Programming a Robot to Go Straight (For Real)

Isaac Yates

Norman Advanced Robotics

Programming a Robot to Go Straight (For Real)

1 Introduction

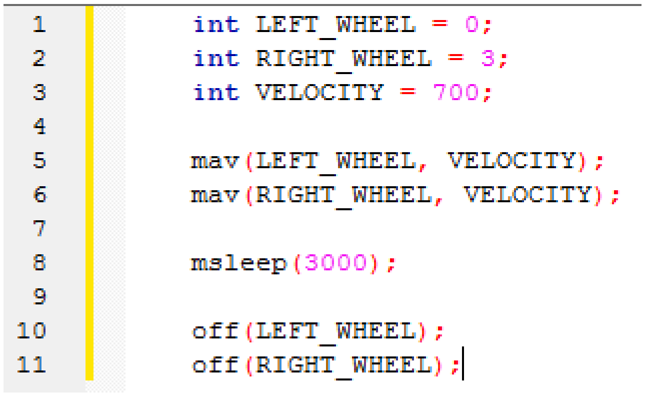
I lazed across the row of chairs at the Global Conference for Educational Robotics, tired out of my mind, half asleep, until the representative from NASA got me up and told me to help him set up another row, chiding me for being unprofessional. Our team, Norman Advanced Robotics, had been working in the living room of one of our senior member's houses during the 3 weeks leading up to the Global Conference for Educational Robotics, instead of finishing up by our own deadline a month before the competition, in the designated room given to us by the leading sponsor to use.

But despite our extra work, I was still not happy with the performance of our robots. To be honest, it wouldn't have mattered of I'd spent another one-hundred hours on the robots, I would still have been getting up early that morning at GCER, changing numbers in the run in a futile attempt to get the robot to go exactly where I wanted it to go, and I awoke far to early to keep my eyes open during the keynote speech at the conference. This is where my following journey of thought began.

1.1 The Big Problem

One of the seemingly hardest things to program a robot to do is drive straight a specified distance. This seems simple to an outsider, someone who has never tried programming robots or never programmed at all.

The logical code to make a robot drive straight for a certain distance seems to be this:



```
int LEFT_WHEEL = 0;
int RIGHT_WHEEL = 3;
int VELOCITY = 700;

mav(LEFT_WHEEL, VELOCITY);
mav(RIGHT_WHEEL, VELOCITY);

msleep(3000);

off(LEFT_WHEEL);
off(RIGHT_WHEEL);
```

There are at least two problems with this code. As the programmer, I may not know it yet, but the robot is in fact NOT going to to go straight using this code. The second problem, which may seem personal (though it really isn't), is that 3000 milliseconds is a terrible way to measure *distance*. Have you ever given someone directions to the bathroom by saying “hey, yeah it's 55 seconds down the hall and to the left!”

Not only are these directions lacking any kind of unit concerned with actual distance, but what if the person trips up and falls halfway to the bathroom? What if he or she trips halfway down the hall, or steps on mega sticky bubble gum and then his/her foot gets temporarily stuck? And say he/she doesn't realize any of this happened, and so keeps counting. This is exactly what happens to robots. You tell them to drive forward for a certain amount of seconds, they get caught on a PVC pole temporarily, and then stop prematurely because they were counting by seconds, not by distance. To be fair, this would happen measuring distance by meters as well, but you still have the issue with a time measurement being dependent on the velocity of the robot, which tends to fluctuate with battery levels.

1.2 Setting my Own Goals

When GCER of 2014 was over, and I took the airplane home, I was thinking about the old, tattered library that NAR had been using for about four years on their robots, a publicly available set of header files named opencode.

After my struggle with simply getting a robot to go straight, I knew that I'd need to study up before trying at it again the next year, but I also knew that I needed to at the very least update the opencode drive-lib, and make it a better, more reliable library.

2 The Problem with User-Set Drift Coefficients

During the fall I mostly played around with using ET sensors to get the robot to drive straight, but the day after the Oklahoma Regionals was over, I was talking with a friend over ways to avoid the old method of driving straight that opencode had adopted over the years, which was using drift coefficients. Drift coefficients look something like this:

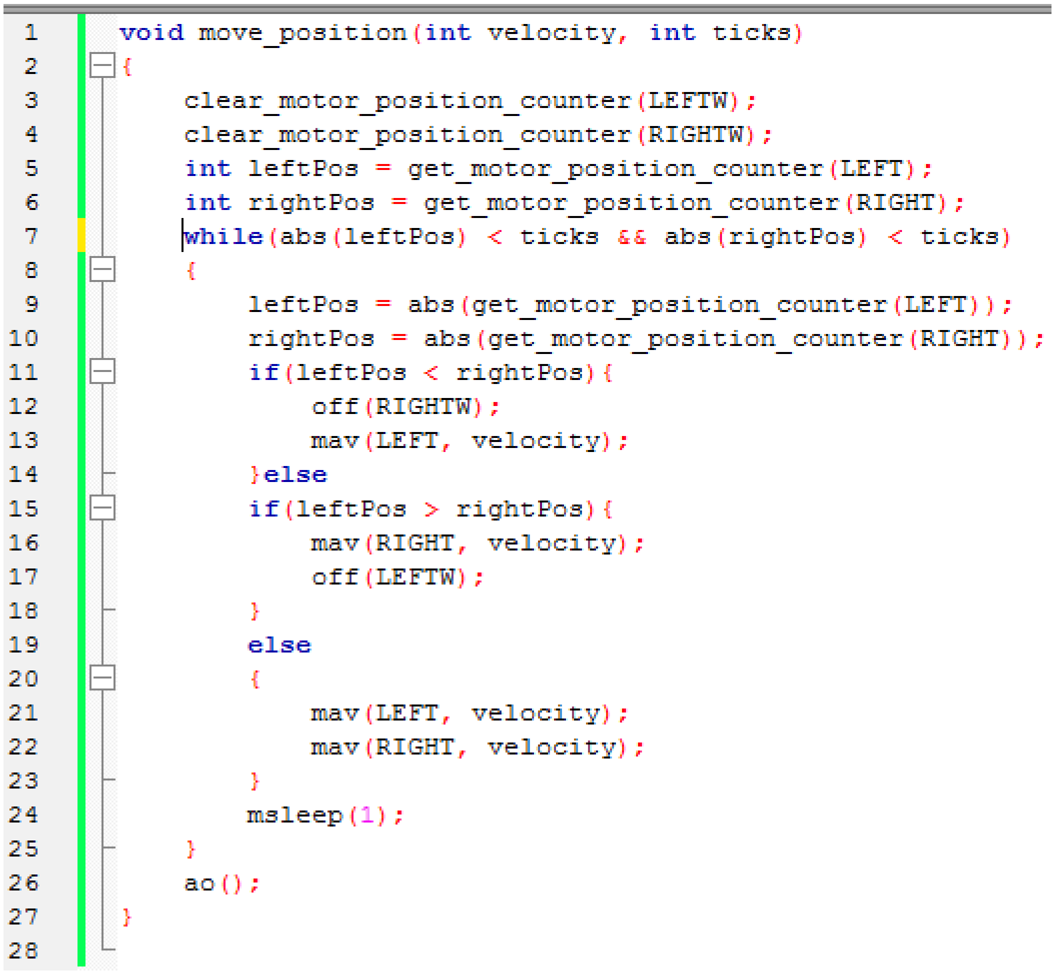
```
float right_wheel_drift = 0.96;
float left_wheel_drift = 1.00;
mav(RIGHT, velocity*right_wheel_drift);
mav(LEFT, velocity*left_wheel_drift);
...etc
```

The big problem with drift coefficients is that they take a lot of test runs to get perfect, and they also can change based on the battery level of the robot. Since motors also often have variable ticks per revolution, the battery problems tend to become ridiculously large. One option was to try to derive an equation for the drift coefficients that takes into account battery power, but I had no idea how long that could take, or whether it would cover all the unknown variables involved with driving a robot straight.

2.1 Using “get_motor_position_counter(#port)” to avoid drift coefficients

During the conversation with a friend after Regionals, I started to think about the functions “get_motor_position_counter(#port)” and “clear_motor_position_counter(#port)”. I went to local coffee shop and pulled out my laptop and started coding without really thinking about what I was doing, and then realized I'd come up with a solution. (Yes it was a very exciting and strange way to solve a problem.) I didn't realize until I had finished the code, but I had actually completely avoided the need to have the programmer calculate the drift coefficients at all. The original code I wrote looked like what is on page 4.

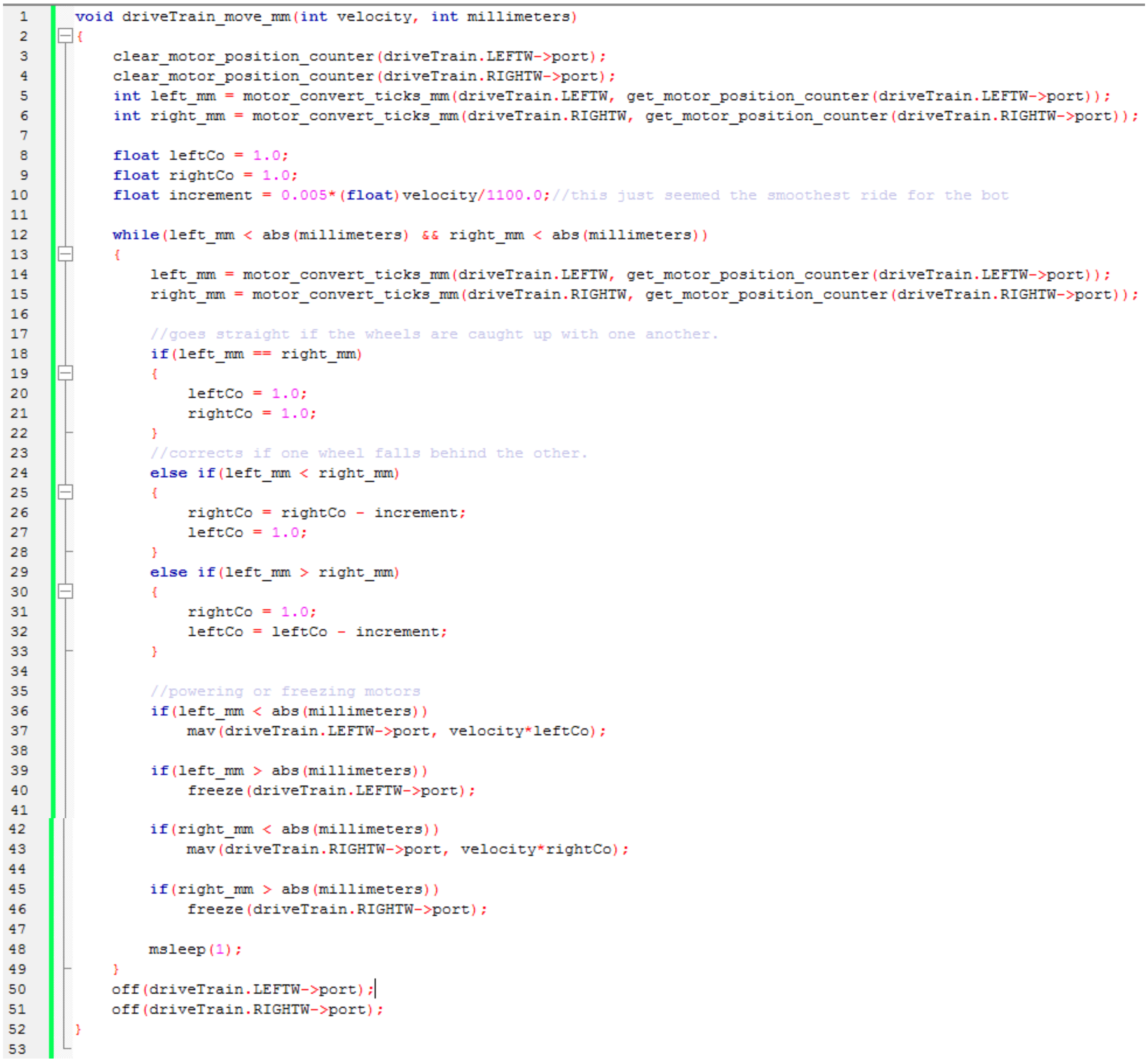
It looks pretty beautiful, but it actually didn't work. The concept was sound, except that the repeated calls to the “off(#port)” function actually made the robot stay still. After about a month of work, I came up with page 5, which is a bit more daunting to read.



```c
void move_position(int velocity, int ticks)
{
    clear_motor_position_counter(LEFTW);
    clear_motor_position_counter(RIGHTW);
    int leftPos = get_motor_position_counter(LEFT);
    int rightPos = get_motor_position_counter(RIGHT);
    while(abs(leftPos) < ticks && abs(rightPos) < ticks)
    {
        leftPos = abs(get_motor_position_counter(LEFT));
        rightPos = abs(get_motor_position_counter(RIGHT));
        if(leftPos < rightPos){
            off(RIGHTW);
            mav(LEFT, velocity);
        }else
        if(leftPos > rightPos){
            mav(RIGHT, velocity);
            off(LEFTW);
        }
        else
        {
            mav(LEFT, velocity);
            mav(RIGHT, velocity);
        }
        msleep(1);
    }
    ao();
}

```

Continue onto next page for more of the paper.



```
void driveTrain_move_mm(int velocity, int millimeters)
{
    clear_motor_position_counter(driveTrain.LEFTW->port);
    clear_motor_position_counter(driveTrain.RIGHTW->port);
    int left_mm = motor_convert_ticks_mm(driveTrain.LEFTW, get_motor_position_counter(driveTrain.LEFTW->port));
    int right_mm = motor_convert_ticks_mm(driveTrain.RIGHTW, get_motor_position_counter(driveTrain.RIGHTW->port));

    float leftCo = 1.0;
    float rightCo = 1.0;
    float increment = 0.005*(float)velocity/1100.0;//this just seemed the smoothest ride for the bot

    while(left_mm < abs(millimeters) && right_mm < abs(millimeters))
    {
        left_mm = motor_convert_ticks_mm(driveTrain.LEFTW, get_motor_position_counter(driveTrain.LEFTW->port));
        right_mm = motor_convert_ticks_mm(driveTrain.RIGHTW, get_motor_position_counter(driveTrain.RIGHTW->port));

        //goes straight if the wheels are caught up with one another.
        if(left_mm == right_mm)
        {
            leftCo = 1.0;
            rightCo = 1.0;
        }
        //corrects if one wheel falls behind the other.
        else if(left_mm < right_mm)
        {
            rightCo = rightCo - increment;
            leftCo = 1.0;
        }
        else if(left_mm > right_mm)
        {
            rightCo = 1.0;
            leftCo = leftCo - increment;
        }

        //powering or freezing motors
        if(left_mm < abs(millimeters))
            mav(driveTrain.LEFTW->port, velocity*leftCo);

        if(left_mm > abs(millimeters))
            freeze(driveTrain.LEFTW->port);

        if(right_mm < abs(millimeters))
            mav(driveTrain.RIGHTW->port, velocity*rightCo);

        if(right_mm > abs(millimeters))
            freeze(driveTrain.RIGHTW->port);

        msleep(1);
    }
    off(driveTrain.LEFTW->port);
    off(driveTrain.RIGHTW->port);
}
```

The cool thing about this code[1] is that robot actually decides what the drift coefficients are as it runs. Using the “get_motor_position_counter(#port)”, (and assuming both wheels have the same amount of ticks per revolution) the robot corrects the drift coefficient of one wheel by a small increment if it gets ahead of the other. This means that the robot drives straight EVERYTIME, and not only that, it also corrects itself whenever one of those infamous motor jerks occurs that every programmer hates.

3 The Importance of Converting into Common Units for Each Wheel

It is extremely important to convert both wheels' ticks per revolution to mm per revolution, so

---

[1] If you are confused by the driveTrain.LEFTW->port call, this is just a bi-product of the library I was programming this code into. Just think of it as the port number for the left wheel (same applies to driveTrain.RIGHTW->port).

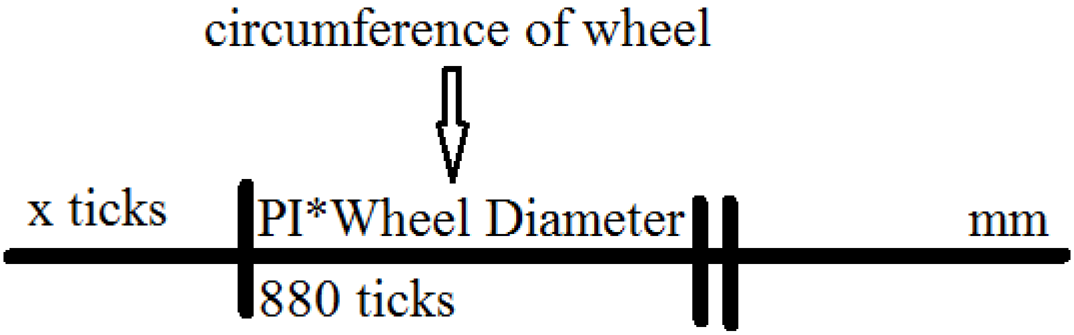
that a wheel with more or less ticks in its cycle than the other is still tracking in the same units. I do this behind the scenes with the call to “motor_convert_ticks_mm(motor motorName, int num_ticks_moved)”.

3.1 A Note On ticks_per_revolution

KIPR programming guides say that there is an average of 1100 ticks in one revolution of a motor. It depends on the wheel, but I have found that for the motors in NAR's possession, the average rests somewhere around 880 ticks per revolution. You can check this number by simply going to the console's motor testing screen. At the top right hand corner of kovan's there should appear a number. As you turn the wheel, the number will increase or decrease. This number is the number of ticks the wheel has moved. Turn the wheel 360 degrees (provided you start with the tick number at 0), and you should have the ticks per one revolution. This is far simpler than running the robot on a table for a distance and doing some algebra, although arguebly less accurate.

3.2 Doing Conversions to get Each Motor into Millimeters

In order to convert ticks to millimeters one just has to do some simple unit analysis; this one assumes that there are 880 ticks in one cycle for a wheel:



In code, assuming wheel diameter and PI are assigned earlier in the program, it would look something like this,:

int ticks_to_travel = (millimeters_specified_by_function*PI*wheel_diameter) / (880 ticks);

The programmer would make a ticks_to_travel_variable for each wheel, and then replace all instances of ticks to total ticks comparison with millimeters to total millimeters traveled comparison, if they were previously measuring distances in ticks.

4 Conclusion

In conclusion, drive trains can be programmed to drive straight using the get_motor_position_counter(#port) function. In order to work, the function must use unit analysis and track the ticks each wheel has moved in a loop. As a programmer, you have the choice to not spend futile hours adjusting values in a program with no clear solution to the problem of your robot not driving straight, because, after all is done, there needs to be solution to driving straight, and changing a few numbers is no the solution.