The Compass Alliance Pathways: Programming

This pathway will cover the basics of programming in FRC. It will focus on text-based languages (such as C++ or Java/Kotlin), although the vast majority of the concepts discussed have their counterparts in LabVIEW.

Level One: Getting Your Robot Up and Running

1. **Picking a Programming Language: Java, C++, or LabVIEW**
    - **Java** is a textual language that is very commonly taught at high schools and is used for the AP CS exams. It is a "safe" language in that runs in its own virtual environment. Unfortunately, this virtual environment, also known as the JVM, means that Java programs are noticeably slower than their lower-level counterparts (like C++) when used for highly intensive tasks. Java is used primarily because of its ease of use, and cross-compatibility. Teams that use Java include 254, 125, 503, 4911, and 1241.
    - **C++** is a very fast textual programming language. It is used in industry for real time systems because of its efficiency, but the learning curve is often much steeper than Java. C++ is also riddled with unusual, or undefined behavior, which can often be difficult for new teams to debug. Many FRC teams use it simply because of its speed, as well as its extensive mathematical libraries. Teams that use C++ include 971 and 1678.
    - Unlike Java and C++, **LabVIEW** is a graphical programming language developed by NI. The programming languages used for FLL are derivatives of LabVIEW, so students coming from Lego programs may be more comfortable starting off with this. Additionally, NI provides extensive debugging tools for Labview. However, LabVIEW comes with its own steep learning curve, due to its dependency systems, odd graphical interface, unneeded verbosity, and slowness. Teams that use Labview include 624.
    - **Choosing what language always depends on what is easiest for *your* team.** For example, it can often make sense to use C++ simply because one of your programming mentors knows C++. On the other hand, it might make sense to use Java because it is easiest to learn without the help of a mentor. No matter what your decision, remember that the choice of programming language is specific to the working environment and people of your team.
2. **Teaching the Programming Language**
    - When teaching programming for FRC, there are two distinct subjects that need to be taught. The first is the semantics and syntax of the programming language itself, and the second is interfacing with FRC components. A guide on learning the C++ language can be found here, Java here, and LabVIEW here.

3. **Picking a robot class**
    - There are four different “classes” that can be used when interfacing with the Robot. A comparison of these classes, and what they mean can be found here.
4. **Once your team has picked a programming language and a robot class, the WPI Screen Steps are a good resource on how to set up your development environment and easily get code onto your robot. These guides are invaluable for FRC programming.**
    - Java
    - C++
    - The official 2019 build season uses gradleRIO and the VSCode editor with the WPILib plugin. More information can be found here. GradleRIO, and a guide to set it up can be found here. The remainder of this guide will be easier to follow with a gradleRIO setup.
5. **Getting code onto your robot!**
    - If using gradleRIO, just type `./gradlew deploy`, and your code will be on the robot.
    - But wait! Your code doesn’t *do* anything yet. Some simple examples for drive code can be found here for Java, and here for C++. The code in the snippets belong in either `Robot.java`, or `Robot.cpp`, which should be auto-created with the gradle/Eclipse project.
6. **Code for mechanisms**
    - Simple drive code can be directly copy-pasted for Java/C++ from here.
    - Most FRC robots have actuated mechanisms other than the drivetrain. This could be anything from a spinning flywheel to a pneumatic catapult. All these mechanisms should be controllable in autonomous, or in teleop. To mechanisms using speed controllers over PWM, there is a guide for C++ and Java here.
    - If using speed controllers over CAN, you must either follow the guide here to treat them as PWM speed controllers, or use the Phoenix API, whose documentation is linked here.
7. **Autonomous**
    - A guide for how to do autonomous actions in FRC programming can be found here.
    - Team 1619 has also compiled some simple code to cross the auto line in Java, which can be found here.

Level Two: Custom Architecture, and Closed-loop Motor Control

1. **Using a custom architecture**
    - Many times, the available robot classes are not enough. For example, you might want to run teleop periodically, and autonomous sequentially. If this is the case, it is likely time to move to a custom architecture.
    - A custom architecture is essentially structuring all the code in a customised way.

- Some examples of custom architectures include 1678's code which is here. 1678's code builds off of 971's code which is here.
- 254 also has a custom architecture. Their 2018 code can be found here.

2. **PID Control**
   - PID Control allows you to control a mechanism based on position, rather than voltage. Using PID, you can tell an arm to turn to 30 degrees, instead of telling it to directly output a voltage. This is especially useful in autonomous. Being able to tell a robot to drive 5 metres instead of full power for 0.5 seconds allows for enhanced repeatability.
   - Some useful documents for PID are:
     - Wesley's Blog
     - CSIM's PID for Dummies
3. **Motion Magic (CAN Only)**
   - If using a TalonSRX speed controller, it is recommended to use MotionMagic for controlling mechanisms, especially something like an arm or an elevator. MotionMagic is essentially a 1KHz PID loop following autogenerated trapezoidal motion profiles. If those words make no sense, don't worry! See the above for information on PID, and here's a document explaining motion profiles.
   - The documentation for Motion Magic is located here.

Level Three: Advanced Drive Paths, MP Control, and Unit-testing

1. **Drive paths and following them**
   - Sometimes raw PID isn't enough for controlling the drivetrain autonomously. For example, you might want the robot to go around the switch and pick up a cube from behind. A clean way to do this would be to create a drive path. A drive path is essentially a set of points that the drivetrain PID loop will follow, and the points will lead to the eventual goal. Jaci from WPILib has created a tool called PathFinder which generates such paths and saves them to a parsable file, which can be found here.
   - Once the points have been generated, there are a variety of ways to follow them. These range from using PID to directly follow the points, to adding a path following algorithm to process the points before giving them to the PID loop. An example of such a path following algorithm can be found here (eqn 5.12).
2. **Model based control**
   - Model based control is a step beyond PID. It allows for keeping a mathematical model of the system in the code, and updating the model with sensor data. Using such a model, one can control a mechanism's position, velocity, acceleration, etc much more precisely. Some teams that use model based control include 1678 and 971.
   - Useful resources for learning model-based control are:
     - Wesley's Blog

- This MIT handout

3. **Unit-testing**
   - Often-times, you want to test your code before deploying it on the robot. This can prevent disaster. Unit-testing is a term for testing portions of the code as standalone programs. For example, you might want to test the portion of the code that runs the elevator, but not the part that makes a few lights flash. Testing mechanisms for FRC is greatly enhanced with model based control, as the model can be used as a simulation of the mechanism, meaning that the whole mechanism can be tested with incredible robustness. Some useful unit testing libraries include:
     - GoogleTest

Appendix A - Revision History

Revision #	Revision Date	Revision Notes
1.0	Sept. 2018	Initial Release