Rerouting: Autonomous Collision Detection & Recovery in Botball

Sihou “Keson” Jin | Radiant 926
07/09/2026

The Problem

In Botball, something in your run is bound to go wrong. Your claw dropping a pom, your bot grabbing a cube in the most unstable angle possible, or not getting Botguy because your opponents did it faster. These are all issues that will cost you points in your run. But other accidents, like missing a line follow and veering off, getting your robot’s arm stuck in a jumble of pvc pipes, or accidentally hitting a wall and never recovering—they’re issues that will completely throw off your robot, and might even cause a domino effect onto your other one.

Today, I’ll be proposing a solution that solves the last issue—accidental crashes into walls—that are infamous for the points it has stolen from the Botball community.

I’ve developed a two-part system to fix this:

1. Collision detection with accelerometer readings
2. Collision recovery with odometry

Collision Detection

Collisions are detected using the robot's onboard accelerometer, exploiting the fact that an impact produces a sharp spike in acceleration distinct from the noise of normal cruising.

Cruise Learning

Rather than comparing acceleration against a fixed threshold—which would have trouble transferring across different fields—the system builds an EWMA adaptive statistical model of what cruising motion looks like and flags large spikes coming into the data. This model maintains an exponentially weighted moving average of the mean and variance (from where SD is derived). Recent readings are weighted more heavily than old ones, so the baseline continuously adapts as driving conditions change.

The running mean and variance are exponentially weighted, which means each input’s weight on the model exponentially decays as new inputs are introduced. This makes the model represent a more recent state.

$$mean = \alpha * accel + (1 - \alpha) * mean$$
$$variance = \alpha * diff^2 + (1 - \alpha) * variance$$

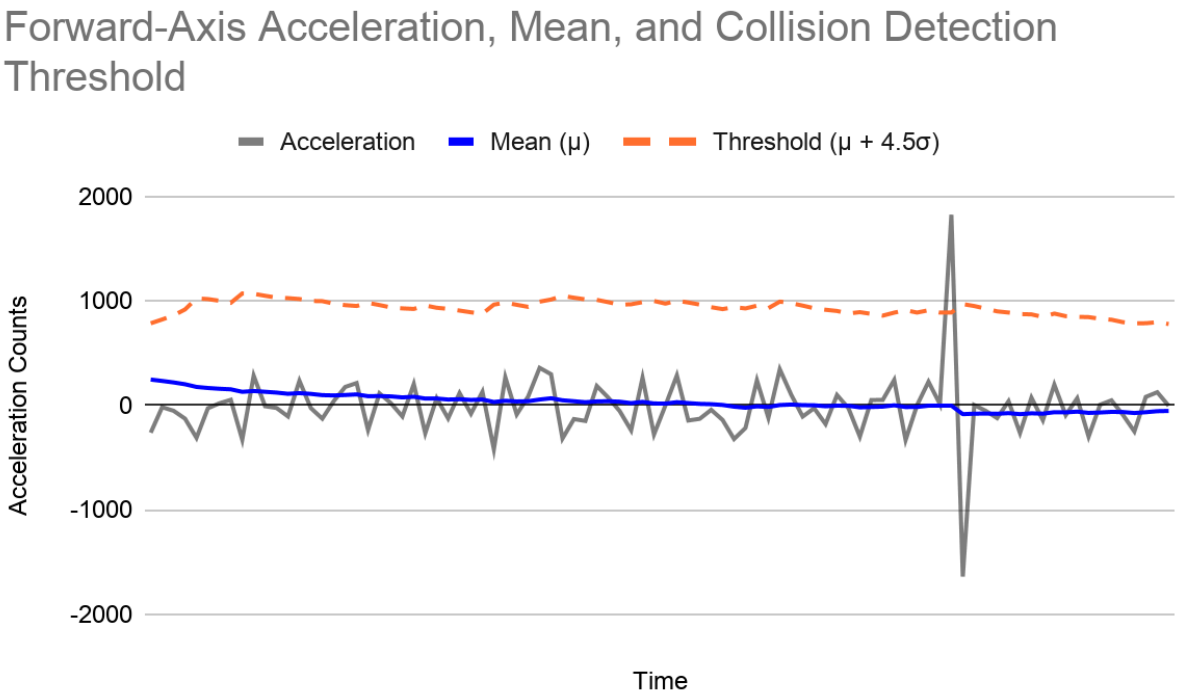
$SD = \sqrt{variance}$

α is the smoothing factor, which I have set to 0.05. A larger α means each input has a larger effect on the model, but noisier data. The first sample will have an effective α of 1.00, where it sets the mean for the model. Variance will be set to 0, however, as there is no variation with a dataset of 1 value.

Each value is also checked for collision. Accelerometer readings are directional: When driving forward it watches for a positive acceleration spike, and when reversing it watches for a negative one, reducing the chance of faulty flags. A collision is declared when a new reading deviates from the running mean by more than κ standard deviations. I've tuned κ to be 4.5.

$input > mean + \kappa\sigma$ (Forward movement)
$input < mean - \kappa\sigma$ (Reversing)

Expressing the threshold in units of standard deviation rather than raw acceleration is key. It makes detection based on the established model. On a smooth surface where acceleration is quiet, the standard deviation is small and even a moderate impact clears the threshold; on a rough surface where the baseline is noisier, the threshold widens automatically, preventing the vibration of normal driving from being mistaken for a crash.



The mean & threshold lines are relatively stable in the run. There is a very noticeable peak in acceleration where it goes well over the threshold, which confidently flags a collision. Behind the positive peak, there is also a negative peak—but it is not as consistent, so it's less reliable.

Other Ways to Reduce Noise & Prevent False Positives

1. The first 10 samples of every new model cannot be flagged as a collision.
The model is not developed enough to reliably flag spikes (SD is unstable), so it simply doesn't.

2. The model is cleared when new conditions arrive.
Whenever a new movement starts or command speed changes sharply, the model resets to clear values.

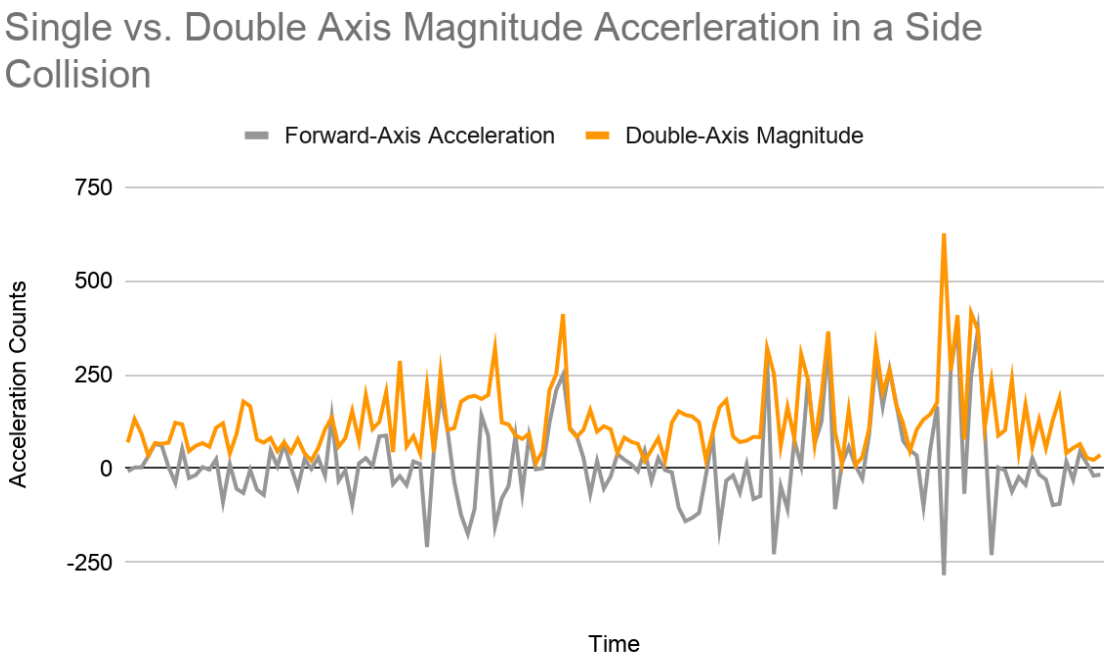
3. Tune your PID.
Especially if your robot runs arcs or line follow, a smooth PID reduces baseline noise, creating a larger distinction between cruising and collision.

Double Axis Detection (Alternate Solution)

Collecting data from both horizontal axes might seem pointless, but it theoretically allows the detection of outside forces colliding. To implement this solution, the magnitude of the horizontal axes is taken:

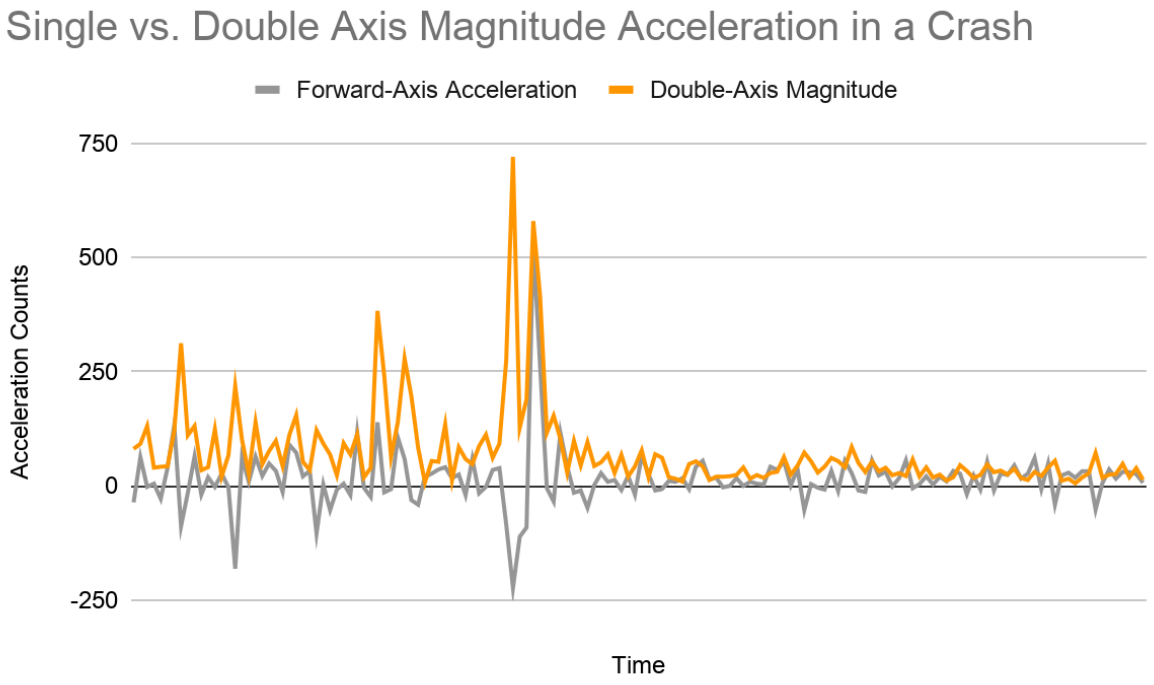
$$magnitude = \sqrt{accel_1^{\,2} + accel_2^{\,2}}$$

Be aware that squaring takes away negative values. Using horizontal magnitude will remove the ability to use direction to more accurately flag crashes.



This graph is acceleration values when the robot is pushed from the side, simulating another robot colliding with it. The double-axis approach has a slight noticeable peak, while the single

axis approach has near no noticeable peak. This difference is too minimal to be spotted with the current detection model. I decided not to pursue this path due to its inability to spot crashes from spiking.



The major tradeoff to this solution is that its baseline is far noisier due to values being squared. In the line chart, the spikes in the double-axis approach are approximately double the peaks of baseline noise, while there is a larger difference between the spike and the baseline in the single-axis approach. The ambiguity poses a small threat to accurate detection.

However, the baseline of the double-axis experiences a significant noise reduction after the crash. This is most likely because the side-axis, which experiences acceleration from vibration, is stabilized when pushing into a wall. This is arguably a more reliable system of detection than spiking for the double-axis approach, because collisions can be detected from sustained inputs well below the mean (which can be evaluated with $mean - \kappa\sigma$).

The major downside of this approach is that by the time that a collision can be reliably detected, the robot's odometry (key to collision recovery) would have been thrown off. If this offset could be calculated in a way, this solution could somewhat be viable.

To implement either approach, you must determine your horizontal or forward axes of your robot, which depends on your wombat orientation. This can be tested by rotating your robot and looking at the accelerometer values. Pointing an axis down will cause it to have a larger number than the others, as gravity acts upon the accelerometer.

Collision Recovery

Collision recovery in this system is odometry-based. I'll explain odometry briefly, as it's one of the most widely implemented systems in competitive robotics behind PID.

Odometry

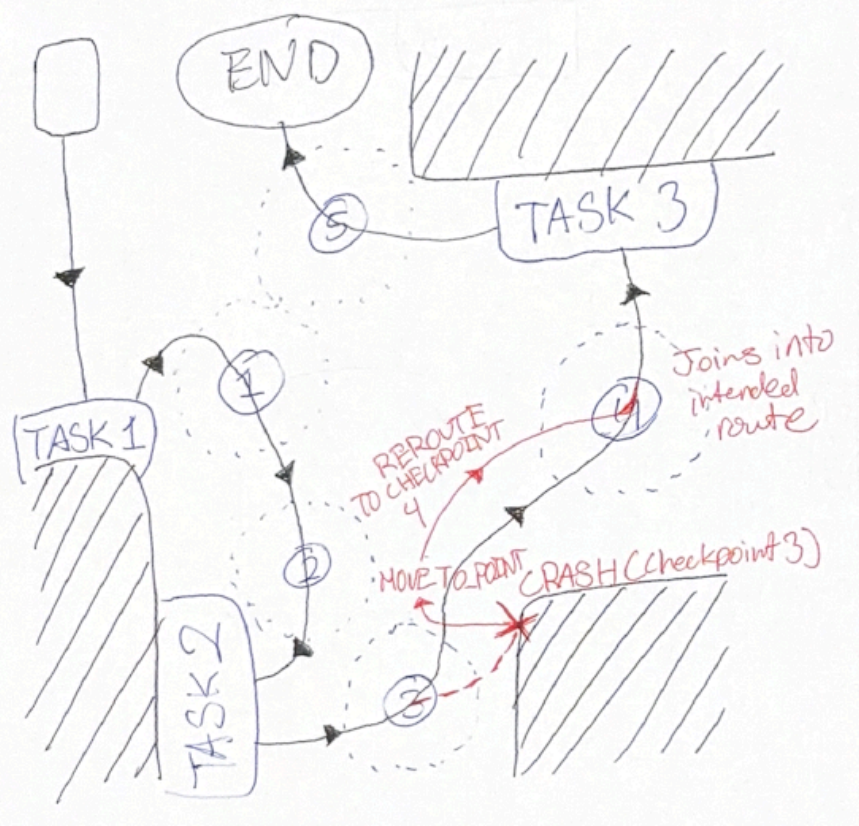
Odometry in Botball uses motor position to keep track of where the robot is on the field, with the predicted position being marked with a xy value. The biggest weakness to odometry is its rapid accumulation of drift. Over time, where the robot thinks it is will deviate from where it actually is, due to wheels slipping or gyroscope inaccuracy. We can reset x or y values when we do line or wall corrections in order to reset odometry and minimize drift.

A key odometry function is used here, called move_to_point (MTP). It commands the robot using its known odometry position to move to a point on the odometry plane.

Checkpoints

When you visualize your robot running its route, you can probably think of a couple of checkpoints it has to cross. If you measure those points, you'll end up with a list of xy values that you can conveniently feed to your robot's odometry system.

With odometry, the robot will be able to detect whether or not it has crossed a checkpoint. This allows the robot to know not just where it is on the field, but how far along it is on the route.



Whenever a collision is flagged, the robot first backs away from whatever it is stuck on. This will prevent odometry from accumulating more error and create some distance. Then, the robot calls MTP 1 or more times to arrive at the next checkpoint.

Checkpoints are most effectively placed before and after tasks, but ultimately, it's based on the route.

Checkpoints should also come with a robot state, which returns the robot to exactly its intended position at that checkpoint in the original route.

Major Areas of Improvement

This system is nowhere near perfect—I've only tested individual parts, which have worked well, but never implemented it on a realistic route before. This paper should serve as a base for people to experiment from and expand on. I would love to see the development of different techniques that approach the problems differently. Some parts of this system are brute-force fixes and could use more proper methods. Others may require a design change. Here are some issues that I deem major in the code:

1. Implementation is difficult.
To have this system working, you'll have to tune and test, and making autos will mean creating a new list of checkpoints. If there was a solution that was easier to implement without costing consistency, it would probably see more implementations being done. At GCER, routes change. You might not have the time to tweak your collision system as well. Checkpoints must also be manually created.

There may need to be extra points as there might not always be a clear path between the stuck robot and the next checkpoint. A path planner application for Botball might be a good idea, but this could also be solved by completely replacing the checkpoint system.

2. Expected acceleration isn't actually measured.
When commanded motion changes, the robot simply just doesn't flag the first 10 samples as collisions. If the robot crashes at the start of a movement, it has no way to detect it. If you could find a way to calculate predicted acceleration, or create some crazy physics model, it could solve this issue.

If you are trying to tackle this problem, I would advise watching out for motor inconsistencies or difficult implementations.

3. There are fixed values, despite being adaptive.
In stuck detection, κ is a fixed value. SD is adaptive by definition, but its coefficient κ may still have to be tuned. With testing & logging, a trend or equation for the optimal κ value could be found. But possibly, the EWMA model could be replaced with some other model more fit for detecting peaks.

That's all. Good luck at GCER!