Introduction to Position Tracking

Revision 1 | Last Modified October 7, 2018

Abstract

This document provides an introduction to and overview of the position tracking and odometry system developed by Team 5225 over the past few years. The document is targeted at VEX team members, specifically programmers, though we've tried to make it understandable to all participants regardless of programming or competition experience; there is no code in this document. Additionally, the applications of any odometry system do, hypothetically, extend beyond the VEX Robotics Competition.

Contents

Background 3
Motivation 3
Units 3
Definitions and Conventions 3
Tracking Theory (aka Odometry) 4
Tracking Using Arcs 4
Proof of Axial Position Independence of Tracking Wheels 6
Tracking Algorithm 7
Writing Motion Algorithms 8
General Programming Tips 9
Conclusion 10

Background

Readers should be familiar with the following concepts before proceeding with this document:

- Vectors in 2D space
- Cartesian (rectangular) and polar coordinates in 2D space
- Basic 2D kinematics (position, velocity, acceleration, etc.)
- Basic geometry concepts such as right-angle trigonometry, arcs, and chords

Motivation

The Absolute Positioning System (APS) had been a concept that our programmers had thought about and discussed for a few years, mostly hypothetically. The APS is a system that keeps track of the absolute position (i.e. cartesian coordinates) and orientation of the robot during a game autonomous or programming skills run. After the conclusion of the Nothing but Net season, our programmers decided to start developing it for real. Over the following ~2 years, we went through several iterations of both the tracking algorithm itself (which uses sensors to keep track of position and orientation), and the coupled motion control algorithms (which use the position and orientation information to follow a planned route). Function calls to the motion control algorithms were then used as building blocks for game autonomous and programming skills runs.

After the use of this system played a major role in us becoming the 2018 World Tournament Champions and Robot Skills World Champions, we've had many requests from various other members of the VEX community for information on how our system was made. While we are always tempted to boil it down to "thousands of hours of time and most of our sanity", we also decided to write and release this document as a resource for all.

Units

We recommend inches as units for distance (and thus position), as all VEX field measurements are in imperial/customary units. We recommend radians for all angles (at least internally), because they are the native output of the algorithm, and also the native unit for built-in trigonometric functions. Above all, however, it is extremely important to use the same units everywhere (i.e. if you use inches, then everything distance-related is in inches, speed is in inches per second or per millisecond, etc.). Not doing so will require additional unit conversions, which are easy to mess up, and also require more calculations than necessary (this can already push the limits of the Cortex M3).

Definitions and Conventions

In this document, we use a column notation for vectors (it looks like a 2x1 matrix); for example, a vector with x-value a and y-value b would be written as:

$$\begin{bmatrix} a \\ b \end{bmatrix} \tag{1}$$

The following variables are used in this document to represent physical parameters and other known values:

- s_L is the left-right distance from the tracking center to the left tracking wheel
- s_R is the left-right distance from the tracking center to the right tracking wheel

- s_S is the forward-backward distance from the tracking center to the back tracking wheel
- $\overrightarrow{d_0}$ is the previous global position vector
- θ_0 is the previous global orientation
- θ_r is the global orientation at last reset

Tracking Theory (aka Odometry)

This is the core of the position tracking system, providing the rest of the robots' code with live data on the current position and orientation (represented as a position vector $\vec{d}$, and orientation θ) of a predefined point on the robot (called the "tracking center", see Figure 1). Note that the tracking wheels can be placed anywhere along the dotted lines without affecting the math; it is the *perpendicular* distance to the tracking center that matters, as explained below.

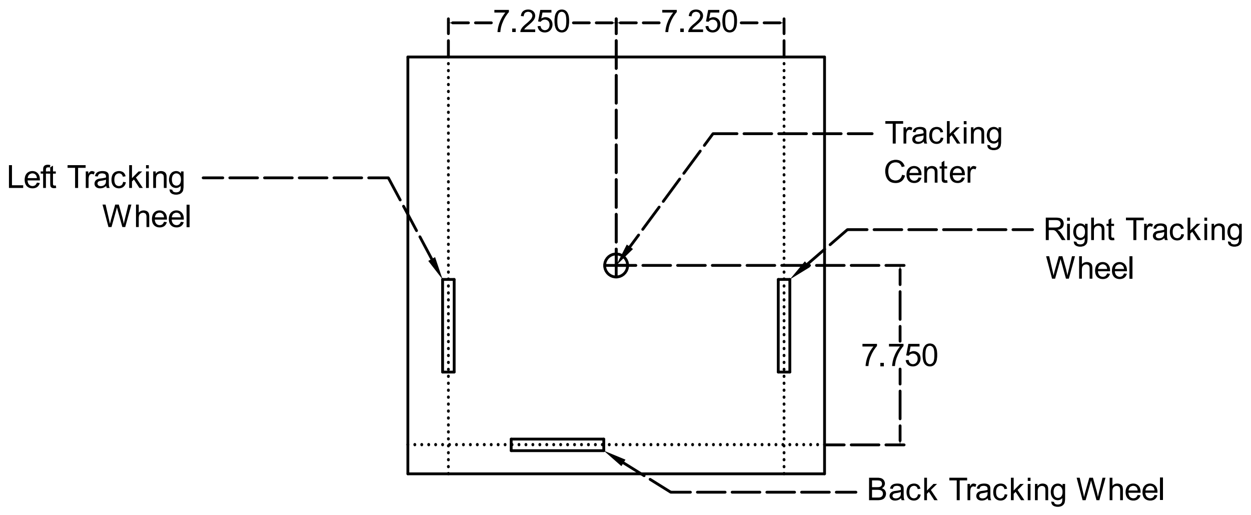

Figure 1: A sample robot base with tracking wheels.

Tracking Using Arcs

Position tracking works by modelling the robot's motion (i.e. the motion of the tracking center) as an arc over an infinitely short time interval. Figure 2 shows the same robot as above following an arc with a 5' radius for 15°. Note that we are assuming for this simple example that the robot does not move toward or away from the arc center during the maneuver, and that the tracking wheels are lined up with the center of the robot; we'll show after that neither of these are necessary. The final robot position is shown in light gray. The left and right wheel paths are both arcs that are concentric with the tracking center's arc. (Concentric means that the arcs have the same center point; this applies in this case, as the robot is effectively rotating around the arc center.)

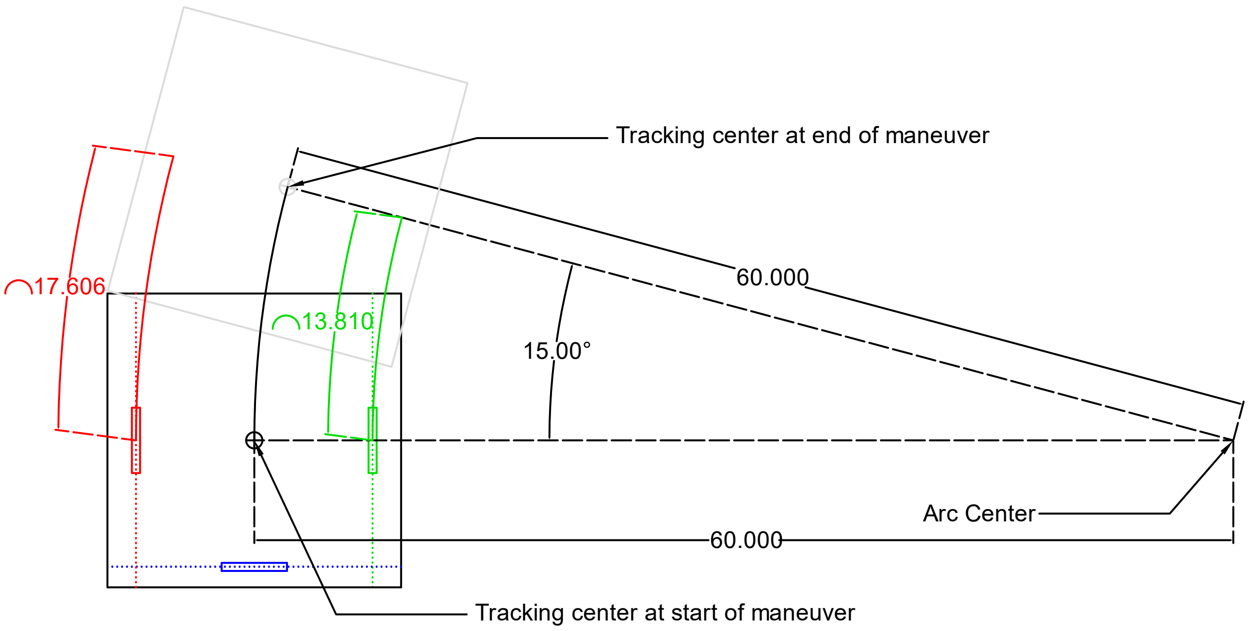
Tracking center at end of maneuver
17.606
13.810
60.000
15.00°
Arc Center
60.000
Tracking center at start of maneuver

Figure 2: A simple example maneuver, with the left and right wheel paths shown.

There are a few important points to realize from this diagram. Firstly, the arc angle is the same as the change in orientation of the robot. This angle (call it $\Delta\theta$) can easily be calculated by starting with the arc length formula:

$$\begin{aligned} \Delta L &= r_L\Delta\theta, & \Delta R &= r_R\Delta\theta \\ \Delta L &= (r_A + s_L)\Delta\theta, & \Delta R &= (r_A - s_R)\Delta\theta \\ \frac{\Delta L}{\Delta\theta} &= r_A + s_L, & \frac{\Delta R}{\Delta\theta} &= r_A - s_R \\ r_A &= \frac{\Delta L}{\Delta\theta} - s_L, & r_A &= \frac{\Delta R}{\Delta\theta} + s_R \end{aligned} \tag{2}$$

These two equations can be combined, eliminating the radius (the only unknown):

$$\begin{aligned} \frac{\Delta L}{\Delta\theta} - s_L &= \frac{\Delta R}{\Delta\theta} + s_R \\ \Delta L - s_L\Delta\theta &= \Delta R + s_R\Delta\theta \\ \Delta L - \Delta R &= \Delta\theta(s_L + s_R) \\ \Delta\theta &= \frac{\Delta L - \Delta R}{s_L + s_R} \end{aligned} \tag{3}$$

Note that this gives a value in radians. Testing it with our example, we get:

$$\begin{aligned} \Delta\theta &= \frac{17.606 - 13.810}{7.250 + 7.250} \\ \Delta\theta &= 0.2618\,rad = 15.00° \end{aligned} \tag{4}$$

As we can see on the diagram, this is correct. This is an example of a "state function"; if you know any two of the starting orientation, ending orientation, and distance between left and right wheel travel ($\Delta L - \Delta R$), you can find the other one. Since any translation without rotation does not change the *difference* between these numbers (it changes both by the same amount), this formula applies regardless of where the robot is on the field, or how it got there. This means that orientation can be calculated as an absolute quantity, rather than a relative change from the last arc approximation.

Also based on this arc, the translation (change in position) can be calculated. To do this, we need to know the radius of the tracking center's arc. This can be calculated as the radius of the right side arc plus the offset of the right tracking wheel from the tracking center:

$$r_R = \frac{\Delta R}{\Delta\theta} + s_R$$

We also need a local coordinate system. Theoretically, any will work, but one that makes the math easier is such that the straight line path from the initial position to the final position is in the positive y direction; this is an offset from the robot's initial "forward" by $\frac{\theta}{2}$. Using this convention, the local translation x-coordinate is zero (as we are still ignoring the back wheel), while the y-coordinate is the chord length. The chord length (the straight-line distance between the two endpoints of the arc) is based on the radius and angle change; both are known, therefore the local y-axis translation can be calculated:

$$\Delta\overrightarrow{d_{ly}} = \begin{bmatrix} 0 \\ 2\sin\frac{\Delta\theta}{2} \times \left(\frac{\Delta R}{\Delta\theta} + s_R\right) \end{bmatrix} \tag{5}$$

Now, we have to consider what happens if the robot strays from the arc somewhat, resulting in additional translation. This forms a second arc, representing a second component of the robot's movement. This time, the axis is offset from the robot's initial "right" by $\frac{\theta}{2}$, making it perpendicular to our y component (and thus representative of our local x axis). This arc's radius and chord length can be calculated in much the same as the y component, giving our complete local translation vector:

$$\Delta\overrightarrow{d_l} = 2\sin\frac{\theta}{2} \times \begin{bmatrix} \frac{\Delta S}{\Delta\theta} + s_S \\ \frac{\Delta R}{\Delta\theta} + s_R \end{bmatrix} \tag{6}$$

This coordinate system is offset by $\theta_0 + \frac{\Delta\theta}{2}$ from the global coordinate system. Therefore, by rotating $\Delta\overrightarrow{d_l}$ back by that amount, the global translation vector can be calculated. The current position at any time is simply the starting position, plus the summation of all global translation vectors up to that time.

Proof of Axial Position Independence of Tracking Wheels

It was mentioned earlier in this document that the tracking wheels can be moved along their line of motion (dotted lines in Figure 1) without affecting the math at all. To demonstrate this, we can prove mathematically that the rotational motion of an omni wheel does not depend on its position along its line of motion; specifically, it depends only on its line of motion.

There are two types of motion that the robot can undergo: translational and rotational. It is trivial to show that the position of the wheel does not matter for translational motion, as the velocity is the same at all points on the robot. However, the proof does get more complicated for rotational motion.

As a scenario, we set up a wheel that is rotating around a point at $\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$ from the center of the wheel (using the forward direction of the wheel as a positive y-axis), as shown in Figure 3.

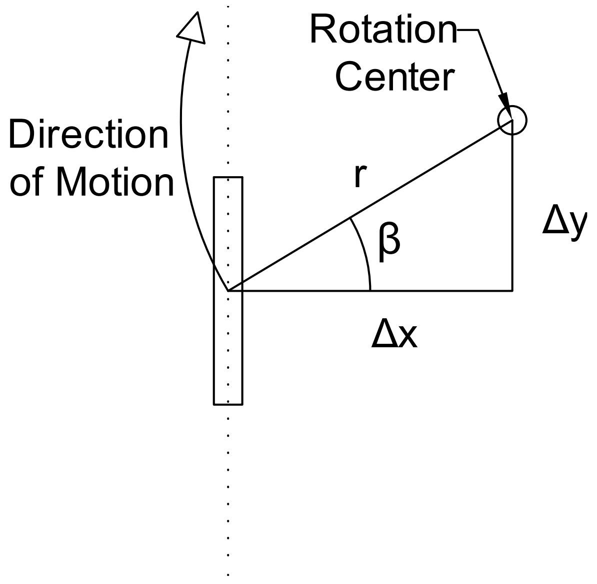

Figure 3: Scenario for position independence proof.

By combining the radius of the arc followed by the wheel with the angular velocity with respect to the rotation center, ω, the tangential velocity of the wheel, v_t can be found:

$$v_t = \omega r \tag{7}$$

This can then be used to calculate the y-component of the velocity (i.e. what the wheel will measure), v_y, and simplified using trigonometric ratios:

$$\begin{aligned} v_y &= v_t \times \cos\beta \\ v_y &= \omega r \times \frac{\Delta x}{r} \\ v_y &= \omega \Delta x \end{aligned} \tag{8}$$

As can be seen, the radius (which is dependent on the y-axis offset of the wheel), cancels out of the equation; the motion of the wheel (as described by its measured velocity) is thus independent of its position along its line of motion.

Tracking Algorithm

The algorithm itself consists of a single procedure to perform calculations, which should be called frequently (a cycle period of no more than 10 milliseconds). We recommend using a dedicated runtime task/thread for this process.

The procedure can be broken down into a few steps:

1. Store the current encoder values in local variables

2. Calculate the change in each encoders' value since the last cycle, and convert to distance of wheel travel (for example, if the wheel has a radius 2" and turned 5°, then it travelled approximately 0.1745"); call these ΔL, ΔR, and ΔS
3. Update stored "previous values" of encoders
4. Calculate the total change in the left and right encoder values since the last reset, and convert to distance of wheel travel; call these ΔL_r and ΔR_r
5. Calculate new absolute orientation $\theta_1 = \theta_r + \frac{\Delta L_r - \Delta R_r}{s_L + s_R}$; please note that the second term will be in radians, regardless of the units of other variables
6. Calculate the change in angle $\Delta\theta = \theta_1 - \theta_0$
7. If $\Delta\theta = 0$ (i.e. $\Delta L = \Delta R$), then calculate the local offset $\Delta\overrightarrow{d_l} = \begin{bmatrix}\Delta S \\ \Delta R\end{bmatrix}$
8. Otherwise, calculate the local offset $\Delta\overrightarrow{d_l} = 2\sin\frac{\theta}{2} \times \begin{bmatrix}\frac{\Delta S}{\Delta\theta} + s_S \\ \frac{\Delta R}{\Delta\theta} + s_R\end{bmatrix}$ (Equation 6)
9. Calculate the average orientation $\theta_m = \theta_0 + \frac{\Delta\theta}{2}$
10. Calculate global offset $\Delta\vec{d}$ as $\Delta\overrightarrow{d_l}$ rotated by $-\theta_m$; this can be done by converting your existing Cartesian coordinates to polar coordinates, changing the angle, then converting back
11. Calculate new absolute position $\overrightarrow{d_1} = \overrightarrow{d_0} + \Delta\vec{d}$

Writing Motion Algorithms

The team has decided not to release the complete motion algorithms that have been created and used by the team. There are several reasons for this, including the ~~hundreds~~ thousands of person-hours that went into their cumulative development, and that while we are more than happy to help out teams that are interested and wish to learn, we do not want to become technical support for a large suite of complex algorithms. Most importantly, the goal is not to release a library for other teams to use; rather, it is to facilitate an incredible learning opportunity that involves complex mathematics and control theory.

That being said, we do want to offer as much of a guideline as possible. Below is an arms-length tutorial for developing such an algorithm:

1. **You'll need at least two algorithms, as a minimum.**
   One for driving to a point, and the other for turning on the spot. Let's start with the **turning on the spot algorithm**.
2. **Pick a convention.**
   It can be more intuitive to specify angles in degrees, but remember that everything uses radians internally. Pick one or the other as the units for angles passed as parameters to your functions, and stick with it. Also, try to keep the order of equivalent parameters consistent (X before Y, etc.).
3. **Start with something you've used before.**
   There's no right answer, and the math can get very complicated very quickly, so start with something that you're familiar with (it could be PID or TBH, but a plain proportional loop works just as well to start).

4. **Every situation is different.**
   Your algorithm shouldn't be a black box that you assume works perfectly for every scenario; remember that there's no practical limit on how many parameters a function can have. This doesn't necessarily mean passing custom PID constants (or equivalent) to each call, but you should have some way to fine-tune the algorithm(s) for one particular use case without messing up others. A more customizable algorithm means a more useful one.
5. **Deal in absolutes.**
   Generally speaking, no parameters should indicate a value relative to your current position or orientation; they should be field-centric. This means you don't say "turn 25 degrees to the right", you say "turn to face 90 degrees from the audience direction" or "turn to face the middle post".
6. **Think about making it more versatile in the future.**
   Start with turning to face an orientation, but turning to face a point might be useful too. You also might want to add support for automatically turning to the closest equivalent angle (to avoid doing 270 clockwise when 90 counterclockwise would work). This might require changes to how your code is structured. The more possibilities, the better.
7. **Try a wide range of scenarios to see what should be adjustable.**
   We tested angles from 10 to 360 degrees, with and without a mobile goal, for our final version in In The Zone. In the end, we made a spreadsheet showing 'ideal' parameters to pass depending on the scenario, and they were further tuned based on that for each use.
8. **Take it step by step.**
   Get each feature working before you start on the next. Not only does this help with time and source code management, but you'll probably learn something that might help you with the next feature. Moving onto the **algorithm to drive to a target**; start by driving until the robot is past the target before you worry about correcting angle. Then, consider which of the two major types of algorithms you want: one that continuously tries to point at the target (fastest), or one that tries to follow a predefined line as closely as possible (most consistent). Also, think about what might happen if the robot is too far off, and make sure you can't end up in an infinite loop.
9. **Play it safe.**
   Regardless how perfect your testing environment is, at some point you probably want to use this at competition. The field might be different in ways not as negligible as you might think, or there might be another robot (even your alliance partner) in the way. There's no limit to the number of things that can go wrong at competition, so make sure that you consider what your robot will do in these scenarios. A good starting point is: if you're trying to move but you're not moving, stop the autonomous. It might also be easier to write this as a separate task that is running continuously, regardless of what your autonomous is doing.

General Programming Tips

We've also decided to include some general suggestions that aren't strictly related to position tracking, but (we hope) will help you with this and future projects nonetheless.

- **Make it work. Make it right. Make it fast.**
  This quote, attributed to programmer Kent Beck, provides a good starter priority list for any non-trivial software project. The first priority should be to fulfil the basic functionality

requirements by any means possible. Then, without changing the functionality or core algorithms, rewrite/refactor the code until it is easy to use, easy to understand, and thus easy to maintain (within reason). Finally, make it fast; in this context, this applies to both efficiency of the code (factoring out repeated identical calculations is always a good idea), and to the time it takes to reach the target, which can always be faster. It might be better to apply this philosophy at a smaller scale over each iteration or feature addition of your algorithms, but really it is up to you. As always, there is no one right way to do things.

- **Take advantage of version control systems.**
  We highly recommend that you learn to (properly) use a version control system (VCS) such as Git. Not only is this a great way to maintain backups and allow collaboration, using tools such as 'merge' and 'revert', it is possible to very finely manage changes to the code without messing with source files directly.
- **Use parallel programming wisely.**
  Parallel programming has different names on different platforms. In ROBOTC, it's called "tasks". On modern desktop, mobile and real-time operating systems, including PROS, it's typically called "threads". Regardless, parallelization of code offers many benefits, including having loops waiting on a sensor value; in a traditional, "synchronous" program, this loop would prevent other important code (such as your position tracking algorithm) from running. That being said, it is a very easy technology to overuse. Programs that have too much parallelization can quickly become bug-prone and unmaintainable. If you're relatively new to programming, we suggest limiting (or avoiding entirely) the use of tasks or threads, at least when starting. Regardless, before you start using them heavily, we suggest you read up on race conditions and deadlocks. There are several online resources on those topics, and if in doubt, Wikipedia is always a good place to start.

Conclusion

We hope that you find this document useful and informative. If you have any further questions, feel free to contact us; we are always happy to help people trying to learn. This is the first publication of this sort that the team has released, so we welcome your feedback. You can contact us on the VEX Forum, on the unofficial VEX Teams of the World Discord server, or via email at team5225@gmail.com.