Compétition de robotique *FIRST*® 2024

Guide de programmation C++ pour KitBot

1 Table des matières

2 Présentation du document

Ce document vous guidera à travers la façon de préparer votre KitBot 2024 et de le faire fonctionner en utilisant l'exemple de code C++ fourni. Pour éviter la duplication de contenu, ce document fait fréquemment référence à la documentation WPILib [NdT La documentation WPILib est disponible en français] pour des étapes spécifiques du développement. En plus de vous rendre fonctionnel avec le code fourni, ce document passera en revue la structure de ce code afin que vous puissiez comprendre comment il fonctionne. Enfin, nous passerons en revue certains des changements les plus probables que vous voudrez peut-être apporter au code et fournirons des exemples concrets de la façon d'apporter ces modifications.

Pour commencer avec l'exemple de code, ou pour y apporter certaines modifications, une compréhension minimale de C++ est nécessaire. Le code et les exemples de modification fournis vous permettront d'avancer avec suffisamment de structure. Pour comprendre la procédure pas à pas, ou pour apporter des modifications non décrites dans ce document, une compréhension plus complète de C++ est probablement requise. Le module Intro to Programming (voa) sur Thinkscape est un excellent moyen d'en apprendre davantage sur Java en utilisant WPILib et les plates-formes robotisées Romi ou XRP. Pour d'autres options, consultez les liens sur la page Web d'Introduction De Zéro à Robot.

Ce document, et l'exemple de code fourni, suppose l'utilisation des contrôleurs SPARK MAX fournis dans le kit de lancement des recrues.

3 Prise en main de votre code KitBot

3.1 Câblage de votre robot

Utilisez le document Introduction au câblage d'un robot FRC sur WPILib pour vous aider à câbler votre robot. Quelques notes spécifiques au KitBot 2024 :

- Le KitBot 2024 n'utilise pas de pneumatique. Vous pouvez ignorer les instructions concernant le concentrateur pneumatique ou le module de commande pneumatique, sauf si vous ajoutez de la pneumatique à votre design.
- Afin d'utiliser les mêmes ID pour le fonctionnement avec PWM et CAN, le code KitBot 2024 n'utilise pas le port PWM 0. Câblez les ports PWM en fonction des ID dans Constants.java (Gauche = 1,2; Droite = 3,4) ou modifiez les constantes pour refléter votre câblage.
- Le KitBot 2024 contient deux moteurs supplémentaires non inclus dans le document de câblage de base. Câblez-les de la même manière que les moteurs du système motopropulseur. Si vous utilisez les PWM, connectez le moteur de l'admission (le plus près du centre du robot) au port PWM 5 et le moteur du lanceur (le moteur plus proche de l'extérieur du robot) au port PWM 6.

3.2 Configuration du matériel et de l'environnement de développement

Avant de pouvoir charger du code et tester votre robot, vous devrez configurer votre matériel (roboRIO, radio, etc.) et configurer votre environnement de développement. Suivez les étapes 2 à 4 du guide d'Installation des composants logiciels de WPILib pour tout configurer et vous assurer que vous pourrez déployer un projet de robot de base.

Si vous utilisez les PWM, assurez-vous que les 6 MAX SPARK sont en mode « brushed » (voa). Lorsqu'alimenté, la DEL doit clignoter en jaune ou en bleu, pas en magenta ou en cyan. Pour changer le mode, vous pouvez soit maintenir le bouton Mode enfoncé pendant 3 secondes, soit utiliser la connexion USB et le client matériel REV. Vous pouvez également vérifier les modes neutres, frein ou libre (voa). Il est recommandé de régler les moteurs de l'admission et du lanceur en mode libre (jaune clignotant). Pour modifier le mode, appuyez brièvement sur le bouton Mode (moins de 3 secondes) ou utilisez la connexion USB et REV Hardware Client. Il n'y a pas de recommandation spécifique pour les moteurs de la propulsion, mais vous voudrez probablement que les 4 moteurs de propulsion soient sur le même mode, vous voudrez peut-être essayer de piloter avec chaque réglage pour décider ce que vous préférez.

3.3 Ouverture de l'exemple KitBot 2024

L'exemple de code KitBot 2024 est fourni dans des fichiers zip individuels pour chaque langage sur la page Web KitBot. Pour ouvrir le code C++ :

1. Téléchargez et décompressez l'exemple de code C++. Assurez-vous de décompresser ou de copier vers un emplacement permanent, pas dans un dossier temporaire.
2. Ouvrez WPILib VS Code à l'aide du menu Démarrer ou des raccourcis du bureau.

3. En haut à gauche, cliquez sur **File->Open Folder** et accédez au dossier « C ++ » à l'intérieur de l'exemple de code décompressé, puis cliquez sur **Select Folder**.

3.4 Passage au contrôle via CAN

Si vous avez câblé vos contrôleurs de moteur SPARK MAX à l'aide de CAN, vous devrez effectuer d'autres modifications de configuration et de code avant de continuer. Si vous utilisez PWM, passez à la section 3.5 pour déployer et tester le code.

3.4.1 Configuration des SPARK MAX

Avant d'utiliser les SPARK MAX avec contrôle CAN, chacun d'eux doit se voir attribuer un ID unique. Étant donné que vos SPARK commencent tous avec le même ID, vous pouvez débrancher le bus CAN de chaque appareil pendant que vous mettez à jour et attribuez un ID.

1. Installez le client matériel REV Hardware Client (voa)
2. Avec le robot éteint, connectez un câble USB entre l'ordinateur et le port USB du SPARK MAX. Laissez le robot éteint garantit que seul le SPARK MAX est alimenté et évite de changer les ID sur des appareils non désirés.
3. Installez la mise à jour du firmware on the SPARK MAX (voa)
4. Sélectionnez le ID CAN et le type de moteur (CAN ID and Motor Type -voa). Vous pouvez passer la limite de courant et sauvegarder les changements.
    a. Les ID CAN de chaque appareil se trouvent dans Constants.java. Vous pouvez soit définir les appareils pour qu'ils correspondent à ces ID, soit définir les ID comme vous le souhaitez (certaines équipes définissent ID CAN = le numéro de canal auquel l'appareil est connecté sur le PD), puis mettez à jour ces constantes.
    b. Remarque : Si vous souhaitez « Faire tourner le moteur » comme décrit sur cette page, assurez-vous que le robot est dans un état sûr pour le faire (roues ne touchant pas le sol ou la table).
5. Répétez l'opération pour les 6 appareils du robot.
6. Bien que cela ne soit pas nécessaire, si vous utilisez le PDH de REV, vous voudrez peut-être vérifier maintenant qu'il a également le dernier firmware. Ne changez pas l'ID du PDH hors de la valeur par défaut, chaque type d'appareil a un espace distinct d'identification et votre PDH n'entrera pas en conflit avec votre SPARK MAX même s'il est placé au même ID.

Maintenant que tous vos appareils sont configurés, vous pouvez effectuer une vérification préliminaire que votre bus CAN est câblé correctement à l'aide du client matériel REV. Lorsque vous êtes branché avec un câble USB sur n'importe quel appareil REV de votre bus CAN, mettez le robot sous tension et vous devriez voir tous les autres appareils répertoriés dans le volet gauche du client matériel REV, sous l'en-tête CAN Bus. Si vous ne voyez pas tous les appareils, vous avez probablement un ou plusieurs problèmes avec le câblage de votre bus CAN :

1. Vérifiez que votre bus CAN commence par le roboRIO et se termine par une résistance de 120 ohms, ou le terminateur intégré d'un concentrateur de distribution d'énergie ou d'un panneau de distribution électrique (avec la terminaison réglée sur ON à l'aide du cavalier ou du commutateur approprié.

2. Vérifiez que vos connexions de bus CAN correspondent toutes à jaune-jaune et vert-vert.
3. Vérifiez que toutes les connexions de fil CAN sont sécurisées les unes aux autres et que les connecteurs sont installés en toute sécurité dans chaque SPARK Max
4. Si vous rencontrez toujours des problèmes, déplacer la connexion USB vers différents appareils. Vérifier ce que chaque appareil peut « voir » dans le bus peut aider à localiser l'emplacement d'un problème.

3.4.2 Installation de REVLib

La bibliothèque logicielle du SPARK MAX en mode CAN est fournie par le fournisseur (REV Robotics). La configuration de la bibliothèque de tierce partie est déjà incluse dans le projet, mais vous devrez installer la bibliothèque elle-même. Il y a deux façons de le faire :

1. **Recommandé** Installer la bibliothèque hors ligne - Cela garantira que la bibliothèque demeure sur votre ordinateur même si vous ne créez pas de nouveau code pendant un certain temps (les installations en ligne peuvent être nettoyées automatiquement par Gradle).
    a. Téléchargez la dernière version de REVLib à l'aide du lien de la documentation REV (voa).
    b. Décompressez dans le répertoire *C :\Users\Public\wpilib\2024* sous Windows et le répertoire *~/wpilib/2024* sur les systèmes de type Unix.
2. Installer en ligne - Pendant que l'ordinateur est connecté à Internet, cliquez sur l'icône WPILib en haut à droite de la fenêtre VSCode pour afficher l'invite d'extension WPILib, puis commencez à taper « Build Robot Code » et sélectionnez cette option lorsqu'elle apparaît. Cela récupérera automatiquement la bibliothèque en ligne.

3.4.3 Mise à jour du code

Le code est principalement en place dans le projet pour passer du contrôle PWM au contrôle CAN, il vous suffira de faire quelques modifications pour basculer.

1. Dans RobotContainer.h:

```
13  //#include "subsystems/PWMDrivetrain.h"
14  //#include "subsystems/PWMLauncher.h"
15  #include "subsystems/CANDrivetrain.h"
16  #include "subsystems/CANLauncher.h"
```

```
39  //PWMDrivetrain m_drivetrain;
40  CANDrivetrain m_drivetrain;
41  //PWMLauncher m_launcher;
42  CANLauncher m_launcher;
```

    a. Décommentez les instructions include pour CANDrivetrain et CANLauncher. Vous pouvez porter en commentaire ou supprimer les instructions pour les sous-systèmes PWM si vous le souhaitez.

b. Portez en commentaire les lignes de déclaration des variables membres pour les sous-systèmes PWM et décommentez ceux des sous-systèmes CAN.

2. Dans les fichiers Autos, LaunchNote et PrepareLaunch .h :

```
// #include "subsystems\PWMLauncher.h"
#include "subsystems\CANLauncher.h"
```

```
33      //PWMLauncher* m_launcher;
34      CANLauncher* m_launcher;
```

```
22    LaunchNote(CANLauncher* launcher);
```

   a. Décommentez include pour le sous-système CAN et portez en commentaire ou supprimez celui pour PWM.
   b. Décommentez la déclaration de variable membre pour le sous-système CAN et portez en commentaire ou supprimez celle du sous-système PWM. Ignorez ceci pour Autos.h
   c. Modifiez le type de paramètre dans le constructeur de sous-système PWM vers sous-système CAN.

3. Dans les fichiers Autos, LaunchNote et PrepareLaunch .cpp :

```
12    LaunchNote::LaunchNote(CANLauncher* launcher) : m_launcher(launcher) {
```

   a. Modifiez le type de paramètre de méthode de sous-système PWM vers sous-système CAN

3.5 Déploiement et test de l'exemple pour KitBot

Pour déployer l'exemple sur votre robot, vous devrez définir le numéro d'équipe dans le projet. Cliquez sur l'icône WPILib dans le coin supérieur droit de la fenêtre VS Code (W à l'intérieur d'un engrenage) pour ouvrir l'invite WPILib et commencez à taper « Set Team Number » et sélectionnez cette option lorsqu'elle apparaît. Entrez votre numéro d'équipe (pas de préfixe 0 - par exemple 123 ou 9996) et appuyez sur Enter.

Vous êtes maintenant prêt à déployer l'exemple pour KitBot tout comme vous avez déployé le projet test à l'étape 4 du guide Zéro à Robot.

Avertissement: Assurez-vous d'avoir de l'espace dans toutes les directions lorsque vous faites fonctionner un robot. Même avec un code connu, le robot peut se déplacer à une vitesse ou dans des directions inattendues. Soyez prêt à désactiver le robot (Enter) ou à faire un arrêt d'urgence E-stop (barre d'espace) si nécessaire. Le code pour KitBot 2024 contient une routine autonome très simple qui déplacera le robot vers l'arrière à une vitesse de 1/2 pendant 1 seconde lorsque le robot est activé en mode autonome.

3.6 Configuration des manettes de jeu

Le code est configuré pour utiliser la classe de manette Xbox. Les manettes de jeu Logitech F310 fournies dans le kit de pièces apparaîtront comme des manettes Xbox dans le logiciel WPILib si elles sont configurées dans le mode correct. Pour configurer les contrôleurs, vérifiez que le commutateur à

l'arrière du contrôleur est défini sur le paramètre « X ». Ensuite, lorsque vous utilisez le contrôleur, assurez-vous que le voyant à côté du bouton Mode est éteint, s'il est allumé, appuyez sur le bouton Mode pour le basculer. Lorsque le bouton Mode est allumé, le contrôleur permute la fonction du bâton analogique gauche et du D-pad.

3.7 Que fait le code?

Le code fourni implémente les commandes de robot suivantes dans le mode téléopéré :

- La manette de contrôle est une manette Xbox dans l'emplacement 0 de DriverStation
    - Contrôle la propulsion du robot à l'aide de Split-stick Arcade Drive
        - L'axe Y (vertical) du stick gauche contrôle le mouvement avant-arrière de la plateforme pilotable
        - L'axe X (horizontal) du stick droit contrôle la rotation de la plateforme pilotable
- La manette d'opérateur est une manette Xbox dans l'emplacement 1 de la station de pilotage
    - Gâchette avant gauche - Active les deux roues du lanceur de pièce de jeu vers l'intérieur à des vitesses différentes pendant que le bouton est maintenu. Cela permet au robot d'admettre une pièce de jeu
    - Bouton A – Exécute une courte séquence pour lancer une pièce de jeu pendant que le bouton est maintenu
        - Démarre la roue avant pour atteindre sa pleine vitesse
        - Attend 1 seconde
        - Active la roue arrière pour transmettre une pièce de jeu vers la roue avant qui tourne

4 Structure globale du code

Le code fourni utilise la structure de programmation orientée commandes (command-based) fournie par WPILib. Cette structure divise les actionneurs du robot en « sous-systèmes » qui sont contrôlés par des « commandes » ou des collections de commandes (nommez « groupes de commandes »). La structure orientée commandes peut être un peu exagérée pour un robot aussi simple, mais elle s'adapte très bien aux équipes qui cherchent à ajouter des fonctionnalités supplémentaires à leur KitBot, et elle fournit de nombreux outils utiles pour gérer les actions et les séquences comme on le verra en regardant le code du lanceur de pièces de jeu. De plus, cette structure de code a été utilisée par plus de 60% des équipes en 2023, ce qui augmente la probabilité que les équipes autour de vous puissent fournir de l'aide avant ou pendant l'événement.

Pour en savoir plus sur la structure orientée commandes, consultez le chapitre sur la Programmation orientée commandes sur WPILib.

4.1 Façons de créer des commandes

Il existe plusieurs façons de définir des commandes dans la structure orientée commandes. Ce projet utilise bon nombre de ces différents types afin d'illustrer ce à quoi ils ressembleraient dans un projet de robot complet. Si un type vous semble étrange ou n'a pas beaucoup de sens, ne vous inquiétez pas, vous devriez être en mesure d'utiliser la documentation sur ce que fait la commande combinée avec des exemples d'autres façons de créer des commandes pour la recréer sous la forme que vous préférez. Les méthodes courantes de création de commandes utilisées dans ce projet sont les suivantes :

- Définie comme leur propre classe dans leur propre fichier (par exemple PrepareLaunch et LaunchNote)
- Via une « méthode Fabrique de commande » dans le sous-système (par exemple GetIntakeCommand() dans les sous-systèmes Launcher)
- Inline où la commande est liée à un bouton ou définie par défaut (par exemple, la ligne 19 de RobotContainer.cpp)

Ces mêmes méthodes s'appliquent également aux groupes de commandes. Ce projet utilise :

- Groupe de commandes inline où la liaison se produit. Cela se fait via des « décorateurs », des méthodes auxquelles vous pouvez faire appel sur une commande pour la transformer en un groupe de commandes avec des propriétés spécifiques, comme « withTimeout » (par exemple, la ligne 30 de RobotContainer.cpp)
- Fabrique de groupe de commandes dans son propre fichier (Autos)

Ce projet ne crée aucun groupe de commandes dans son propre fichier, un exemple peut être vu au bas de la page Groupes de commandes liée ci-dessus ainsi que sur la page Web Organisation des projets orientés commandes sur WPILib.

La plupart des équipes n'utiliseront pas tous ces styles dans leur code, optant plutôt pour sélectionner un ou deux types qui leur conviennent le mieux.

5 Présentation du code

5.1 Sous-systèmes

Comme décrit dans Qu'est-ce que la programmation « orientée commandes » ? les « sous-systèmes représentent des collections d'équipements contrôlées indépendamment (tels que des contrôleurs de moteur, des capteurs, des actionneurs pneumatiques, etc.) qui fonctionnent ensemble.

Pour le KitBot 2024, nous avons regroupé les 6 moteurs en 2 groupes, la propulsion et le lanceur. Pour ce robot, ces choix étaient assez faciles, les 4 moteurs de la propulsion doivent toujours travailler ensemble pour déplacer le robot sur le terrain et les 2 moteurs du lanceur doivent toujours travailler ensemble pour manipuler les pièces de jeu. Parfois, les limites entre les sous-systèmes peuvent ne pas être si claires, si vous avez un bras avec une articulation de l'épaule et du poignet et un ensemble de roues motorisées à l'extrémité, est-ce un sous-système ou plusieurs ? La règle générale à suivre est de réfléchir aux actions, ou commandes, que vous pourriez avoir pour contrôler les sous-systèmes. Pensez-vous que vous voudrez que deux pièces soient contrôlées indépendamment l'une de l'autre (par exemple, actionner l'admission tout en déplaçant le bras ou le poignet ?). Si vous n'êtes pas sûr, allez vers des sous-systèmes plus petits ; vous pouvez toujours créer des commandes qui nécessitent plusieurs sous-systèmes, mais si vous finissez par vouloir des commandes distinctes pour contrôler un seul sous-système en même temps, vous devrez refactoriser le sous-système pour le diviser.

5.1.1 PWMDrivetrain.h

Cette classe est le sous-système de la propulsion si vous avez câblé votre contrôleur de moteur à l'aide de PWM. Si vous avez câblé votre contrôleur de moteur par filage CAN, consultez la section 3.4 pour obtenir des renseignements sur la façon de décommenter l'utilisation de cette classe et remplacez-la par la classe CANDrivetrain.

5.1.1.1 Inclusions

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

#pragma once

#include <frc2/command/SubsystemBase.h>
#include <frc/drive/DifferentialDrive.h>
#include <frc/motorcontrol/MotorControllerGroup.h>
#include <frc/motorcontrol/PWMSparkMax.h>

#include "Constants.h"
```

Cette section commence par la ligne « # pragma once ». Cette ligne indique au compilateur de ne définir cette classe qu'une seule fois, même si elle est incluse à partir de plusieurs endroits différents (telles que plusieurs commandes qui utilisent toutes le même sous-système à des moments différents). Ceci est important pour s'assurer que le code n'essaie pas de définir notre matériel plusieurs fois, ce qui entraînerait des erreurs.

Le reste de cette section se compose d'instructions include, des déclarations qui indiquent au compilateur sur quel autre code ce code s'appuie. La majorité des include ici proviennent de WPILib, ceux-ci peuvent être identifiés par la notation <> et la première partie du chemin étant frc ou frc2. Le dernier include est un autre fichier dans ce projet, il est référencé en utilisant la notation "".

5.1.1.2 Déclaration de classe, variables de membre et constructeur

```
14  class PWMDrivetrain : public frc2::SubsystemBase {
15   public:
16    PWMDrivetrain();
```

La première ligne de cette image est la déclaration de classe. Cela déclare le nom de notre classe et indique qu'il s'agit d'une extension de la classe SubsystemBase. Tous les sous-systèmes doivent étendre cette classe qui fournit certaines fonctions utilitaires concernant la définition du nom du sous-système, son enregistrement auprès du planificateur et l'envoi d'informations à ce sujet au tableau de bord.

La section suivante déclare le constructeur. L'implémentation réelle se trouve dans le fichier .cpp.

5.1.1.3 Méthodes

```
18   /**
19    * Will be called periodically whenever the CommandScheduler runs.
20    */
21   void Periodic() override;
22
23   void ArcadeDrive(double speed, double rotation);
```

La section suivante déclare les méthodes de classe, les implémentations se trouvent dans le fichier .cpp. La méthode Periodic est fournie dans le cadre de la classe de sous-système, elle est donc annotée par « override » pour indiquer que nous sommes en train de remplacer l'implémentation de base. La méthode ArcadeDrive prend des paramètres d'entrée de type double des valeurs de vitesse et de rotation, correspondant à la classe WPILib correspondante que nous appellerons.

5.1.1.4 *Membres de classe*

```
private:
 // Components (e.g. motor controllers and sensors) should generally be
 // declared private and exposed only through public methods.
 // The motor controllers
 frc::PWMSparkMax m_left1;
 frc::PWMSparkMax m_left2;
 frc::PWMSparkMax m_right1;
 frc::PWMSparkMax m_right2;

 // The motors on the left side of the drive
 frc::MotorControllerGroup m_leftMotors{m_left1, m_left2};

 // The motors on the right side of the drive
 frc::MotorControllerGroup m_rightMotors{m_right1, m_right2};

 // The robot's drive
 frc::DifferentialDrive m_drive{m_leftMotors, m_rightMotors};
```

La dernière section concerne les membres de classe. Ceux-ci sont tous inclus dans la section private, l'équipement ne doit être accessible directement que par les méthodes du sous-système, si la commande ou un autre code a besoin d'accéder à l'équipement, nous devrions écrire de nouvelles méthodes de sous-système pour exposer ce dont l'autre code a besoin. Les membres de notre classe de base pilotable sont les 4 contrôleurs de moteur, les deux groupes de contrôleurs (qui sont composés des contrôleurs), et l'objet Differential Drive que nous utiliserons pour contrôler la base pilotable comme une seule unité.

5.1.2 PWMDrivetrain.cpp

5.1.2.1 *Déclarations Include et Using*

```
5   #include "subsystems/PWMDrivetrain.h"
6
7   using namespace DrivetrainConstants;
```

La première section consiste d'instructions include et using. Il y aura généralement une instruction include pour inclure le fichier .h pour la classe à laquelle appartient le fichier .cpp, dans de nombreux cas, ce sera le seul include. En incluant notre fichier .h, les fichiers que .h inclut peuvent également être référencés ici. Des include supplémentaires ne sont nécessaires que si le fichier .cpp référence des classes qui ne sont pas référencées dans le fichier .h.

L'instruction using nous permet d'utiliser une syntaxe plus compacte lorsque nous faisons référence aux membres de ce namespace, dans ce cas, au lieu d'avoir DrivetrainConstants ::kLeftFrontID, nous sommes en mesure d'utiliser uniquement kFrontLeftID.

5.1.2.2 *Constructeur de classe*

```
9   PWMDrivetrain::PWMDrivetrain()
10      : m_left1{kLeftFrontID},
11        m_left2{kLeftRearID},
12        m_right1{kRightFrontID},
13        m_right2{kRightRearID} {
14    // Invert left side motors so both sides drive forward with positive output
15    // values
16    m_leftMotors.SetInverted(true);
17    m_rightMotors.SetInverted(false);
18  }
```

La section suivante du code est le constructeur de la classe. La première ligne indique la classe. Le prochain ensemble de lignes après le « : » est appelé une liste d'initialiseur. Elle initialise les variables membres de la classe, dans ce cas nos contrôleurs de moteur. Les autres variables de classe de cette classe ont leur initialisation définie dans le fichier .h. La dernière partie du constructeur est toute méthode supplémentaire à faire appel lors de la construction. Dans cette classe, nous inversons les moteurs sur le côté gauche de la base pilotable afin que les deux côtés du robot avancent avec des valeurs de sortie positives.

5.1.2.3 *Méthodes*

```
20  // This method will be called once per scheduler run
21  void PWMDrivetrain::Periodic() {}
22
23  void PWMDrivetrain::ArcadeDrive(double speed, double rotation) {
24    m_drive.ArcadeDrive(speed, rotation);
25  }
```

Cette section définit l'implémentation des méthodes de la classe. La méthode Periodic fait partie de la classe Subsystem de base et est appelée une fois par itération du planificateur (~ toutes les 20 ms). Vous pouvez utiliser cette méthode pour mettre à jour les valeurs des capteurs, effectuer la journalisation ou d'autres tâches que vous souhaitez effectuer, quel que soit ce que le sous-système fait réellement. Dans notre propulsion simple, nous n'avons rien que nous souhaitons faire périodiquement. La dernière méthode est la méthode ArcadeDrive, qui appelle la méthode du même nom sur notre objet de pilotage.

5.1.3 CANDrivetrain

Cette classe est le sous-système de la base pilotable si vous avez câblé votre contrôleur de moteur par filage CAN. Si vous avez câblé votre contrôleur de moteur à l'aide de CAN, consultez la section 3.4 pour plus d'informations sur la mise en commentaire de l'utilisation de PWMDrivetrain et son remplacement par la classe CANDrivetrain. La classe CANDrivetrain est très similaire à la classe PWMDrivetrain, consultez la section ci-dessus pour l'explication de la plupart des composants. Les différences sont mises en évidence ci-dessous.

5.1.3.1 CANDrivetrain.h – les Include

```
10    #include "rev/CANSparkMax.h"
```

À la place du include PWMSparkMax de WPILib qui a été utilisé dans PWMDrivetrain, nous avons plutôt ce include CANSparkMax de la bibliothèque tierce de REV Robotics. Cet include utilise la notation “” comme les include d’utilisateur, au lieu de la notation <> utilisée pour WPILib. La partie “rev/” du chemin d’accès vous indique que l’include provient de la bibliothèque REV.

5.1.3.2 CANDrivetrain.h - Variables membres

```
29    rev::CANSparkMax m_left1;
30    rev::CANSparkMax m_left2;
31    rev::CANSparkMax m_right1;
32    rev::CANSparkMax m_right2;
```

Tout comme notre changement à include, le type de nos variables membres du contrôleur de moteur change également.

5.1.3.3 CANDrivetrain.cpp – Initialiseur du constructeur

```
 9  CANDrivetrain::CANDrivetrain()
10      : m_left1{kLeftFrontID, rev::CANSparkMax::MotorType::kBrushed},
11        m_left2{kLeftRearID, rev::CANSparkMax::MotorType::kBrushed},
12        m_right1{kRightFrontID, rev::CANSparkMax::MotorType::kBrushed},
13        m_right2{kRightRearID, rev::CANSparkMax::MotorType::kBrushed} {
```

Lors de l’initialisation des SPARK MAX sur CAN, un paramètre supplémentaire est fourni, indiquant le type de moteur. Pour les moteurs CIM utilisés dans le KitBot, ce sont des moteurs brossés.

5.1.3.4 CANDrivetrain.cpp – Limite de courant

```
19    // Set current limits for all motors
20    m_left1.SetSmartCurrentLimit(kCurrentLimit);
21    m_left2.SetSmartCurrentLimit(kCurrentLimit);
22    m_right1.SetSmartCurrentLimit(kCurrentLimit);
23    m_right2.SetSmartCurrentLimit(kCurrentLimit);
```

Lors du contrôle des SPARK MAX par filage CAN, nous sommes en mesure d’accéder facilement et de définir des paramètres supplémentaires qui contrôlent le fonctionnement. Dans ce cas, nous fixons une limite de courant sur les moteurs de la base pilotable.

5.1.4 PWMLauncher.h

Cette classe est le sous-système du lanceur si vous avez câblé votre contrôleur de moteur par filage PWM. Si vous avez câblé votre contrôleur de moteur à l’aide de CAN, consultez la section 3.4 pour plus

d'informations sur la mise en commentaire de l'utilisation de cette classe et son remplacement par la classe CANLauncher.

5.1.4.1 Inclusions, déclarations de classe, et constructeur

Les premières sections de ce sous-système sont très similaires au sous-système PWMDrivetrain. Voir la section 5.1.1 pour une description plus détaillée de chacune de ces parties du code.

5.1.4.2 Méthodes – Fabrique de commande

```
24    frc2::CommandPtr GetIntakeCommand();
```

Cette méthode est ce qu'on appelle une "Fabrique de commande" car elle crée des instances d'une commande. Nous pouvons appeler cette méthode de n'importe où où nous mettons en place des boutons, créons des groupes de commande, etc. pour obtenir une instance de cette commande. Le type retourné par notre méthode est un CommandPtr, ce type est utilisé dans le cadre de commande WPILib pour indiquer la propriété des objets de commande.

5.1.4.3 Méthodes – Contrôle des équipements

```
26  // An accessor method to set the speed (technically the output percentage) of the launch wheel
27  void SetLaunchWheel(double speed);
28
29  // An accessor method to set the speed (technically the output percentage) of the feed wheel
30  void SetFeedWheel(double speed);
31
32  // A helper method to stop both wheels. You could skip having a method like this and call the
33  // individual accessors with speed = 0 instead
34  void Stop();
```

Le reste de la classe de sous-système est des méthodes d'accès au matériel. Ici, nous définissons toutes les méthodes que les commandes peuvent avoir besoin d'appeler pour obtenir l'état ou effectuer des actions sur le sous-système. Pour notre lanceur, cela inclut des méthodes pour régler la vitesse de chaque roue et une méthode pour arrêter les deux roues à la fois. La méthode Stop() est un choix de design, vous pouvez absolument ignorer cette méthode et il suffit d'appeler les méthodes de définition de vitesse de chacune des deux roues partout où vous voudriez les arrêter.

5.1.4.4 Membres de classe

```
36  private:
37   // Components (e.g. motor controllers and sensors) should generally be
38   // declared private and exposed only through public methods.
39   frc::PWMSparkMax m_launchWheel;
40   frc::PWMSparkMax m_feedWheel;
```

Le lanceur KitBot a deux moteurs, de sorte que notre classe a deux objets membres pour se référer à eux.

5.1.5 PWMLauncher.cpp

5.1.5.1 Instructions Include et Using, constructeur de classe

Les premières sections de ce sous-système sont très similaires au sous-système PWMDrivetrain. Voir la section 5.1.1 pour une description plus détaillée de chacune de ces parties du code.

5.1.5.2 Méthodes – Fabrique de commandes

```
20  frc2::CommandPtr PWMLauncher::GetIntakeCommand() {
21    // The startEnd helper method takes a method to call when the command is
22    // initialized and one to call when it ends
23    return StartEnd(
24        // When the command is initialized, set the wheels to the intake speed
25        // values
26        [this] {
27          SetFeedWheel(kIntakeFeederSpeed);
28          SetLaunchWheel(kIntakeLauncherSpeed);
29        },
30        // When the command stops, stop the wheels
31        [this] { Stop(); });
32  }
```

La commande elle-même est créée à l'aide d'un assistant de commande inline (dans ce cas StartEnd). Pour voir les différentes options disponibles, consultez APIDocs for the Subsystem (voa) et recherchez les méthodes qui retournent un objet CommandPtr.

Dans ce cas, nous utilisons StartEnd() qui appelle le premier paramètre lorsque la commande démarre et le deuxième paramètre lorsque la commande est interrompue (par exemple, par une nouvelle commande planifiée ou un bouton libéré). Lorsque la commande démarre, nous voulons commencer à faire tourner les deux roues vers l'intérieur à une vitesse spécifique pour faire entrer la pièce de jeu et quand elle se termine, nous voulons arrêter les roues.

La technique de programmation utilisée ici s'appelle une « expression lambda ». Pour en savoir plus sur les expressions lambda, consultez cette section des documents WPILib sur les expressions Lambda en C++.

5.1.5.3 Méthodes – Contrôle des équipement

```
34  // An accessor method to set the speed (technically the output percentage) of
35  // the launch wheel
36  void PWMLauncher::SetLaunchWheel(double speed) {
37    m_launchWheel.Set(speed);
38  }
39
40  // An accessor method to set the speed (technically the output percentage) of
41  // the feed wheel
42  void PWMLauncher::SetFeedWheel(double speed) {
43    m_feedWheel.Set(speed);
44  }
45
46  // A helper method to stop both wheels. You could skip having a method like this
47  // and call the individual accessors with speed = 0 instead
48  void PWMLauncher::Stop() {
49    m_launchWheel.Set(0);
50    m_feedWheel.Set(0);
51  }
```

Le reste de la classe de sous-système est des méthodes d'accès au matériel. Ici, nous définissons toutes les méthodes que les commandes peuvent avoir besoin d'appeler pour obtenir l'état ou effectuer des actions sur le sous-système. Pour notre lanceur, cela inclut des méthodes pour régler la vitesse de chaque roue et une méthode pour arrêter les deux roues à la fois. La méthode stop() est un choix de design, vous pouvez absolument ignorer cette méthode et il suffit d'appeler les méthodes de définition de vitesse de chacune des deux roues partout où vous voudriez les arrêter.

5.1.6 CANLauncher

Cette classe est le sous-système du lanceur si vous avez câblé votre contrôleur de moteur par filage CAN. Si vous avez câblé votre contrôleur de moteur à l'aide de CAN, consultez la section 3.4 pour obtenir des renseignements sur la façon de porter en commentaire l'utilisation de la classe PWMLauncher et la remplacer par la classe CANLauncher.

Cette classe est très similaire à la classe PWMLauncher définie ci-dessus avec l'ajout des limites de courant décrites dans CANDrivetrain. Dans ce cas, les limites de courant correspondent à la valeur par défaut du SPARK MAX, elles sont donc incluses à titre d'exemple.

5.2 Commandes

Cette section couvrira les 3 fichiers de commande dans le dossier commands. Des commandes inline supplémentaires sont définies dans le fichier RobotContainer, elles seront couvertes dans la section suivante.

5.2.1 Autos

Le fichier Autos est un exemple d'une "Fabrique de commandes statique". Votre programme ne doit jamais créer un objet Autos (comme le montre le constructeur affichent simplement un message d'erreur) au lieu de cela, vous devez appeler des méthodes de classe statiquement en utilisant la

syntaxe de type autos ::ExampleAuto(). Cette structure est l'un des moyens de définir des groupes complexes qui impliquent plusieurs sous-systèmes (bien que notre exemple ici ne soit pas complexe et ne nécessite qu'un seul sous-système). Le fichier .h pour cette classe est assez simple, il n'est donc pas représenté ici.

```
 9  frc2::CommandPtr autos::ExampleAuto(PWMDrivetrain* drivetrain) {
10    // Drive half speed reverse for 1 second and then stop.
11    return frc2::cmd::Run([drivetrain] { drivetrain->ArcadeDrive(-.5, 0); },
12                          {drivetrain})
13        .WithTimeout(1.0_s)
14        .AndThen(frc2::cmd::Run([drivetrain] { drivetrain->ArcadeDrive(0, 0); },
15                                {drivetrain}));
16  }
```

Notre exemple de fichier n'a qu'une seule routine autonome à obtenir. Vous pouvez facilement étendre ce modèle en ajoutant des méthodes supplémentaires pour définir plus de routines autonomes et vous pouvez choisir entre elles à l'aide d'un SendableChooser sur le tableau de bord.

Cette routine autonome simple demande au robot de reculer pendant 1 seconde à 50% de puissance en utilisant le décorateur WithTimeout pour définir un délai d'attente de 1 seconde sur la commande de pilotage. Il utilise le décorateur AndThen pour dire au robot d'arrêter de bouger une fois la première commande terminée. Les différents types de compositions de commandes intégrées via des décorateurs et des méthodes Fabrique sont décrits dans la page groupes de commande.

5.2.2 LaunchNote

La commande LaunchNote est le premier de deux exemples dans ce projet d'une commande en tant que classe indépendante définie dans son propre fichier. Cette structure est généralement très utile pour les commandes complexes, mais peut également être utilisée pour des commandes simples, comme ici, en fonction des préférences. La commande LaunchNote exécute les deux roues du lanceur pour éjecter la pièce de jeu du robot. Cette classe est assez simple, donc certains éléments sont omis ici.

5.2.2.1 LaunchNote.h – Définition de classe

```
20  class LaunchNote : public frc2::CommandHelper<frc2::Command, LaunchNote> {
```

La première ligne de cette section définit la classe comme une extension de CommandHelper. Cette classe de base définit de nombreuses méthodes d'assistance utilisées pour gérer l'obtention et la définition des exigences, l'interaction avec le Planificateur et la gestion des décorateurs qui modifient la commande ou la collectent dans un groupe. Comme indiqué dans le commentaire au-dessus de cette ligne, il est important de s'étendre à partir de CommandHelper et non directement à partir de Command.

5.2.2.2 LaunchNote.cpp – Constructeur

```
12  LaunchNote::LaunchNote(PWMLauncher* launcher) : m_launcher(launcher) {
13    // indicate that this command requires the launcher system
14    AddRequirements(launcher);
15  }
```

Ce constructeur prend le sous-système comme paramètre afin qu'il puisse être enregistré localement. La dernière ligne du constructeur indique que cette commande requiert ce sous-système. Cette déclaration d'exigences est ce qui maintient l'architecture orientée commandes ensemble. Toute commande qui exécute des actions de sortie sur un sous-système doit requérir ce sous-système pour aider le Planificateur à maintenir une seule commande simple contrôlant un sous-système à la fois.

5.2.2.3 LaunchNote.cpp – Méthodes de cycle

```
18  void LaunchNote::Initialize() {
19    // Set the wheels to launching speed
20    m_launcher->SetLaunchWheel(kLauncherSpeed);
21    m_launcher->SetFeedWheel(kLaunchFeederSpeed);
22  }
```

Viennent ensuite les méthodes « cycliques » de la commande. Ce sont les méthodes qui sont appelées lorsque la commande est planifiée (Initialize), pendant qu'elle est en cours d'exécution (Execute, IsFinished) et quand elle est terminée ou interrompue (End). Ces méthodes sont mises de côté dans la classe de commande de base et sont généralement remplacées par des classes enfants (ceci est indiqué par l'annotation de remplacement à la fin de la déclaration de méthode dans le fichier .h). Pour la classe LaunchNote, nous définissons les roues pour une certaine vitesse lorsque la commande est initialisée. Nous pouvons le faire parce que la vitesse souhaitée ne changera pas pendant l'exécution de la commande. Si vous vouliez modifier la vitesse pendant l'exécution de la commande (en fonction d'une entrée de joystick par exemple), le réglage de ces vitesses en exécution serait plus approprié.

```
25  void LaunchNote::Execute() {
26    // There is nothing we need this command to do on each iteration. You could
27    // remove this method and the default blank method of the base class will run.
28  }
```

Étant donné que notre commande LaunchNote utilise une seule vitesse ajustée durant initialize(), nous n'avons rien à faire ici dans la méthode execute. Cette méthode est appelée à chaque exécution du planificateur (généralement toutes les 20ms) pendant que la commande est en cours d'exécution.

```
30  // Called once the command ends or is interrupted.
31  void LaunchNote::End(bool interrupted) {
32    // Stop the wheels when the command ends.
33    m_launcher->Stop();
34  }
```

La commande End() est appelée quand la commande est retirée du Planificateur. Vous devriez effectuer tout « nettoyage » nécessaire sur votre mécanisme ici. Souvent, mais pas toujours, vous voudrez arrêter le mécanisme dans cette méthode, comme c'est notre cas ici. Le planificateur passe une valeur booléenne pour indiquer si la commande a été interrompue ou non (non interrompu = terminée d'elle-même avec retour de True depuis IsFinished()). Dans notre cas, nous ne nous soucions pas de savoir si la commande a été interrompue ou non, nous voulons arrêter les roues lorsque la commande se termine.

```
36  // Returns true when the command should end.
37  bool LaunchNote::IsFinished() {
38    // Always return false so the command never ends on it's own. In this project
39    // we use the scheduler to end the command when the button is released.
40    return false;
41  }
```

La méthode IsFinished() est également appelée à chaque exécution du planificateur, après Execute() pour vérifier si la commande a terminé son exécution. Dans notre cas, nous voulons que les roues continuent de fonctionner tant que les pilotes maintiennent le bouton pressé, nous retournons donc toujours False pour indiquer que la commande n'est pas terminée. Le Planificateur gérera l'annulation de la commande lorsque le bouton sera relâché (ceci sera couvert dans la section suivante sur RobotContainer).

5.2.3 PrepareLaunch

La commande PrepareLaunch est le deuxième de deux exemples dans ce projet d'une commande en tant que classe indépendante définie dans son propre fichier. La commande PrepareLaunch ne fait tourner que la roue extérieure du lanceur pour lui permettre d'atteindre la vitesse suffisante avant le lancement. Le code de commande PrepareLaunch est presque identique à LaunchNote, à deux exceptions près.

```
18  void PrepareLaunch::Initialize() {
19    // Set launch wheel to speed, keep feed wheel at 0 to let launch wheel spin
20    // up.
21    m_launcher->SetLaunchWheel(kLauncherSpeed);
22    m_launcher->SetFeedWheel(0);
23  }
```

Dans la méthode Initialize(), la vitesse de la roue de lancement (roue extérieure) est définie, mais la roue d'admission est ignorée, de sorte que la Note n'est pas encore introduite dans la roue de lancement.

```
32  }
34    \\ in order for launch
33    \\ Do nothing when the command ends. The launch wheel needs to keep spinning
35  void PrepareLaunch::End(bool interrupted) {
```

Dans la méthode End(), nous n'arrêtons pas les roues car nous avons besoin que la roue de lancement continue de fonctionner dans le cadre de la séquence de lancement. Cela signifie que nous devrons

gérer l'arrêt de la roue si la séquence est interrompue lors de l'exécution de cette commande; ce que nous couvrirons dans la section RobotContainer 5.5 ci-dessous.

5.3 Constantes

Cette classe contient des constantes nommées utilisées ailleurs dans le code. Les sous-classes sont utilisées pour organiser les constantes en groupes distincts, dans notre cas par sous-système. Le fichier contient des commentaires pour indiquer ce que les constantes représentent.

5.4 Robot.cpp

Ce fichier est identique au modèle orienté commandes par défaut. Vous trouverez une description des éléments de la classe Robot dans Structurer un projet orienté commandes.

5.5 RobotContainer

La classe RobotContainer est l'endroit où les instances des sous-systèmes et contrôleurs du robot sont déclarées et où les commandes par défaut et les mappages des boutons aux commandes sont définis.

5.5.1.1 RobotContainer.h – Définition de classe, constructeur et méthode Auto

```
25  class RobotContainer {
26   public:
27    RobotContainer();
28
29    frc2::CommandPtr GetAutonomousCommand();
```

Cette section de code déclare la classe et le constructeur et déclare une méthode publique pour obtenir l'AutonomousCommand qui doit être utilisé. Cette méthode est appelée à partir de Robot.cpp lorsque le robot passe en mode autonome pour déterminer quelle commande exécuter. Voir la section 6.4 pour plus d'informations sur le développement des modes autonomes.

```
31  private:
32   // Replace with CommandPS4Controller or CommandJoystick if needed
33   frc2::CommandXboxController m_driverController{
34       OperatorConstants::kDriverControllerPort};
35   frc2::CommandXboxController m_operatorController{
36       OperatorConstants::kOperatorControllerPort};
37
38   // The robot's subsystems are defined here...
39   //PWMDrivetrain m_drivetrain;
40   CANDrivetrain m_drivetrain;
41   //PWMLauncher m_launcher;
42   CANLauncher m_launcher;
```

La section définit les variables membres de classe. Pour RobotContainer, cela inclut généralement tous vos sous-systèmes et périphériques de contrôle. Ce code utilise CommandXboxController pour représenter les manettes de jeu car il contient un certain nombre de méthodes d'assistance qui

facilitent grandement la relation des commandes aux boutons. Nous les définissons comme privées. Si des choses en dehors de RobotContainer ont besoin d'y accéder (comme des commandes), elles doivent accepter un pointeur vers elles comme paramètre ou des méthodes doivent être ajoutées au RobotContainer pour exposer les informations nécessaires (comme un axe de joystick).

```
13  RobotContainer::RobotContainer() {
14    // Initialize all of your commands and subsystems here
15
16    // Configure the button bindings
17    ConfigureBindings();
18
19    m_drivetrain.SetDefaultCommand(frc2::cmd::Run(
20        [this] {
21          m_drivetrain.ArcadeDrive(-m_driverController.GetLeftY(),
22                                   -m_driverController.GetRightX());
23        },
24        {&m_drivetrain}));
25  }
```

Le constructeur contient un appel à configureBindings() que nous allons couvrir ci-dessous. Cette méthode est utilisée pour configurer les liaisons de boutons. Elle définit également la commande par défaut pour le sous-système de la base pilotable. Vous êtes invités à mettre ce code dans la méthode ConfigureBindings si cela vous semble plus logique, ou même à ajouter une nouvelle méthode juste pour configurer les commandes par défaut !

Nous voulons qu'une commande s'exécute sur notre base pilotable pour nous permettre de piloter le robot avec des joysticks chaque fois que nous n'avons pas d'autre commande utilisant la base pilotable (comme la commande ExampleAuto). Pour ce faire, nous utilisons la méthode SetDefaultCommand() du sous-système. Cela définit la commande qui s'exécutera chaque fois que le Planificateur ne voit rien d'autre s'exécuter sur ce sous-système.

Pour configurer la commande, nous utilisons une définition de commande inline à l'aide de la classe frc2 ::cmd ::Run. La classe Run est utilisée pour transformer un appel de méthode unique en commande. La méthode que nous transmettons à Run est insérée dans la section execute() du cycle de la commande tel que discuté dans la section LaunchNote, ce qui signifie qu'elle sera appelée à plusieurs reprises tant que la commande est planifiée. La méthode d'exécution est à nouveau capturée à l'aide d'une expression Lambda.

Dans notre code, la méthode que nous voulons appeler est la méthode ArcadeDrive du sous-système de la base pilotable. Pour le mouvement avant / arrière, nous passons la valeur de l'axe Y (vertical) du joystick gauche du contrôleur, mais nous en inversons le signe. C'est parce que les joysticks définissent généralement pousser le joystick comme négatif et tirer le joystick comme positif (un résultat de l'utilisation originale dans des simulateurs de vol). Nous voulons pousser le joystick pour faire avancer le robot, donc nous inversons la valeur. Similairement pour la valeur de virage où nous inversons l'axe X (horizontal) du joystick droit du contrôleur. Le joystick considère pousser vers la droite comme une valeur positive, mais les classes WPILib considèrent la rotation dans le sens des aiguilles d'une montre (ce à quoi on s'attendrait en poussant le joystick à droite) comme négative.

```
27  void RobotContainer::ConfigureBindings() {
28    // Configure your trigger bindings here
29
30    m_operatorController.A().WhileTrue(
31        PrepareLaunch(&m_launcher)
32            .WithTimeout(LauncherConstants::kLauncherDelay)
33            .AndThen(LaunchNote(&m_launcher).ToPtr())
34            .HandleInterrupt([this]() { m_launcher.Stop(); }));
35
36    m_operatorController.LeftBumper().WhileTrue(m_launcher.GetIntakeCommand());
37  }
```

La méthode configureBindings() est l'endroit où nous plaçons tout le code structurant qui indique aux commandes quand s'exécuter.

Tout d'abord, nous mettons en place un groupe de commande inline pour exécuter une séquence de lancement tandis que les pilotes maintiennent le bouton A du contrôleur enfoncé. Nous utilisons d'abord la méthode d'assistance A() sur le contrôleur pour obtenir un déclencheur (Trigger), puis nous utilisons la méthode WhileTrue() pour exécuter la commande pendant que le bouton est maintenu, et en l'annulant lorsque le bouton est relâché. Vous pouvez voir d'autres options disponibles pour les méthodes Trigger sur la page Trigger API Doc (voa) ou sur la page WPILib Liaison de commandes à des Triggers.

La première commande de notre séquence est PrepareLaunch(). Nous utilisons ensuite le décorateur WithTimeout() pour que cette commande soit exécutée pendant une durée fixe avant de se terminer. Le décorateur AndThen() est utilisé pour exécuter la commande LaunchNote après l'expiration du délai d'attente. Enfin, le décorateur HandleInterrupt() est utilisé pour fournir une méthode à exécuter si la commande est interrompue ; cela est nécessaire pour s'assurer que les roues s'arrêtent si les pilotes lâchent le bouton pendant que la commande PrepareLaunch est en cours d'exécution car la méthode End() de cette commande n'arrêtera pas les roues par elle-même.

6 Apporter des modifications

Cette section détaille quelques modifications possibles que vous voudrez peut-être apporter au code KitBot et fournit quelques références sur la façon de réaliser ces modifications.

6.1 Modification des boutons d'action

L'une des modifications les plus faciles à apporter au code du robot orienté commandes est de changer quels boutons ou quels comportements de bouton contrôlent une commande. Les commandes utilisées dans le KitBot 2024 ne se terminent pas (isFinished renvoie toujours false) elles ne doivent donc généralement être utilisées qu'avec le comportement whileHeld(), mais changer les boutons auxquels elles correspondent peut être fait très simplement.

Les mappages de boutons dans l'exemple de code sont effectués vers la fin de la méthode configureBindings() dans le fichier RobotContainer. Les liaisons utilisées pour ce projet sont effectuées à l'aide des méthodes d'assistance de la classe CommandXboxController. Ces méthodes d'assistance existent pour chaque bouton du contrôleur et renvoient un objet Trigger qui peut ensuite être utilisé pour spécifier un comportement pour la liaison.

Comme prévu, le code connecte le bouton **A** à la séquence de tir et la gâchette avant gauche leftBumper à l'admission d'une pièce de jeu. Pour les changer, il suffit de changer les méthodes d'assistance a() ou leftBumper(), pour la méthode d'un des autres boutons ! Vous pouvez voir toutes les options disponibles en parcourant CommandXboxController API Docs (voa) pour des méthodes qui ne prennent aucun paramètre et retournent un objet Trigger.

Par exemple, pour remplacer la commande d'admission de la gâchette avant gauche par le bouton **X**, remplacez simplement le leftBumper() par x().

```
35    //Before
36    m_operatorController.LeftBumper().WhileTrue(m_launcher.GetIntakeCommand());
37    //After
38    m_operatorController.X().WhileTrue(m_launcher.GetIntakeCommand());
```

6.2 Modification du comportement des joysticks de pilotage

Un autre changement facile à faire est de modifier quels joysticks du contrôleur sont utilisés pour le pilotage du robot et comment. Le code fourni effectue ce mappage lors de la configuration de la commande par défaut de la plateforme pilotable en haut de configureBindings() dans RobotContainer.

L'exemple de code utilise l'axe Y du joystick gauche pour avancer et reculer et l'axe X du joystick droit pour tourner. Ceux-ci peuvent facilement être échangés ou déplacés sur un seul joystick ! Pour examiner les options disponibles, recherchez les méthodes qui renvoient un **double** dans CommandXboxController API Docs (voa) Pour effectuer ce type de modification, localisez l'appel de méthode que vous souhaitez modifier, tel que getLeftY(), et remplacez-le par la nouvelle méthode souhaitée, telle que getRightY().

Exemple : changer la propulsion avant-arrière sur l'axe Y du joystick droit et laisser la rotation sur l'axe X du joystick droit.

```
19      m_drivetrain.SetDefaultCommand(frc2::cmd::Run(
20          [this] {
21            m_drivetrain.ArcadeDrive(-m_driverController.GetRightY(),
22                                     -m_driverController.GetRightX());
23          },
24          {&m_drivetrain}));
25    }
```

Vous pouvez également modifier les valeurs de joystick. Une modification courante consiste à porter au cube les valeurs. Cela préserve le signe de la valeur (positif reste postif, négatif reste négatif) et la valeur maximale ne réduit pas la vitesse maximale du robot tout en fournissant moins de sensibilité pour de faibles entrées, permettant potentiellement un contrôle plus précis à basse vitesse. Pour effectuer ce type de modification, vous pouvez appliquer la modification de la valeur du joystick là où elle est obtenue.

Exemple : modification uniquement de l'axe de rotation au cube :

```
19      m_drivetrain.SetDefaultCommand(frc2::cmd::Run(
20          [this] {
21            m_drivetrain.ArcadeDrive(-m_driverController.GetLeftY(),
22                                     -pow(m_driverController.GetRightX(), 3));
23          },
24          {&m_drivetrain}));
```

Une autre modification courante consiste à réduire par défaut les valeurs, mais à autoriser la valeur maximale si vous appuyez sur un bouton (mode turbo). Ce type de modification peut également être effectué au point de lecture, bien qu'à mesure que la complexité augmente, vous voudrez peut-être passer d'une définition de commande inline à un type différent où vous pourrez définir le comportement de commande plus clairement.

Exemple : Mise à l'échelle de la vitesse avant-arrière de 50 % à moins que la gâchette avant droite ne soit enfoncée.

```
19      m_drivetrain.SetDefaultCommand(frc2::cmd::Run(
20          [this] {
21            m_drivetrain.ArcadeDrive(-m_driverController.GetLeftY()*m_driverController.GetRightBumper()?1:.5,
22                                     -pow(m_driverController.GetRightX(), 3));
23          },
24          {&m_drivetrain}));
25    }
```

Cet exemple utilise deux choses spécifiques qui méritent d'être expliquées. La première est qu'il utilise la méthode GetRightBumper sur le contrôleur de pilote. Cela renvoie la valeur booléenne du bouton plutôt qu'un objet Trigger associé au bouton comme RightBumper() le ferait.

L'autre chose que cet exemple utilise est l'opérateur ?, appelé opérateur ternaire (voa). Cet opérateur nous permet d'écrire une simple déclaration « if » de manière très compacte, si la gâchette avant droite est enfoncée, nous multiplions par 1, sinon, par 0,5.

6.3 Modification du type de propulsion

Le dernier changement probable que nous couvrirons est le passage de la propulsion Arcade Drive à Tank Drive. Contrairement à le propulsion Arcade qui mappe un axe de joystick à la rotation et un axe de joystick au déplacement avant / arrière, le propulsion de style tank mappe un axe de joystick (généralement l'axe Y) de chaque côté d'une plateforme pilotable différentielle. Pour effectuer ce changement, vous devrez aller au-delà de RobotContainer car les sous-systèmes de transmission fournis ne proposent pas une méthode de propulsion de style Tank. Dans le sous-système de plateforme pilotable approprié (PWMDrivetrain ou CANDrivetrain), faites une nouvelle méthode appelée TankDrive(). Assurez-vous de l'ajouter aux fichiers .h et .cpp ! Cette méthode devrait ressembler beaucoup à la méthode ArcadeDrive. Ensuite, modifiez le mappage de commandes par défaut dans RobotContainer pour utiliser cette nouvelle méthode avec l'axe de joystick approprié.

Exemple:

```
25      void TankDrive(double left, double right);
```

```
27  void PWMDrivetrain::TankDrive(double left, double right) {
28    m_drive.TankDrive(left, right);
```

```
19    m_drivetrain.SetDefaultCommand(frc2::cmd::Run(
20        [this] {
21          m_drivetrain.TankDrive(-m_driverController.GetLeftY(),
22                                 m_driverController.GetRightY());
23        },
24        {&m_drivetrain}));
25  }
```

6.4 Développer des routines autonomes

Le code fourni contient un mode autonome très basique qui roule vers l'arrière à 1/2 de puissance pendant 1 seconde. Des modes autonomes supplémentaires peuvent être développés, soit en ajoutant des méthodes supplémentaires dans le fichier Autos (voir l'exemple de projet Hatchbot Inlined (voa) pour un exemple avec un code autonome plus complexe), soit en créant des fichiers séparés pour chaque routine autonome (voir Hatchbot Traditional (voa) pour un exemple de cette approche).

Il est courant (mais certainement pas nécessaire !) d'avoir plusieurs routines autonomes que vous voudrez peut-être exécuter en fonction de différents emplacements de départ ou différentes stratégies. Si vous poursuivez dans cette voie, le moyen le plus courant de sélection pour chaque match est de choisir entre elles à l'aide d'un SendableChooser sur le tableau de bord.