but de notre projet étant d'adapter la vitesse de la trottinette à la vitesse des obstacles détectés dans son chemin, il nous était primordial d'intégrer à notre circuit un détecteur de distance permettant de calculer cette vitesse relative. Nous avons passé beaucoup de temps à effectuer des recherches sur différents types et modèles de capteurs. Nous en avons aussi testé certains qui étaient à notre disposition, sans jamais réussir à mesurer quoi que ce soit. Nous avons donc finalement choisi un détecteur de distance dont la documentation était facilement accessible sur Internet et dont nous savions qu'une bibliothèque existait déjà pour le contrôler : le TF-Luna LiDAR. Ce type de détecteur, envoyant des impulsions lumineuses courtes qui rebondissent sur les surfaces environnantes afin de mesurer le temps qu'il faut pour que la lumière réfléchie revienne au capteur, est reconnu pour sa précision et son utilisation simplifiée, le tout pour une somme modique. Il satisfaisait aussi la portée minimale que nous avions mesurée plus tôt : le TF-Luna peut mesurer avec précision jusqu'à 8 m et nous devons respecter 5 à 7 m pour assurer la sécurité de notre système. Nous avons trouvé en ligne le code suivant permettant de mesurer avec précision une distance.

```
/* This program is the interpretation routine of standard output protocol of TFmini-Plus product on Arduino.
```

For details, refer to Product Specifications.

For Arduino boards with only one serial port like UNO board, the function of software visual serial port is

to be used. */

```
#include <SoftwareSerial.h> //header file of software serial port
SoftwareSerial Serial1(2,3); //define software serial port name as Serial1 and define
pin2 as RX and pin3
/* For Arduinoboard with multiple serial ports like DUEboard, interpret above two
pieces of code and
directly use Serial1 serial port*/
int dist; //actual distance measurements of LiDAR
int strength; //signal strength of LiDAR
float temprature;
int check; //save check value
int i;
int uart[9]; //save data measured by LiDAR
const int HEADER=0x59; //frame header of data package
void setup() {
    Serial.begin(9600); //set bit rate of serial port connecting Arduino with computer
    Serial1.begin(115200); //set bit rate of serial port connecting LiDAR with Arduino
}
void loop() {
    if (Serial1.available()) { //check if serial port has data input
        if(Serial1.read() == HEADER) { //assess data package frame header 0x59
            uart[0]=HEADER;
            if (Serial1.read() == HEADER) { //assess data package frame header 0x59
                uart[1] = HEADER;
                for (i = 2; i < 9; i++) { //save data in array
                    uart[i] = Serial1.read();
                }
                check = uart[0] + uart[1] + uart[2] + uart[3] + uart[4] + uart[5] + uart[6] +
                uart[7];
                if (uart[8] == (check & 0xff)){ //verify the received data as per protocol
                    dist = uart[2] + uart[3] * 256; //calculate distance value

                    Serial.print("dist = ");
                    Serial.print(dist); //output measure distance value of LiDAR
                    Serial.print("\n");
                }
            }
        }
    }
}
```

À noter que les broches 2 et 3 doivent être utilisées afin que ce code soit fonctionnel avec un Arduino Uno, alors que les broches 18 et 19 doivent être utilisées afin que ce code soit fonctionnel avec un Arduino Mega. Nous supposons que la bibliothèque utilise une interruption comme nous l'avons fait avec notre speedomètre. Aussi, la valeur retournée inclue la taille de la composante. Dans notre cas, puisque nous avons collé le LiDAR sur le manche de la trottinette, la mesure se trouve à être de l'objet détecté jusqu'au manche. À cause de l'inclinaison du manche, la distance mesurée lorsque la roue touche à un mur était de 14 cm. Il faut donc prendre cette valeur en considération lors des calculs de la distance à respecter.



Ajustement de la vitesse en fonction des obstacles par notre système

Une fois le détecteur de distance installé et fonctionnel, il ne nous restait plus qu'à définir le comportement de notre programme selon la distance des obstacles. Afin d'optimiser la fonctionnalité adaptative tout en assurant une sécurité accrue, nous avons décidé d'utiliser une fonction exponentielle pour déterminer la distance sécuritaire à respecter en fonction de la vitesse de la trottinette. Ainsi, selon la formule $d = ae^{bv}$, où a et b sont des constantes, d la distance à respecter et v la vitesse de la trottinette, nous avons déterminé à l'aide de l'application Desmos la valeur des constantes pour avoir une courbe qui nous convenait. Voici l'équation :

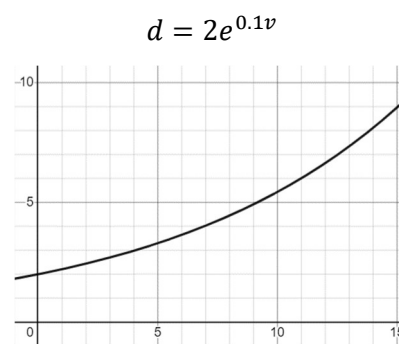


Figure 1. Distance sécuritaire à respecter en fonction de la vitesse de la trottinette.

Selon cette courbe, peu importe la vitesse de la trottinette, la distance sécuritaire est toujours supérieure à 2 m. À basse et moyenne vitesse, la distance augmente légèrement. À haute vitesse (9 mph et plus), la distance est supérieure à 5 m, ce qui avait été convenu comme étant la distance de freinage d'urgence à pleine vitesse. À pleine vitesse, la distance est de 9 m, permettant au système de détecter qu'un freinage d'urgence doit être effectué ou si ralentir suffit. Cela permet aussi à l'utilisateur de réagir après avoir entendu l'avertisseur sonore.

Complexité du freinage automatisé

Afin d'automatiser notre Cruise control, l'idéal aurait été d'y ajouter un système de freinage contrôlé par notre microcontrôleur. Lors de notre analyse préliminaire, nous étions déjà conscients que les chances étaient minces d'avoir le temps de s'y pencher, principalement à cause de la complexité du problème. En effet, puisque la trottinette est munie d'un système de frein à câble, il faut trouver un moyen d'actionner le levier informatiquement tout en y appliquant une force graduelle pour éviter le freinage sec et risquer de tomber. Or, nos seules idées requéraient l'utilisation d'un moteur DC ou d'un électroaimant, lesquels ne seraient pas adaptés à nos besoins. Cette problématique non résolue se trouve à être une bonne piste pour quiconque aurait envie de reprendre et d'améliorer notre projet.

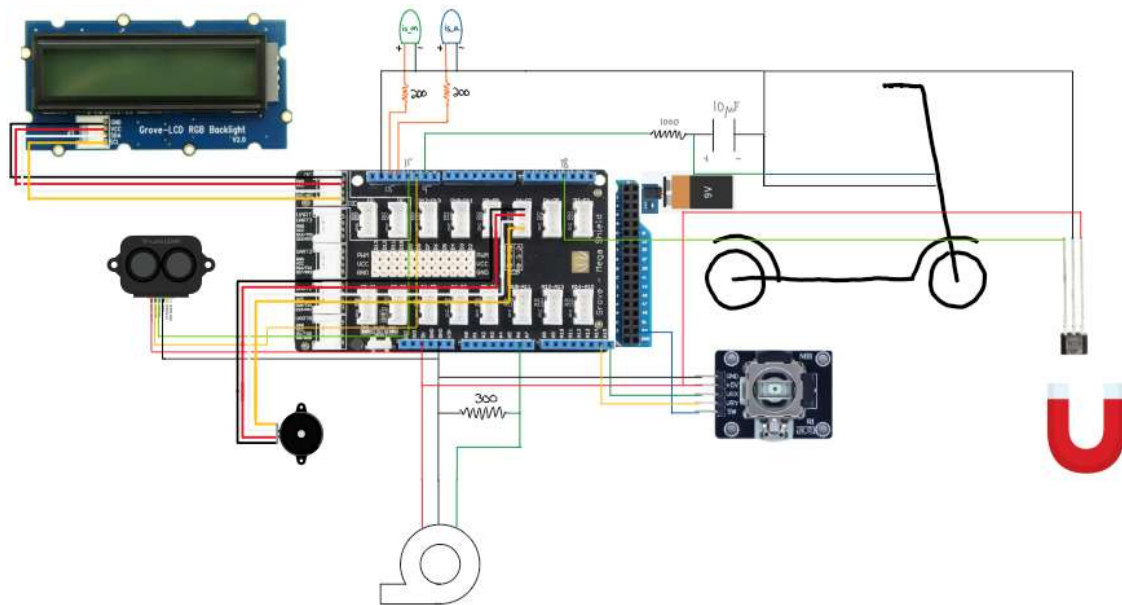
Alternative au freinage automatisé

Puisque nous avons dès le début du projet peu d'espoir d'inclure un système de freinage automatisé à notre circuit, nous avons déjà pensé ajouter un avertisseur sonore pour prévenir l'utilisateur d'un danger imminent. Même si les fonctionnalités adaptatives n'ont finalement pas été terminées, un buzzer fonctionnel a été inclus au code de notre version finale incomplète (la V7). Plus l'écart entre la distance à respectée et la distance réelle avec un obstacle était grand, plus la fréquence était élevée. Il serait aussi possible de raccourcir le délai entre chaque son émis lorsque le danger est accru. La composante à notre disposition, qui faisait partie du Grove Beginner Kit, ne pouvait pas émettre un son assez fort à notre goût. Nous n'avons toutefois pas eu le temps de changer cette pièce dû au manque de temps encouru à la fin du projet, mais il serait envisageable pour une équipe souhaitant reprendre ce projet de changer le buzzer pour en prendre un plus puissant.

Fin du projet

Nous avons été contraints de remettre notre projet avant d'avoir réussi à finaliser le développement de nos fonctions adaptatives. Nous avons donc présenté à la colloque des

sciences un Cruise control fonctionnel muni d'un speedomètre et d'une interface physique. Son utilisation était identique à celle d'une voiture (maintien d'une vitesse en km/h plutôt qu'une puissance moteur constante). Nous avons aussi la distance mesurée par le LiDAR d'affichée sur notre écran, sans qu'elle ne serve à quoi que ce soit. Nous n'avions pas non plus de module de freinage informatisé, mais nous avons toutefois un indicateur sonore pour aviser l'utilisateur qu'il doit appuyer sur le levier de frein. Celui-ci était fonctionnel, mais il n'était appelé par aucune méthode de notre programme, puisqu'aucune fonction adaptative n'était terminée. Il n'a donc pas été démontré. Voici un schéma de notre circuit final, incorporant tous nos modules.



Évolution du programme

Pour chaque changement majeur à apporter à notre programme, nous avons créé un nouveau fichier afin de conserver toutes nos versions. Cela nous a permis de revenir en arrière lorsque nous ne trouvions pas la cause d'un problème en lien avec la logique de notre code ou qu'une nouvelle méthode ne fonctionnait pas. Il était d'ailleurs plus facile d'intégrer certaines composantes de la sorte. En effet, puisque la majorité de nos modules (les boutons, l'écran, le speedomètre, etc.) ont d'abord été développés indépendamment de notre Cruise control, fusionner les deux codes était plus facile en pouvant consulter le programme original. Il est aussi intéressant d'en voir l'évolution. Comme nous comptons joindre à notre documents ces différentes versions, voici un bref résumé de leurs majeures différences et fonctionnalités.

Version 1 : premier montage

Cette première version permettait de contrôler le Cruise control avec les cinq boutons et la gâchette de la trottinette. La puissance à fournir au moteur en valeur analogue était affichée sur l'écran OLED rejeté dû au ralentissement qu'il causait. Il s'agit donc du premier agencement de l'interface physique initiale avec les fonctions permettant de lire la valeur retournée par la gâchette et de contrôler la puissance du moteur.

Version 2 : joystick, écran LCD, LEDs et contrôle du moteur avec un Arduino

La deuxième version fût créée lorsque les boutons ont été remplacés par le joystick et l'écran OLED par l'écran LCD. Le microcontrôleur a également été changé à ce moment : le Arduino Uno a été remplacé par un Arduino Mega. Il s'agit de la dernière version fonctionnelle à ce jour. La vitesse affichée à l'écran est toutefois encore la valeur analogue décrivant la puissance à fournir au moteur plutôt que sa vitesse réelle.

Version 3 : speedomètre et vitesse réelle au lieu d'une valeur analogue

La troisième version intègre le speedomètre. Bien que la vitesse affichée à l'écran soit maintenant la bonne, le Cruise control n'est plus fonctionnel. Au lieu de rechercher le problème, nous avons plutôt décidé de tout de suite passer à une quatrième version.

Version 4 : allègement de la fonction loop()

Puisque notre fichier .ino était très lourd (plus de 400 lignes de code) et qu'il était difficile de déceler l'erreur dans l'IDE d'Arduino, nous avons décidé de refaire le code en optant pour une approche orientée objet en utilisant des fichiers .h et .cpp. Chaque module s'est donc vu attribuer sa propre classe, ce qui rend le programme plus facile à lire. Bien que cette version soit meilleure que la troisième, des problèmes empêchent toujours le Cruise control de bien fonctionner.

Version 5 : simplification du code en abandonnant certaines conventions de la programmation orientée objet

Comme le changement complet de la structure de notre programme avait rendu notre Cruise control dysfonctionnel, le professeur nous avait conseillé de construire notre programme de façon procédurale plutôt que d'employer un langage orienté objet qui complexifiait inutilement notre projet. Nous avons donc décidé de combiner les avantages apportés par ces deux méthodes en rendant toutes nos classes publiques et la majorité statiques. La programmation devenait donc plus aisée, tout en conservant l'indépendance de chaque module. Cette version et les prochaines ne respectent donc pas de nombreuses conventions de programmation apprises lors de nos cours précédents, comme l'encapsulation, mais répond le mieux à nos besoins et rend

notre Cruise control complètement fonctionnel. Seules les fonctionnalités adaptatives manquent à cette version.

Version 6 : implémentation du LiDAR

Cette version intègre le LiDAR à notre programme. La distance mesurée est affichée à l'écran, pendant que le reste du Cruise control continue d'effectuer la tâche demandée. Il s'agit de la version utilisée lors de la présentation de notre projet à la colloque des sciences.

Version 7 : méthodes adaptatives ajustant la vitesse en fonction des obstacles

Cette ébauche est notre première et dernière tentative à rendre notre Cruise control adaptatif. Avec quelques jours de plus, cette version aurait probablement pu être terminée. Le temps nous a toutefois contraint d'abandonner l'ajustement de la vitesse automatique en fonction des obstacles détectés. Aucun test des méthodes de la classe Adaptatif n'a été concluant, cette version est donc obsolète. Il pourrait s'agir d'un bon point de départ pour une équipe souhaitant compléter notre projet.