Robot Autonomous and Autonomy

By Noah Gleason and Eli Barnett

Summary

- What do we do in autonomous? (Overview)
- Approaches to autonomous
  - No feedback
    - Drive-for-time
  - Feedback
    - Drive-for-distance
    - Drive, turn, drive
    - Path following
    - Pursuit control
- Structuring autonomous code
  - Code reuse and iterative development

What do we do in autonomous?

- FRC games vary, but FRC autonomous tasks are usually highly similar. Goals to accomplish include:
  - Drive to spot on field
  - Pick up gamepiece from field
  - Deliver gamepiece to goal
    - Vision target usually present
  - Improve robot position in preparation for the start of teleoperated control.
- The most fundamental problem to tackle is *autonomous movement*. There are several approaches, which we will discuss (roughly) in order of increasing complexity.

Simplest possible (useful) autonomous motion: drive for time

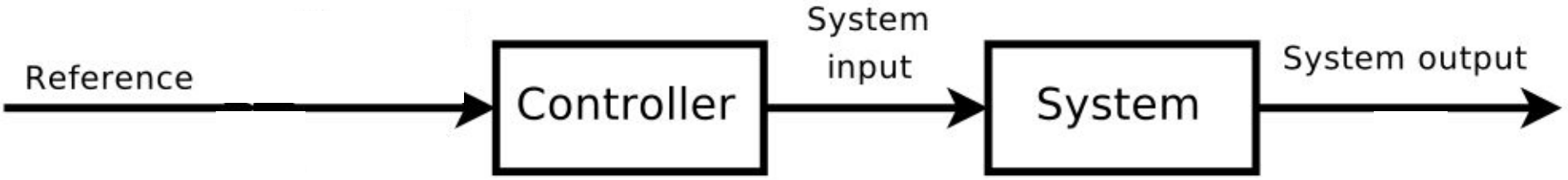
- Store start time in a variable.
- Drive at fixed speed until current time minus start time exceeds some value, then stop driving.
- Will get “mobility points,” possibly place the robot in a better position for start of teleop. However:
  - Won’t drive straight
  - Won’t drive a consistent distance (depends on battery voltage, etc)
  - *Not easily built-upon for more complicated autonomous tasks!*
- Don’t do this.

What else can we do?

- Why is the “drive for time” auto so ubiquitous, if it is so inflexible?
  - Answer: Doing more requires *feedback*.
- The *crucial first step* towards a more-ambitious autonomous routine is the use of drive encoders.
  - Many types of quadrature encoders available in modern FRC: US Digital, Greyhill, CTRE
- FRC programming frameworks make the use of encoders relatively painless.
  - Can even run feedback “locally” on motor controllers
    - CANTalon SRX is a very powerful tool due to its 1Khz loop rate

Simplest feedback-incorporating autonomous: Drive for distance

- Same approach as “drive for time,” but check encoder distance rather than clock for determining when to stop driving.
  - Still won’t drive straight
    - If you can’t drive straight, you can’t predict robot position
    - Not good enough for any nontrivial autonomous routine
- To drive straight, we need to employ...feedback loops!
  - To fully take advantage of sensor feedback, you must be able to use it in real-time to correct your outputs.
  - PID loops are the easiest solution
    - In practice, try for P or PD loops. Integrators *stink!*



From https://commons.wikimedia.org/wiki/File:Feedback_loop_with_descriptions.svg

Driving straight with feedback loops

- Simplest way to drive straight with only encoder feedback is to close a PID loop on the difference in encoder readings between left and right side of drive, and feed the output in opposite directions to each side.
  - Watch the inversion!
  - Can also close on gyro heading
- Velocity servo for each side of drive can also work
  - Must drive at somewhat low speed to avoid angular errors while accelerating
  - Can be combined with previous solution via “cascading” loops
- Tuning takes time - plan for it!

Feed-Forward terms

- Minimizing the error your PID loop has to compensate for keeps you more accurate
- If you can make a “guess” at required loop output, this reduces the amount of work the loop has to do.
- For a velocity servo, can determine the required feedforward from the motor voltage balance equation:

$$\text{Voltage} = K * \text{rotor speed} + IR_{\text{windings}}$$

Feedforward, continued

- In terms of robot movement, we can re-write the previous equation:

$$\text{Voltage} = K_{vel}*\text{velocity} + K_a*\text{acceleration} + \text{intercept}$$

- Just need to determine K_{vel}, K_a and intercept
  - Run robot with a slow voltage ramp to determine K_{vel} and intercept
  - Run robot at fixed voltage to determine K_a, throw it all into multilinear regression
- In practice, K_{vel} term is much bigger than K_a term.
  - Most implementations only allow for Voltage = K_{vel}*velocity; need to “hack” a bit or roll your own code for fully-featured implementation.

Great, but what if a straight line doesn't cut it?

- Driving straight is a huge step forward, but in order to accomplish most higher-complexity auto tasks, we must be able to *turn*.
- Two approaches: gyro and Non-gyro.
  - With a gyroscope, you can simply close a PID loop on the desired heading, and feed the output in different directions to each side of the drive.
    - Can be difficult to tune without oscillations - watch for "stiction" effects, consider cascading to velocity servos.
    - Likely more accurate.
  - Without a gyroscope, you can "drive to distance" but with each side in opposite directions.
    - Must measure "effective wheelbase," which has problems with sharp turns
    - Exactly analogous to "drive to distance" otherwise.

A note on camera integration

- Turning is a natural time to introduce camera integration.
  - Many FRC vision targets are visible after a short drive forward, and require only two straight drives to reach.
  - Obtain desired heading for turn from camera code
  - Turn as usual, then continue with driving.
- Possible to run camera code while driving, but harder; must deal with camera jitter, more sensitive control loop tuning.

Drive, turn, drive: now we're getting somewhere!

- Now we have the "building blocks" for a reasonably-repeatable drive, turn, drive autonomous.
  - Can follow any piecewise-linear path, which is often good enough.
  - Can integrate camera feedback to determine magnitude of a needed turn.
- Not quite ideal, however…
  - Driving straight at fixed speeds presents a problem: when accelerating, wheels might slip
    - Encoders can't detect wheel slip!
    - Limits you to low speeds
    - Need to "ramp" speed somehow...

Motion profiles: learning to love Pathfinder

- The traditional way to “ramp” the velocity is with a trapezoidal-acceleration motion profile
  - Limits jerk, which results in smoother and more repeatable motion.
  - Requires either position or velocity servo to follow profiles
    - Use feedforwards for good results! (This is true of all control loops, really)
- Can easily add profiling to simple drive, turn, drive autonomous through the Talon SRXs Motion Magic mode
- However, if we can follow a profile, why not follow a spline?
  - Jaci’s “pathfinder” (Java/C++) makes generating spline profiles very easy
  - Must use “effective wheelbase,” be careful of units

Shameless plug for the Talon SRX

- Easily the best FRC motor controller on the market, handles much of the work for you
- Kilohertz-frequency control loops are easier to tune, more stable
- Direct encoder readings to controller
- Lots of cool control modes (Motion Magic, motion profiling, more to come…)
- Battery voltage compensation (easy-to-overlook, very important)
- Minimum voltage output to account for stiction
- Watch out for the units!

Spanner in the works: field tolerances

- So, we now can drive our robot to a desired position, purely by dead reckoning.
  - If implemented well, can achieve tolerances of less than an inch - more than good enough for most game tasks.
- However, while your autonomous path might have a half-inch tolerance, the field does not.
  - You *must* have a solution to deal with field-to-field variability. Two approaches:
    - Measure the field. Parameterize your autonomous routine to a field of arbitrary dimensions, and input the measured ones at each competition.
    - Position robot relative to goal. Requires a procedure for accurate robot positioning.

Pursuit control

- Pure motion profile following can’t account for known errors in robot pose
- A “pure pursuit” controller dynamically recomputes the desired robot path based on a current pose estimate.
  - Paths robot to some point on the originally-specified path, some fixed distance ahead of the current nearest point on the path to the robot (another parameter to tune!)
  - Very simple paths can work (254 uses simple circular arcs; cannot match end angle, does not matter due to dynamic recomputing)
  - Requires a pose estimate...

Pose Estimation

- Combine gyro, encoder, and possibly camera data to get an (x, y) position for the robot
- Assuming you have a low-jerk profile, encoder is fairly accurate
- Adding camera can account for wheel slip (if it occurs) and inaccuracies in the field geometry
- Implementations can range from very simple (use the “more trusted” source of pose estimate if it appears to not be erroneous) to complex (Kalman filter).

And now for something completely different: Structuring autonomous code for re-use

- If you rewrite your autonomous code from scratch every year, you are wasting development time reinventing the wheel.
  - Good autonomous code is flexible - it can be used on multiple robots, and reconfigured to run multiple paths.
  - Initial development time increases, but *huge* time savings in the long-run.
  - Time saved is time spent on new functionality
- Loose coupling!
  - Write your autonomous code so that it “doesn’t know” what specific robot it is running on.
  - Separate high-level “behavior” from low-level “implementation”
    - OOP: Use interfaces!
    - The “command-based” java framework can be very powerful here.
    - Consider the “dependency injection” design pattern

Flexible autonomous code example

```
public Auto2017Center(
            Command driveToPeg,
            Command dropGear,
            MappedDigitalInput dropGearSwitch,
            Command driveBack) {
      addSequential(driveToPeg);
      if (dropGearSwitch.getStatus().get(0)) {
            addSequential(dropGear);
      }
      addSequential(driveBack);
    }

new Auto2017Center(new RunLoadedProfile(drive), new ActuatePiston(gearHandler), new
MappedDigitalInput(1),new ExecuteProfile(drive, “backupFromPeg.csv”))
Or…
new Auto2017Center(new DriveForDistance(drive, 5), new ActuatePiston(gearHandler), new
MappedDigitalInput(1),new DriveForTime(drive, 0.5))
```

Planning Autonomous

- Work on a simple but effective auto first
  - E.g. same side switch
- Start adding more complex autos for harder tasks
  - Motion Profiling scale
- If possible, line up for what you want to do to start teleop
  - For scale, we tried to pick up a cube after
- Tune shooter power instead of driving distance

Automation in teleop

- In our experience, drivers hate turn-to-angle buttons
- For field-oriented OI, “snap” angular setpoint to relevant angles
- In theory, if your pose estimation and pursuit control are working, you can push a button to go to a location
- Hitting walls messes up encoder readings, ultrasound/IR sensors can help
- Motion profiling and PID tuning techniques are also applicable to arms and elevators