a better way to swerve by Team EverGreen#7112

our solution leads to:

- sharper turns with less drifting out of your path
- less heating and less wear of the drive motor
- more efficient use of battery power
- prettier code with less edge cases that need to be individually addressed
- less mechanical stress on the robot's chassis

this paper requires:

- basic knowledge in swerve programming
- basic understanding of linear algebra
- basic understanding of calculus

Hello, we are team EverGreen#7112, we recently finished working on our swerve drive and have been learning and experimenting with different ideas for controlling the swerve drive for a few months now, we've noticed a pretty simple trick that can help a lot with making sure the swerve drive is more consistent, safe, and generalized without impacting performance, and only requires a few software changes, no need to change anything mechanically, we've tested it and it works very well, and so we decided to share it with all of you.

We've looked through a lot of other teams' code and we've noticed a repeating trend.
After calculating what speed every module should have as a vector, all teams we saw simply calculated that vector's magnitude and fed that as the speed the drive motor should run at. this at first looks correct, of course naturally the speed you'd want that wheel to drive at should be the speed you give the drive motor, however this is a mistake and causes heating and wear to the drive motor, worse turning with more drifting out of your ideal path, inefficient use of battery power, more mechanical stress on the robots chassis, and causes your code to need to address more edge cases individually.

So what is the solution?

First let's start with the basic terminology:

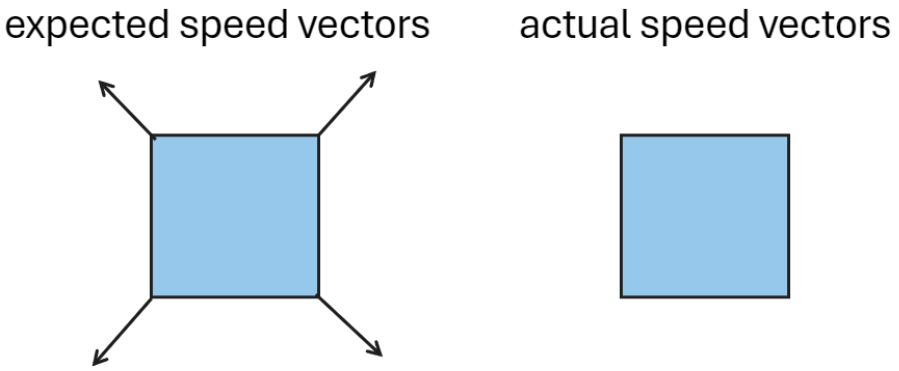
- "target speed vector" is the 2d vector calculated by your code to be the ideal speed and direction in this specific moment in time an individual module needs to drive at. (each module's target speed vector is different and dependent on the action the robot needs to perform at this specific moment in time). denoted T
- "drive motor target speed" is a scalar denoting the speed we want the drive motor to have at a specific moment in time (not necessarily the same as the magnitude of the target speed vector, the rest of this paper will show why it shouldn't be the magnitude of T most of the time). denoted v
- "steering vector" is the 2d vector of magnitude 1 whose direction corresponds to the direction of the module's steering in that specific moment, important to note, it can be calculated by taking the distance (in radians) the steering encoder says the wheel is from 0 and putting the sin of that angle as x and cos of that angle as y. important note, the direction the robot is currently moving is irrelevant, this vector always points

to the direction the expected speed vector would have if the drive motor is given a positive input. denoted D

- "actual speed vector" is the 2d vector that represents the actual speed (and direction) of the module at a specific point in time, it is calculated by multiplying the steering vector by the drive motors speed (read by the encoder). denoted V
- "expected speed vector" - is a 2d vector that represents what we expect the speed vector to be without accounting for the influence of the rest of the swerve system on the actual movement, it is calculated by scaling the steering vector by the drive motor target speed. denoted E
- "target robot speed vector" a 2d vector that represents the speed vector we'd ideally like the robot to have. denoted T_R
- "actual robot speed vector" a 2d vector that represents the speed vector the robot actually has. denoted V_R

now that we have our basic terminology lets understand our goals, the ideal situation is when the target speed vector is the same as the actual speed vector, however for this to occur they both also must be the same as the expected speed vector (since the expected speed vector's direction is equal to the actual speed vector's and for the target and actual speed vectors to be equal the drive motor target speed must be the same as its actual speed i.e the magnitude of the actual speed vector).
and so, in the ideal situation $T = V = E$.

now let's take a look at what happens when that isn't the case:
if $V \neq E$ than that means $||V|| \neq ||E||$ as they have the same direction, that means for some reason the motor didn't drive at the same speed we wanted it to.
An obvious culprit is obviously acceleration and deceleration times, and there's not a lot we can do about them, but there is a third culprit, which is force in the opposite direction to the direction we are trying to move in, generated by the other modules.
example:

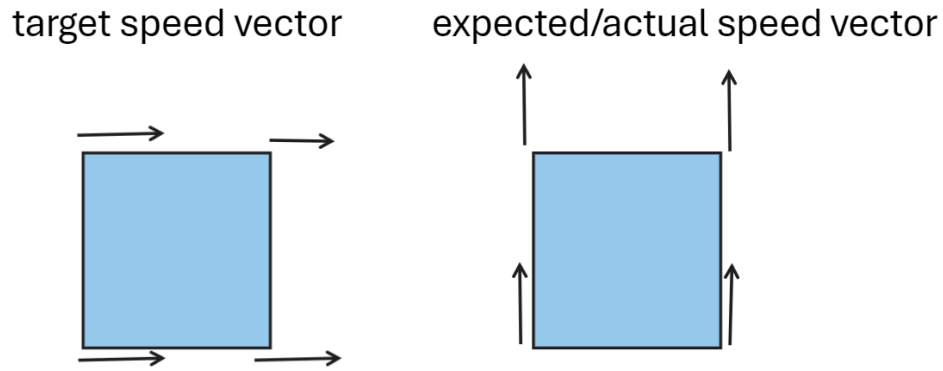


in this example every individual module's speed is canceled by another module's speed, making it so $V_R = 0$ and for every module, even though we are powering the motors $V = 0$
since we are not moving anything all of the energy we are putting to power the motors gets turned to heat energy which heats the motors, this is bad for the motors and a waste of power from the battery, aside from that, all motors working in different directions like that puts a lot of mechanical stress on the robots chassis, the robot is actively trying to tear itself

apart.
in this specific example, no matter the speed we want the robot to move at, as long as we're powering the drive motor and were using $v = ||T||$ we will always have that problem.

now we can identify the problem a bit more easily, our problem is power we input into the motors that ends up not doing any work due to opposing forces from different modules, however this is just a special case in a much more general problem, the general problem can be defined as “modules doing work in an axis they're not supposed to”.

now that we've successfully defined the problem, we can look at another example that is much more likely than the example shown before, this example is the case in which $T \neq E$ in this example the robot starts with $T_R = 0 \quad V_R = 0$ and for every module $V = 0 \quad T = 0 \quad D = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ this is an incredibly likely scenario, we go from this target state to a new target state where $T_R = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, since there is no turning, for every module $V = T_R = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow D = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$.
if we simply go with $v = ||T||$ what we'll end up with is this:



As you can see, the robot is moving in the wrong direction, at least until the steering motors will catch up.
This will lead to accumulation of travel in the y axis all throughout the action of turning by the steering motors, which leads to a drift out of path.
Of course a drift isn't something completely avoidable and it's also calculable and can be compensated for, but it is best practice to try and lower the initial drift as much as you can instead of just compensating for it later..

There are lots of more examples of ways this problem can come into play, the 2 examples given are not necessarily the most common, they are just simple to understand, but now that we understand the problem, what's the solution?

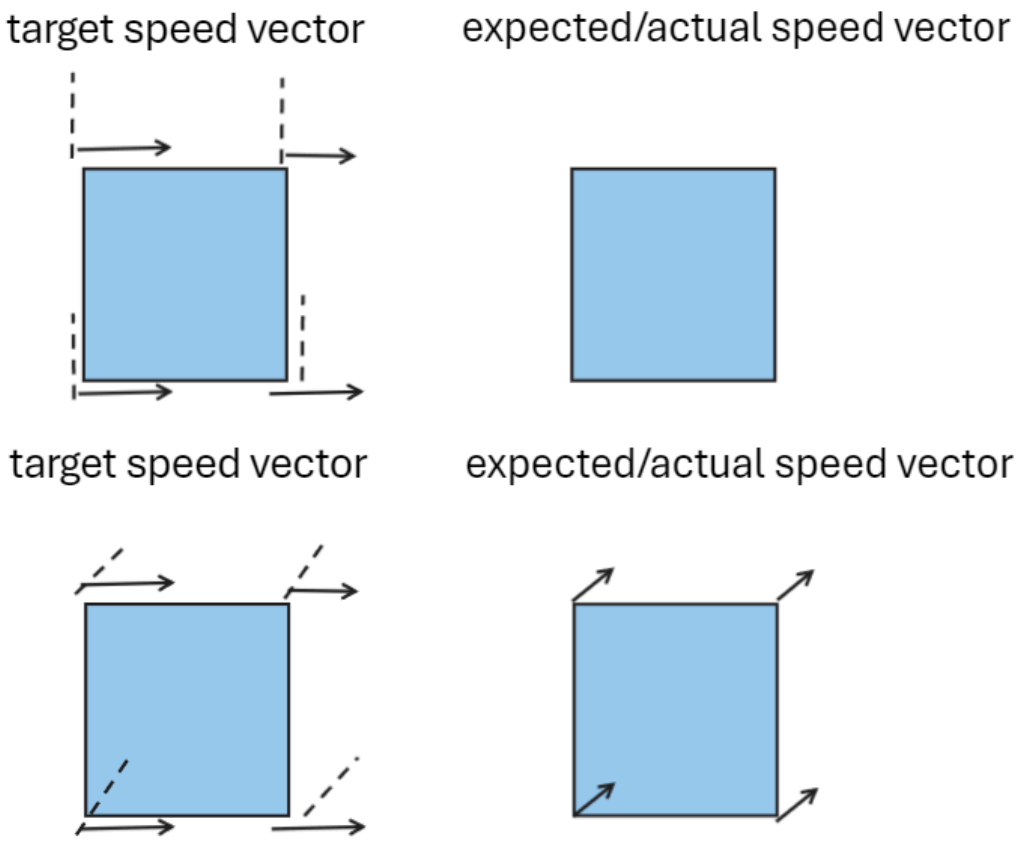
well the solution is actually very simple, instead of using $v = ||T||$, we can use $v = T \cdot D$

but how does that solve it?
well the dot product is actually a sort of projection, due to the fact D is a unit vector, taking the dot product of of T and D is the same as taking the component of T in the axis of D,

meaning we only get the speed we need to drive at in the specific axis we have control of (D).

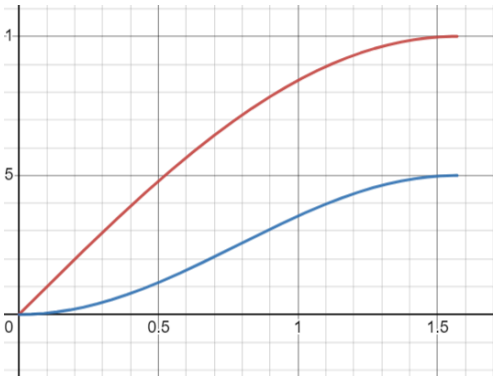
a bonus feature this solution has is this:
most teams would optimize their code so that the robot would never have to make a turn of more than 90 degrees and instead of turning all the way, they'd turn to the opposite point and make sure to just use $v = -||T||$ so the robot would drive in the correct direction, to do that however, most teams would use an if statement and would have to branch the code, using $v = T \cdot D$ however would make it so v would automatically be correctly signed to fit the direction of travel, meaning there's no need to use an if statement.

lets see it in action:



as you can see here, if D is perpendicular to T than E/V would be 0, this is because any movement would not contribute in any way to moving in the desired axis, however as the steering motor begins to catch up the v and therefore E and V begin to increase until eventually once the steering motors reached the right position and then we'll reach $T = V = E$

let's use this example and compare the amount of expected movement in the y axis when using $v = T \cdot D$ and when using $v = ||T||$:
this graph shows the difference in accumulated y axis travel throughout the turn

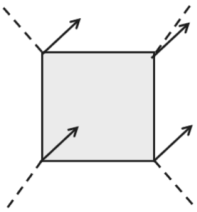


as you can clearly see, the drift caused by using $v = T \cdot D$ is **at most** half the size of the drift caused by $v = ||T||$
to better understand why we can take a look at the derivatives of the drift throughout the turn (or the y velocity throughout the turn) and notice something that should come as no surprise:

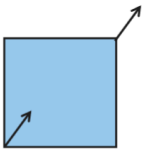
as you can clearly see, the y velocity throughout the turn for $v = ||T||$ grows larger and larger the further you are from the target angle, however for $v = T \cdot D$ you have no y velocity when you're at your furthest from the target angle and throughout the entire turn the y velocity of $v = T \cdot D$ is always lower than $v = ||T||$'s y velocity.
what this goes to show, is that we are now much closer to $E = T$.
of course, a solution that guarantees $E = T$ 100% of the time is not really useful as its something along the lines of $v = 0$ until $D = \hat{T}$

If we go back to the first example, well see:

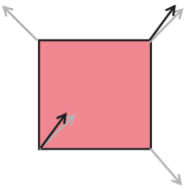
target speed vectors



expected/actual speed vectors



expected and actuall speed vectors



As you may have noticed, we no longer have any clashing speeds that cancel each other out and produce heat.
in fact, since the modules will no longer try to move in an irrelevant axis, as long as the modules' T's arent clashing, we can expect to be closer to $E = V$.
of course, a solution that guarantees $E = V$ 100% of the time is also not useful as its the same solution that guarantees $E = T$ 100% of the time.

link to our swerve repo, you can see our implementation of this trick in the subsystem SwerveModule: https://github.com/EverGreen7112/EverSwerve

Thanks for reading!