Wall Following

ROB 102: Introduction to AI & Programming

Lab Session 3

2021/09/24

Administrative

When batteries are charging, they should be switched to **ON** (–).

When batteries are stored, they should be switched to **OFF** (∘).

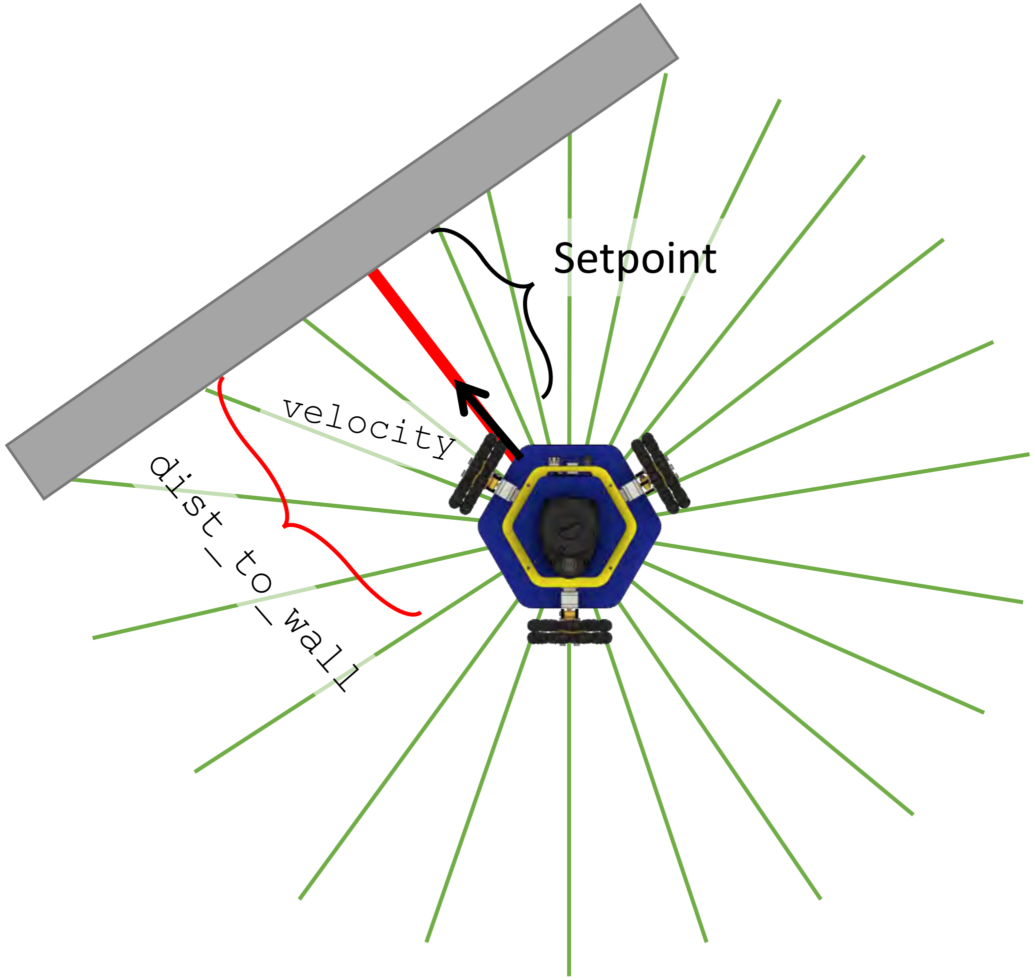
Monday's lecture will be a C++ review (+ Practice Quiz)

Checkpoint: Part 1 (Drive Square) and Part 2 (Safe Drive) should be completed this week.

Today…

1. 2D velocity control
2. Driving parallel to a wall (the cross product)
3. Maintaining a distance from the wall
4. Lab time: work on Project 1

Recall: Bang-Bang Control to Nearest Wall



We can use the same controller (bang-bang or P-control) as last week. But this time, we'll drive in the direction of the shortest ray.

We need to:

1. Find the direction and length of the shortest ray, ✓ Covered on Wednesday
2. Drive the robot in any direction. TODO

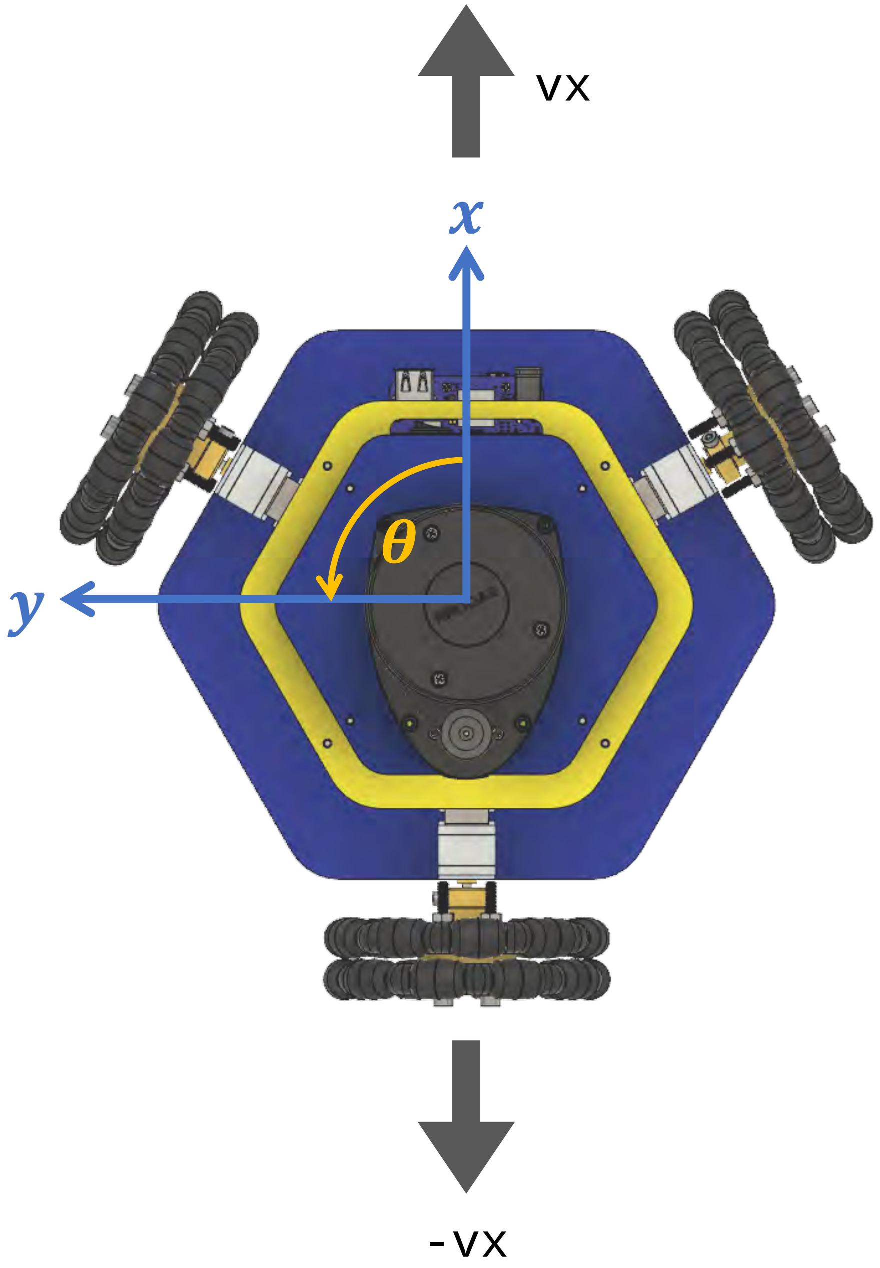
2D Velocity Control

Moving the robot forward:

```
drive(vx, 0, 0);
```

Moving the robot backward:

```
drive(-vx, 0, 0);
```



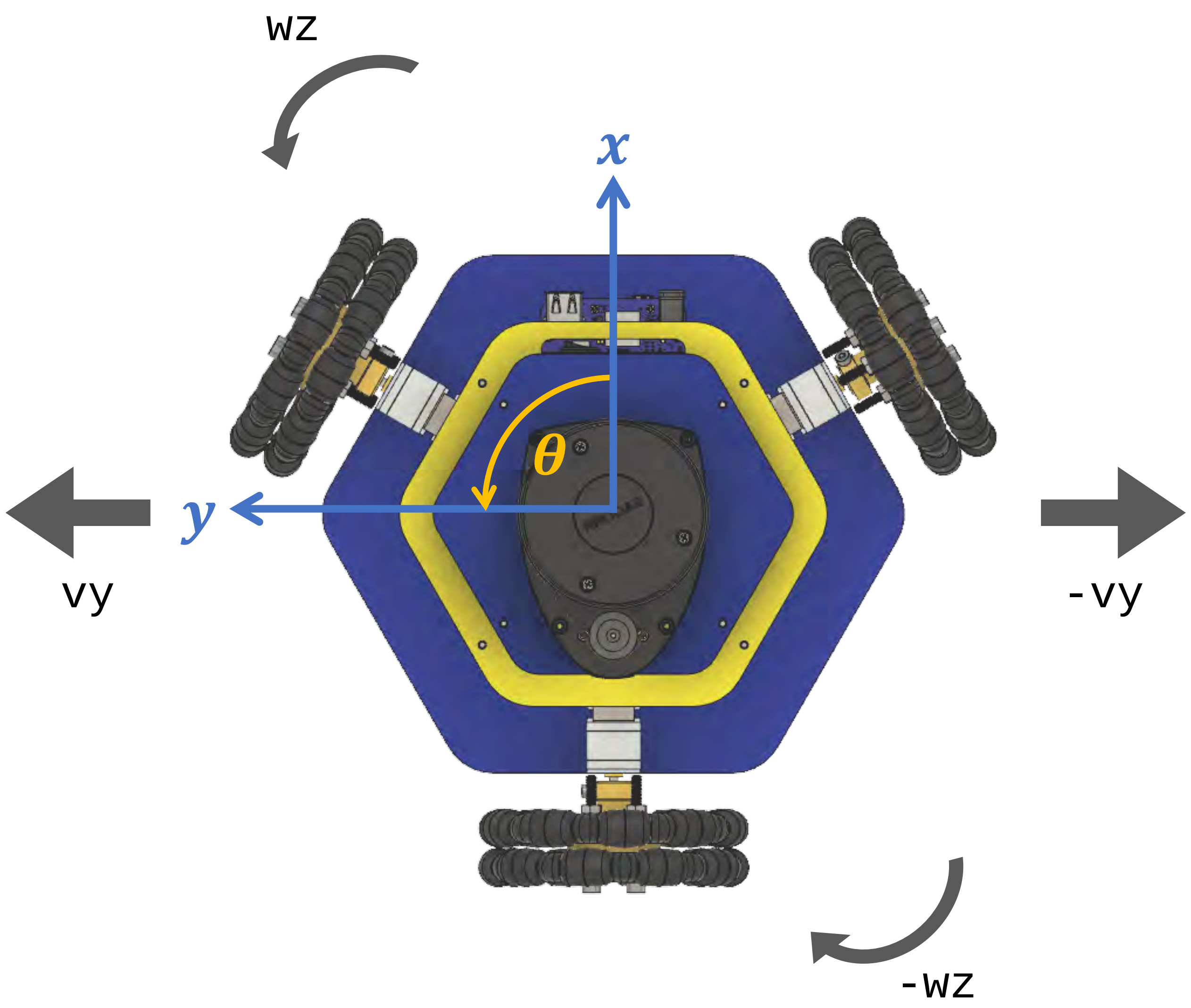
2D Velocity Control

Moving the robot left:

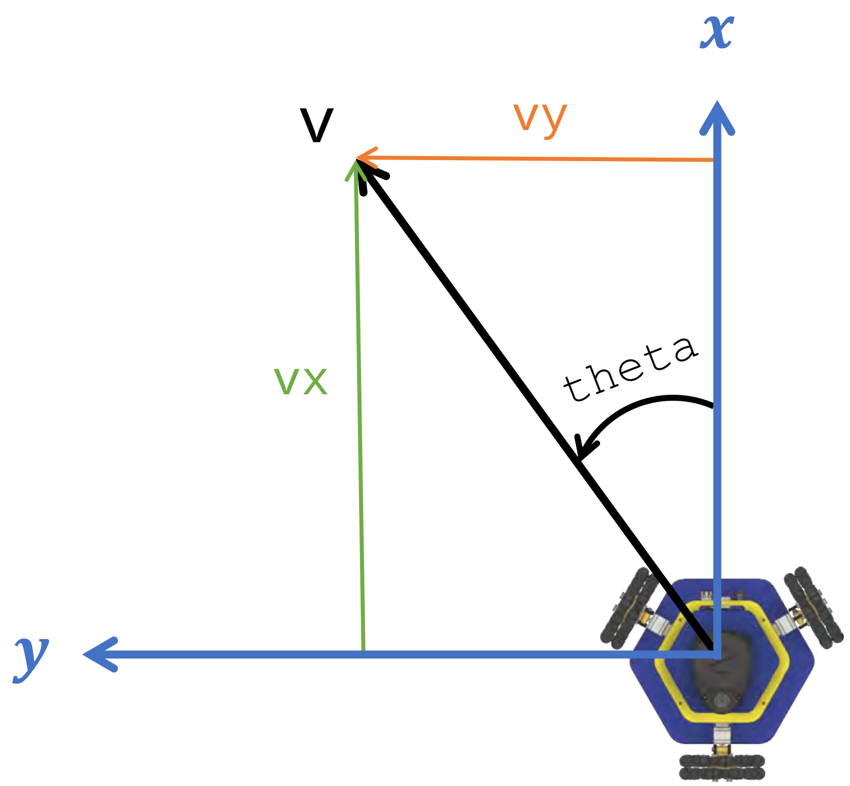
```
drive(0, vy, 0);
```

Rotating counterclockwise:

```
drive(0, 0, wz);
```



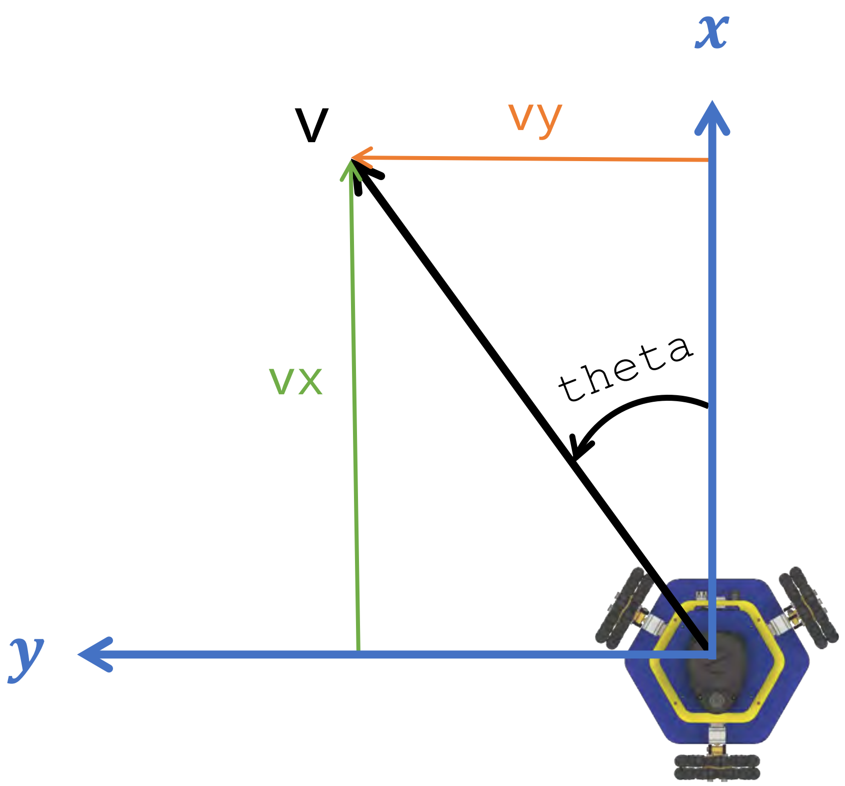
2D Velocity Control: Trigonometry Review



Our velocity is a 2D vector. There are two ways to represent it:

1. Using the magnitude (`v`) and angle (`theta`)
   - Recall from in-class activity: These are polar coordinates.
2. Using the *x* and *y* components of the vector (`vx`, `vy`)

2D Velocity Control: Trigonometry Review



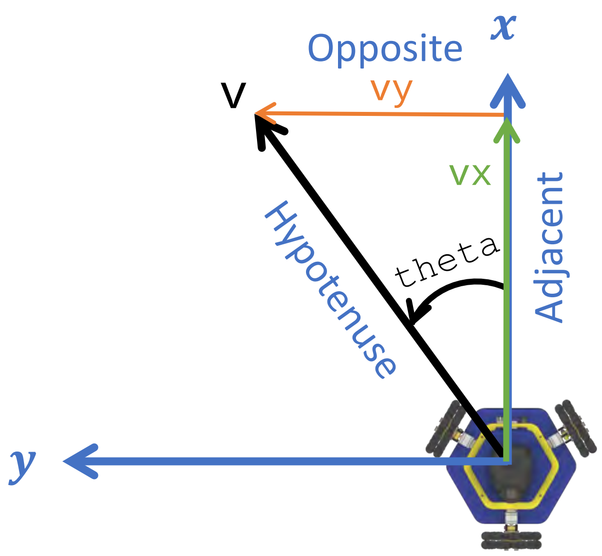
To move at velocity `v`, at angle `theta` (no angular velocity):

```
vx = v * cos(theta)
vy = v * sin(theta)

drive(vx, vy, 0)
```

This will work for any velocity and angle (try it yourself!)

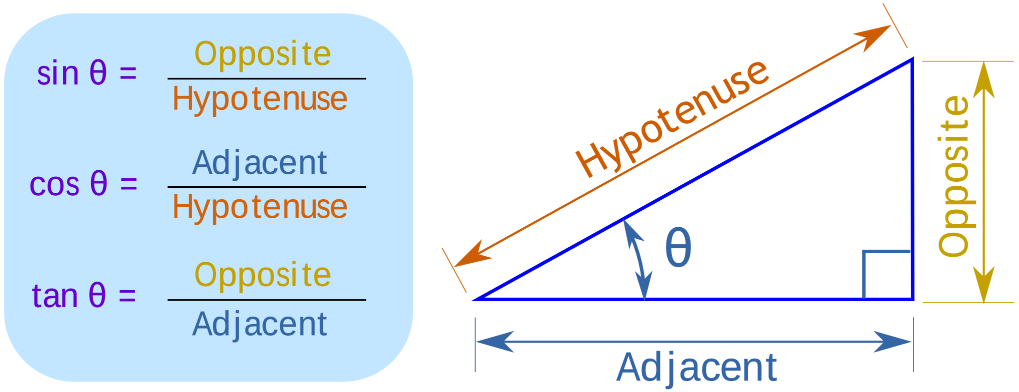
Recall: Trigonometry



To move at velocity v, at angle theta (no angular velocity):

```
vx = v * cos(theta)
vy = v * sin(theta)
```

```
drive(vx, vy, 0)
```



This will work for any velocity and angle (try it yourself!)

The <cmath> library

main.cpp

```cpp
#include <iostream>
#include <cmath>

int main() {
  // Common math expressions
  std::cout << "pow(3, 3) = " << pow(3, 3) << "\n";
  std::cout << "sqrt(2) = " << sqrt(2) << "\n";
  std::cout << "abs(-3) = " << abs(-3) << "\n";
  std::cout << "fabs(-1.5) = " << fabs(-1.5) << "\n";
  std::cout << "Careful! abs(-1.5) = " << abs(-1.5) << "\n";

  // Trig functions
  float pi = 3.14159265359;
  std::cout << "\nsin(1.0) = " << sin(1.0) << "\n";
  std::cout << "tan(pi) = " << tan(pi) << "\n";

  // Logs & Exponentials
  float e = 2.71828;
  std::cout << "\nlog(e) = " << log(e) << "\n";
  std::cout << "exp(1.0) = " << exp(1.0) << "\n";
}
```

Remember to include the library!

Contains common math operations.

Console Shell

```
~/Test$ g++ main.cpp -o main
~/Test$ ./main
pow(3, 3) = 27
sqrt(2) = 1.41421
abs(-3) = 3
fabs(-1.5) = 1.5
Careful! abs(-1.5) = 1

sin(1.0) = 0.841471
tan(pi) = 8.74228e-08

log(e) = 0.999999
exp(1.0) = 2.71828
~/Test$
```

3^3

$\sqrt{2}$

$|-1.5|$

$e^{1.0}$

The `<cmath>` library

For a list of all the functions:

https://www.cplusplus.com/reference/cmath/

This website is a great reference for all things C++!

fx Functions

Trigonometric functions

cos	Compute cosine (function)
sin	Compute sine (function)
tan	Compute tangent (function)
acos	Compute arc cosine (function)
asin	Compute arc sine (function)
atan	Compute arc tangent (function)
atan2	Compute arc tangent with two parameters (function)

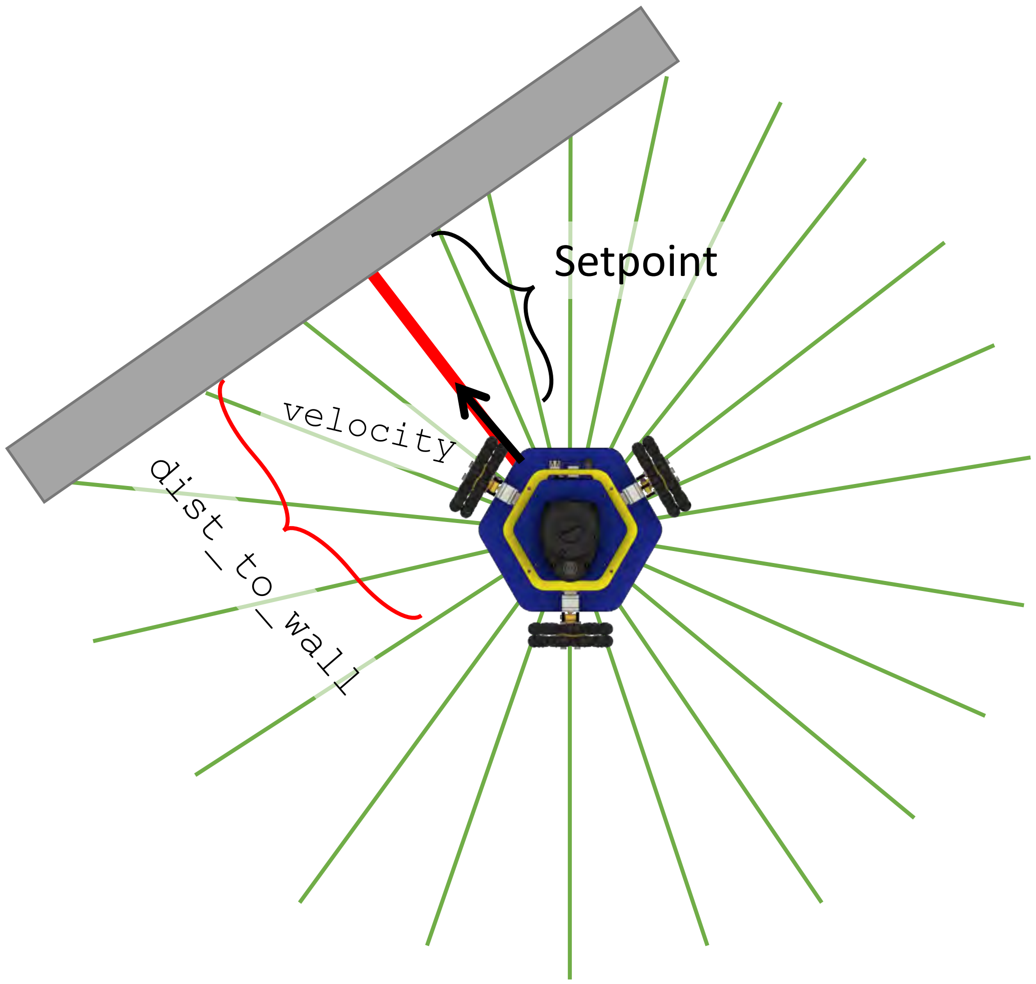
Hyperbolic functions

cosh	Compute hyperbolic cosine (function)
sinh	Compute hyperbolic sine (function)
tanh	Compute hyperbolic tangent (function)
acosh C++11	Compute area hyperbolic cosine (function)
asinh C++11	Compute area hyperbolic sine (function)
atanh C++11	Compute area hyperbolic tangent (function)

Exponential and logarithmic functions

exp	Compute exponential function (function)
frexp	Get significand and exponent (function)
ldexp	Generate value from significand and exponent (function)
log	Compute natural logarithm (function)
log10	Compute common logarithm (function)
modf	Break into fractional and integral parts (function)
exp2 C++11	Compute binary exponential function (function)
expm1 C++11	Compute exponential minus one (function)
ilogb C++11	Integer binary logarithm (function)
log1p C++11	Compute logarithm plus one (function)
log2 C++11	Compute binary logarithm (function)
logb C++11	Compute floating-point base logarithm (function)
scalbn C++11	Scale significand using floating-point base exponent (function)
scalbln C++11	Scale significand using floating-point base exponent (long) (function)

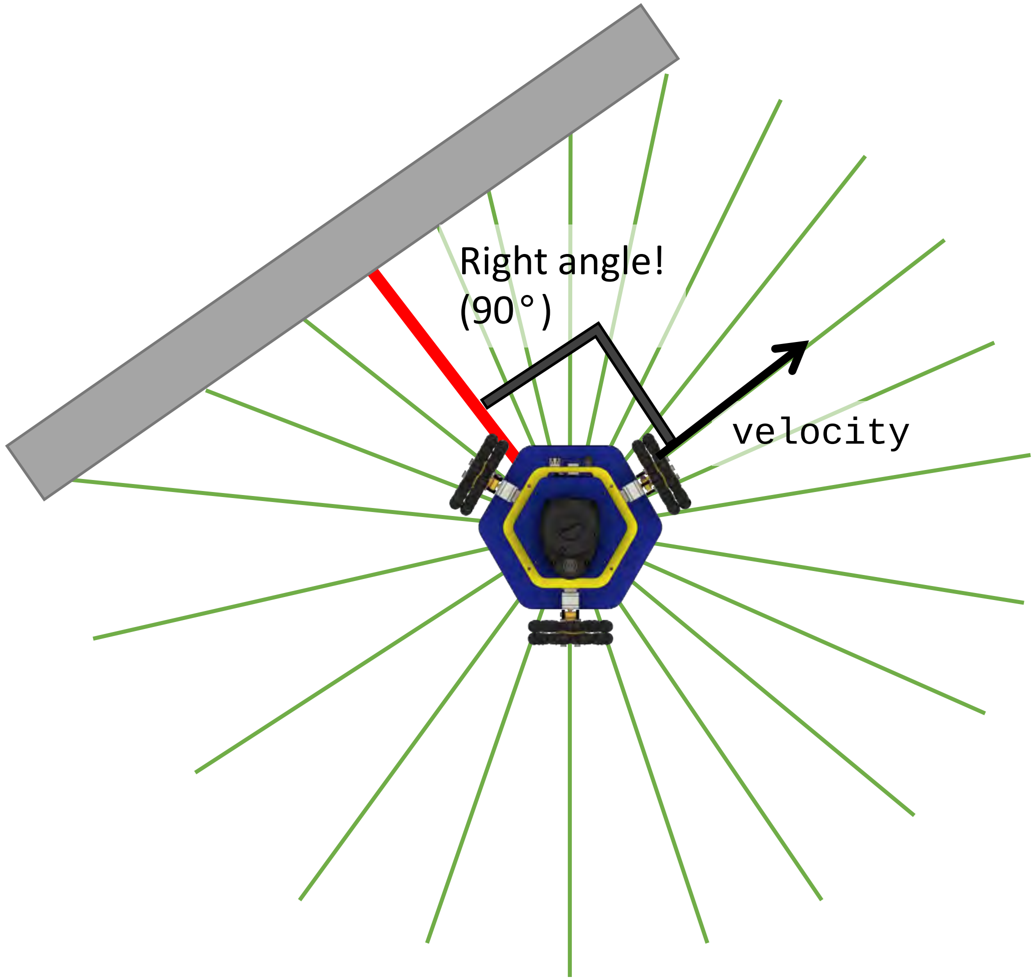
Wall Following



In class, we discussed how we can maintain a distance to the wall by driving in the direction of the shortest ray.

How can we drive along the wall?

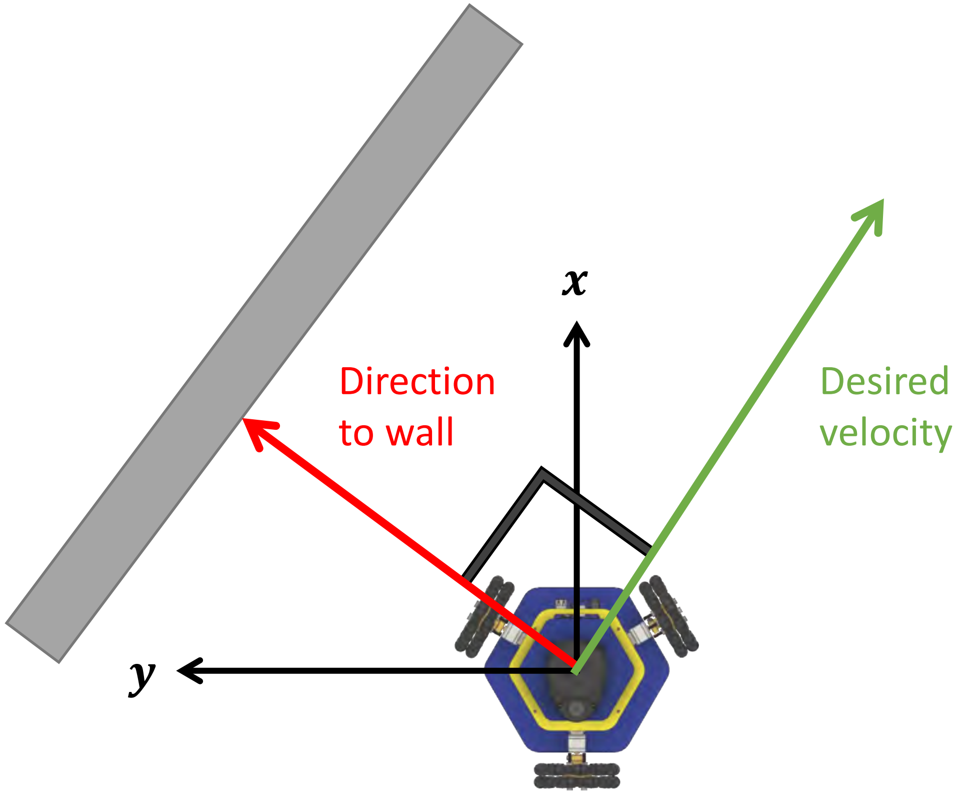
Wall Following



We can pick a velocity *perpendicular* (at a right angle) to the shortest ray.

How do we calculate this velocity vector?

Shortest Lidar Ray



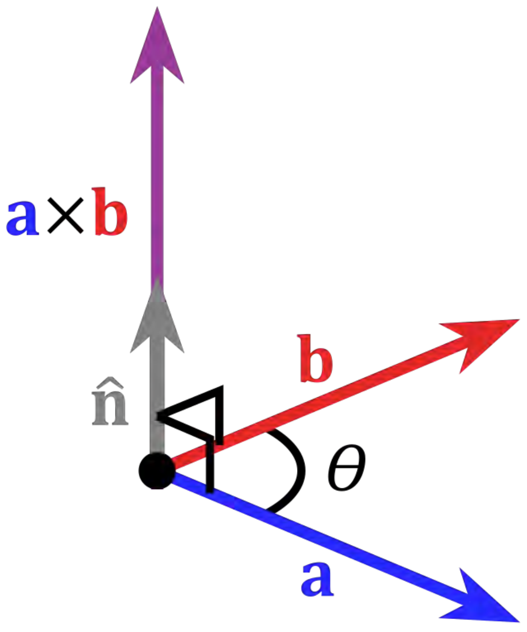
We know the magnitude and the angle of the vector pointing to the wall:

```
// Get the distance to the wall.
float min_idx = findMinDist(scan);
float dist_to_wall = scan.ranges[min_idx];
float angle_to_wall = scan.thetas[min_idx];
```

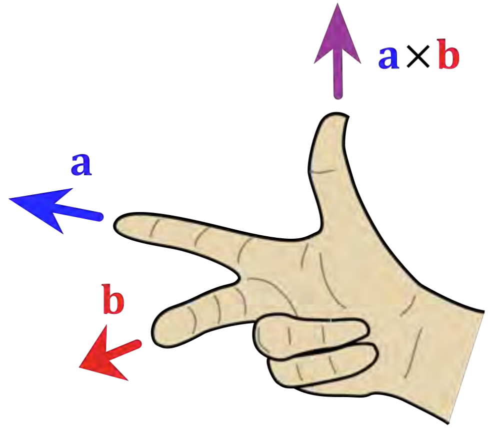
Given the index of the minimum length ray, the magnitude is `dist_to_wall` and the angle is `angle_to_wall`.

The Cross Product

The **cross product** is an operation that finds a 3D vector *perpendicular* to two other 3D vectors.



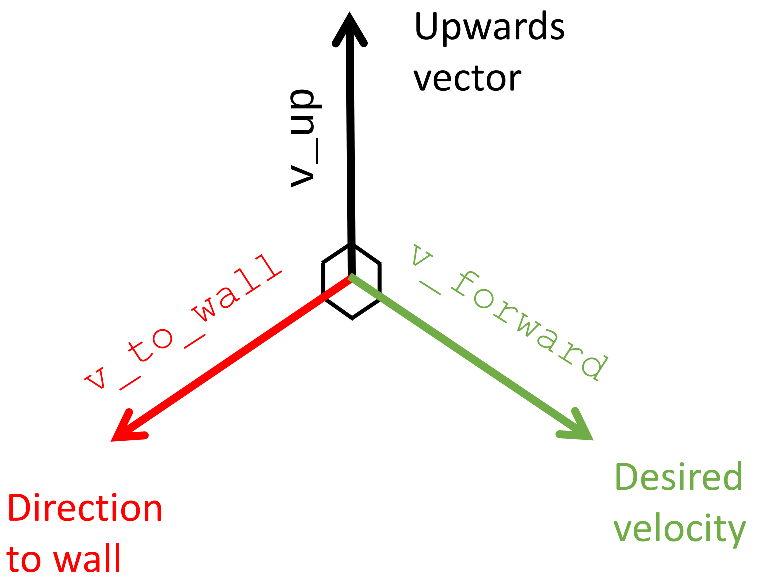
The Right-Hand Rule gives the direction of the resulting vector.



Note: Our robot can only move in 2D, but the velocity vectors can be written as 3D vectors by setting the *z* component to zero.

The Cross Product

The **cross product** is an operation that finds a 3D vector *perpendicular* to two other 3D vectors.



We pick a vector pointing up as the second vector:

```
v_up = [0, 0, 1]  // x,y,z
```

It is guaranteed to be perpendicular to the desired velocity.

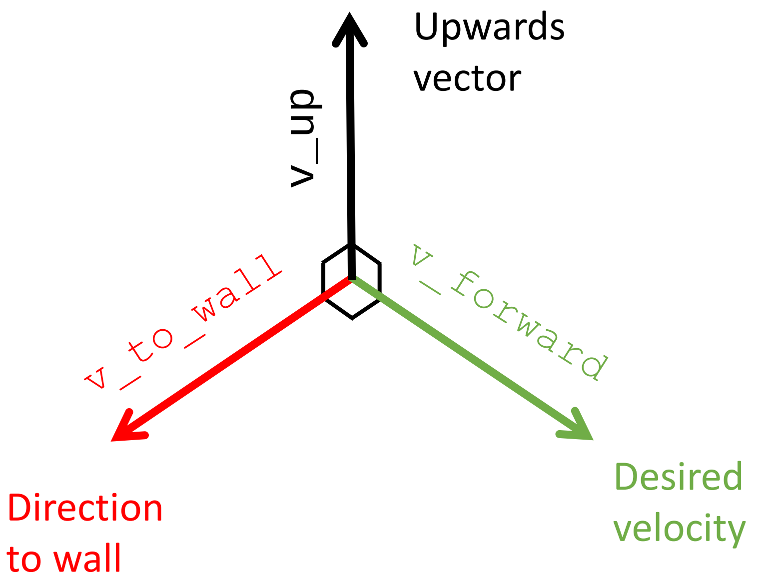
The Cross Product

We pick a vector pointing up as the second vector:

```
v_up = [0, 0, 1]  // x,y,z
```

Now we can use the cross product to find the forward velocity vector:

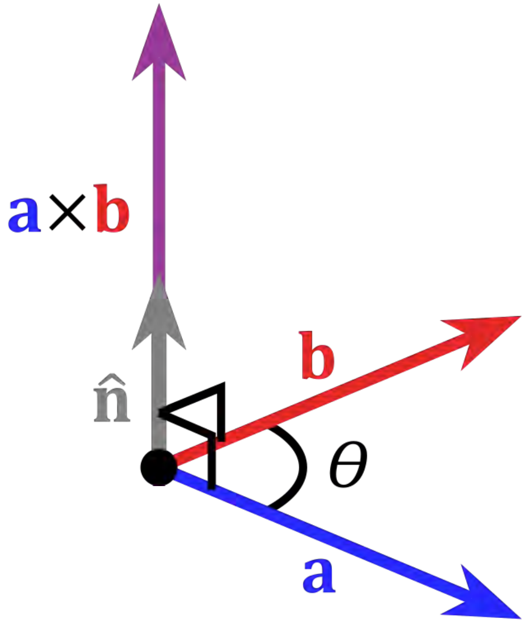
```
v_forward = crossProduct(v_to_wall, v_up)
```



Computing the Cross Product

Given two vectors $\boldsymbol{a} = (a_x, a_y, a_z)$ and $\boldsymbol{b} = (b_x, b_y, b_z)$, their cross product $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$ can be computed like this:

$$
\begin{aligned}
c_x &= a_y b_z - a_z b_y \\
c_y &= a_z b_x - a_x b_z \\
c_z &= a_x b_y - a_y b_x
\end{aligned}
$$

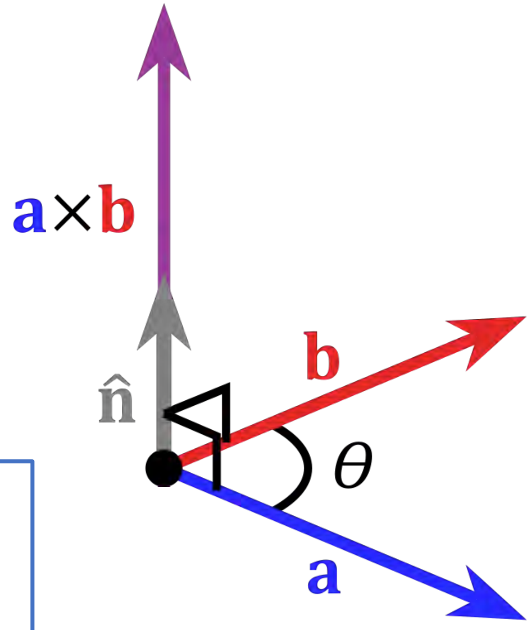


The *magnitude* of $\boldsymbol{c}$ will be the area of the parallelogram made by $\boldsymbol{a}$ and $\boldsymbol{b}$.

Computing the Cross Product

Computing $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$:

$$
\begin{aligned}
c_x &= a_y b_z - a_z b_y \\
c_y &= a_z b_x - a_x b_z \\
c_z &= a_x b_y - a_y b_x
\end{aligned}
$$




```cpp
Vector3D crossProduct(const Vector3D& v1, const Vector3D& v2)
{
    Vector3D res;
    /**
     * TODO: (P1.3.1) Take the cross product between v1 and v2 and store the
     * result in res.
     **/
    return res;
}
```

Your turn! Compute the cross product in this function.

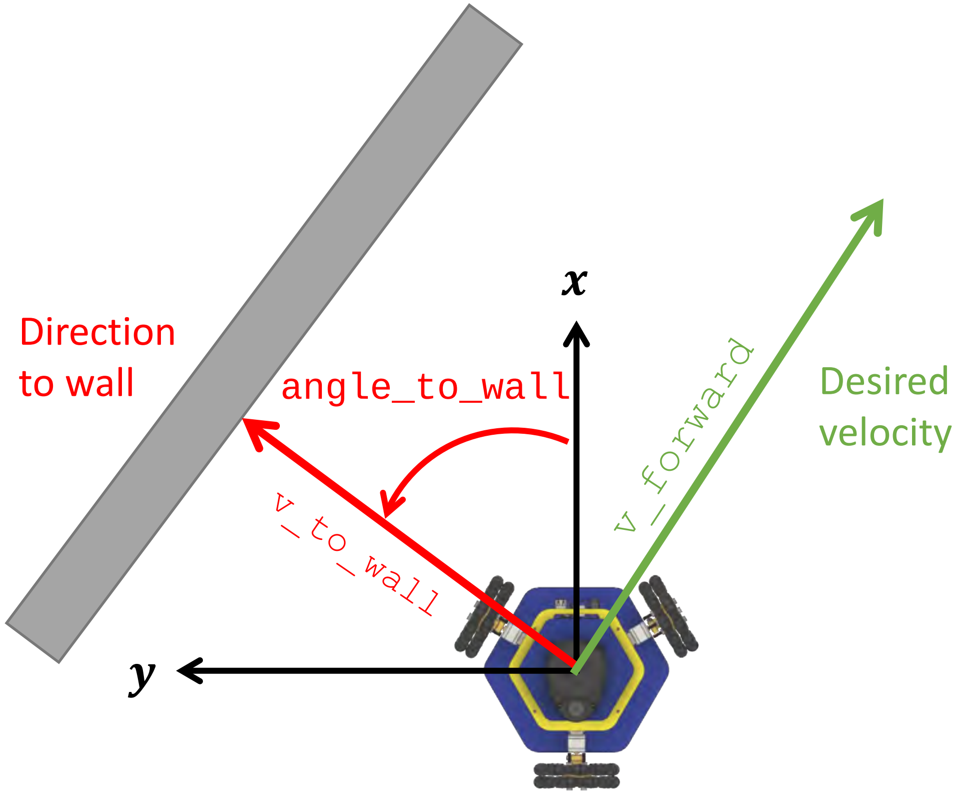
src/common/utils.cpp

```cpp
struct Vector3D
{
    Vector3D() :
        x(0),
        y(0),
        z(0)
    {};

    float x, y, z;
};
```

include/wall_follower/common/utils.h

Computing the Cross Product



We know the magnitude and the angle of the vector pointing to the wall:

```
// Get the distance to the wall.
float min_idx = findMinDist(scan);
float dist_to_wall = scan.ranges[min_idx];
float angle_to_wall = scan.thetas[min_idx];
```

What is v_to_wall?

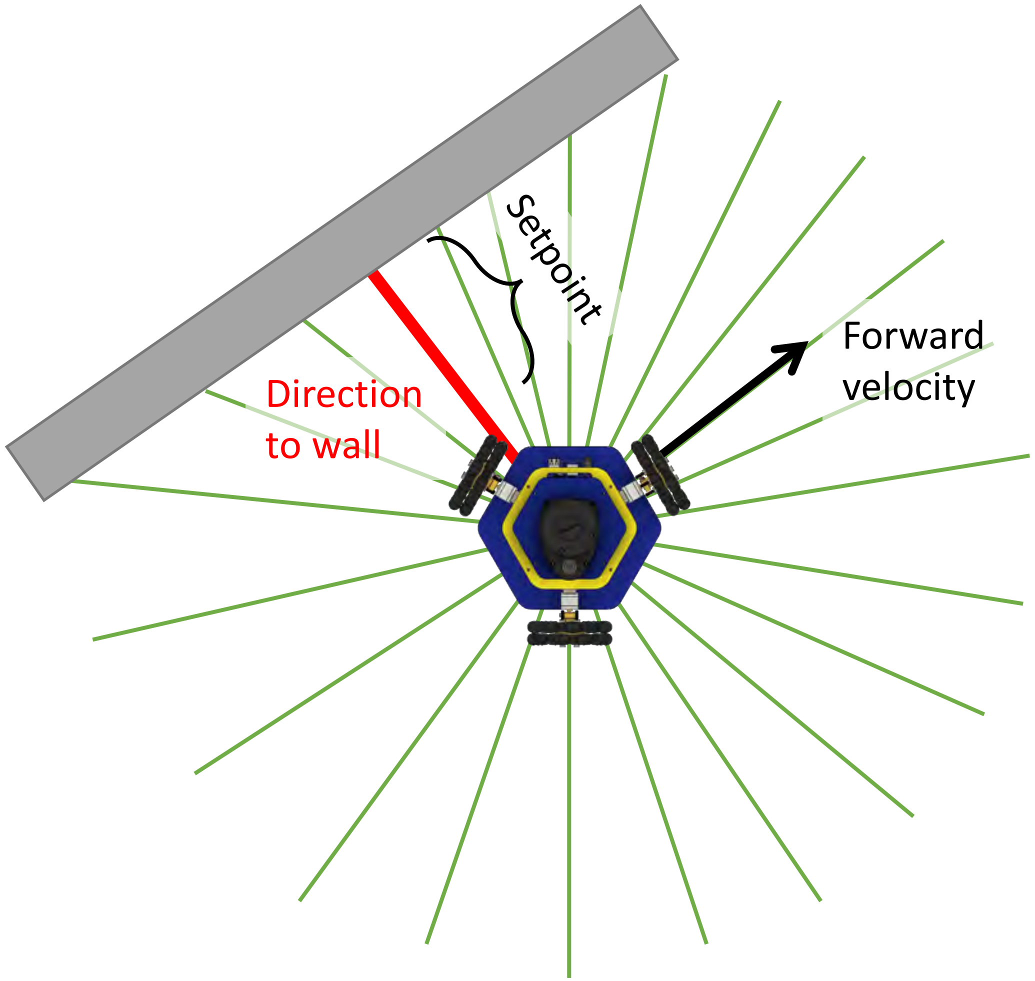
x

```
v_to_wall = [1*cos(angle_to_wall),
             1*sin(angle_to_wall), 0]
```

y z

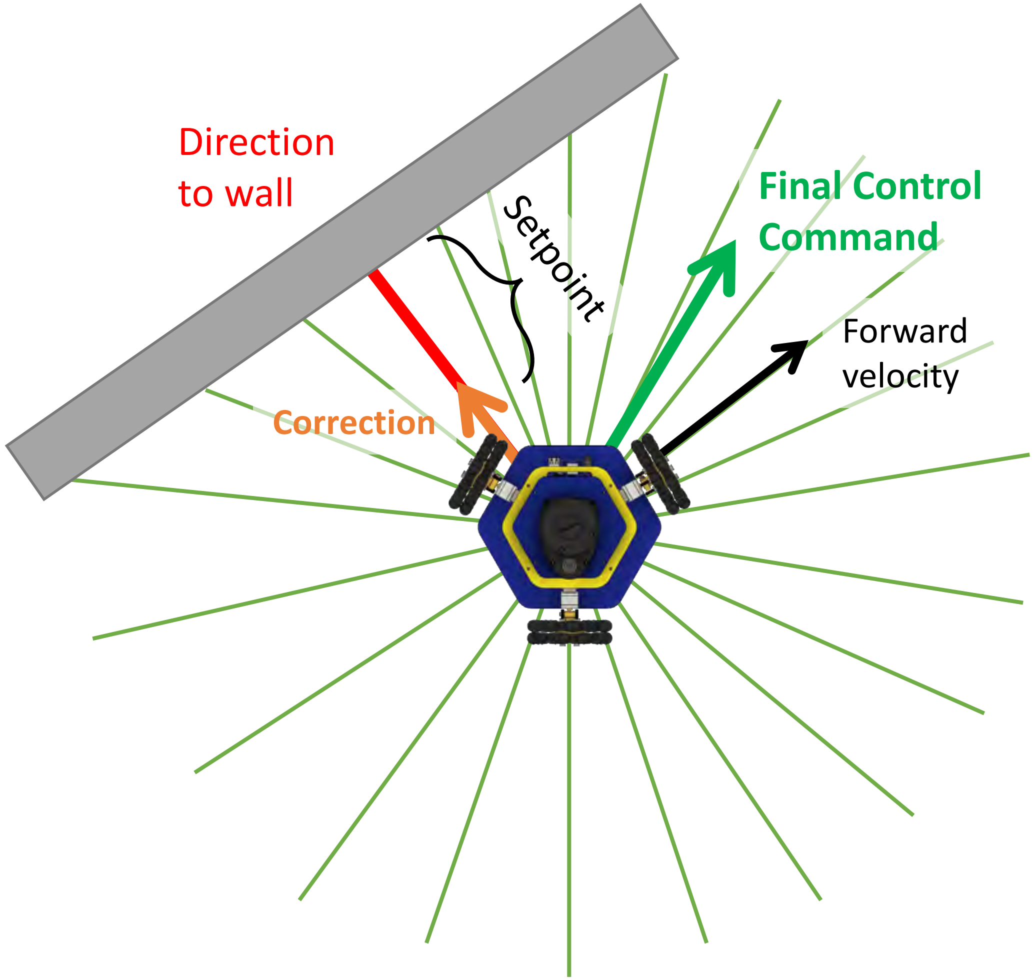
If we set the magnitude of v_to_wall to 1, v_forward will have magnitude 1. We can multiply by a chosen drive velocity before sending the control commands to the robot.

Wall Following: What are we missing?



The robot should maintain a setpoint distance from the wall!

Wall Following: Bang-Bang Control

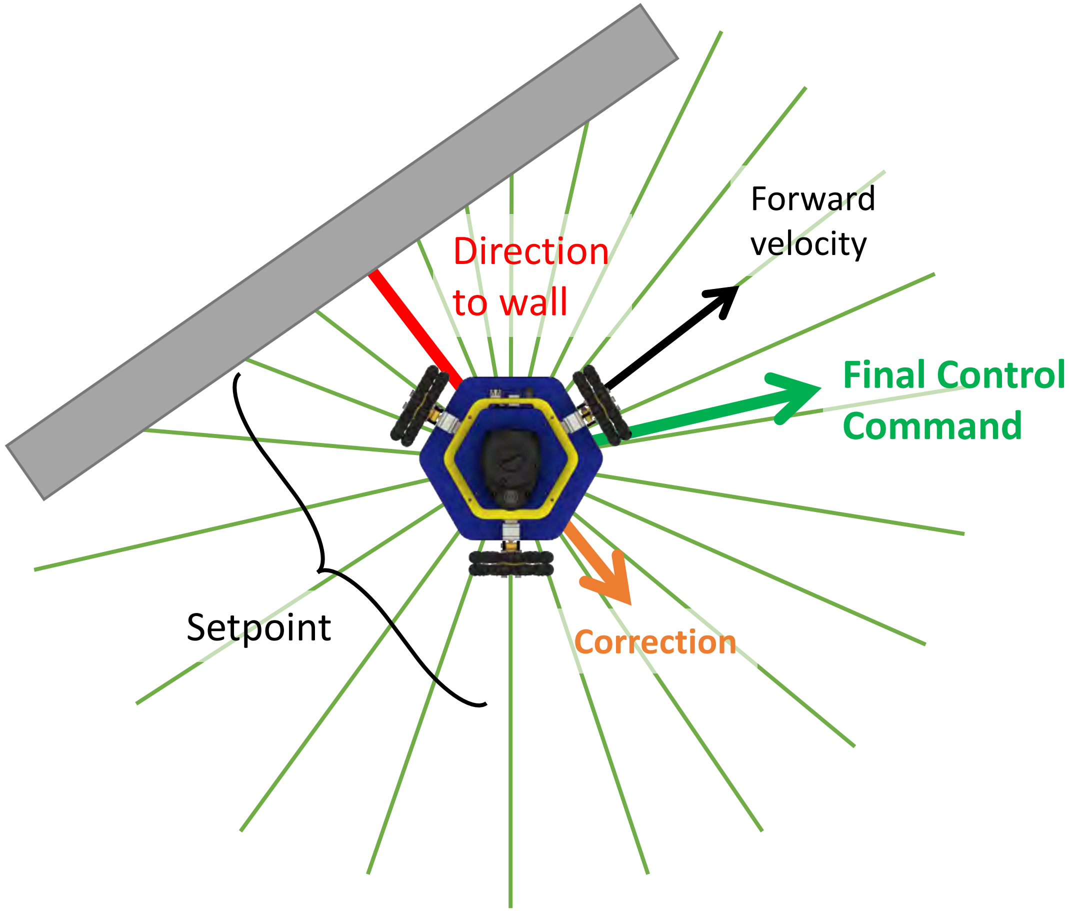


We can add a correction to the forward velocity to drive the robot closer or farther from the wall.

Too far from wall! Move closer.

The final control command is the forward velocity plus the correction (a vector addition!).

Wall Following: Bang-Bang Control



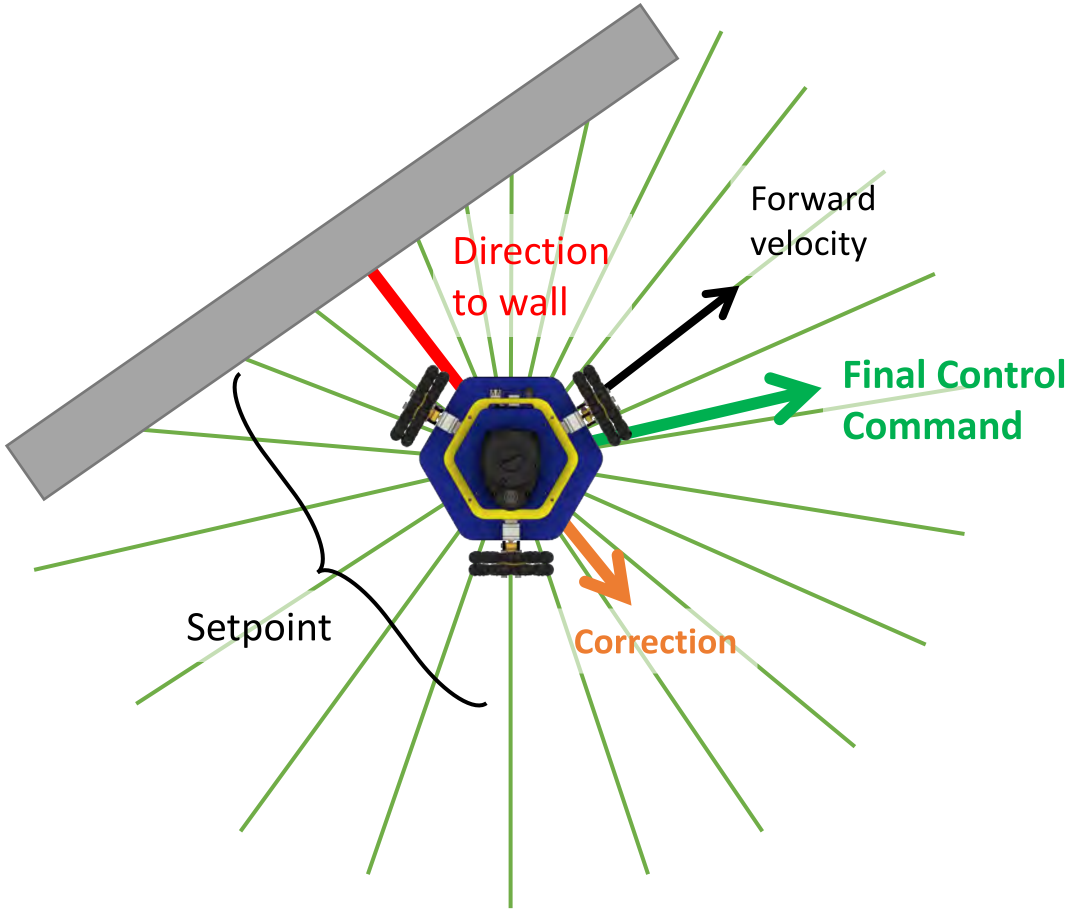
We can add a correction to the forward velocity to drive the robot closer or farther from the wall.

Too close to wall! Move farther.

For **Bang-Bang control**, the magnitude of the correction is *fixed*.

For **P-control**, the magnitude of the correction is *proportional to the error*.

Wall Following: Algorithm



Loop forever:

1. Read a scan from the lidar
2. Find the shortest ray
3. Compute a vector pointing to the wall
4. Take a cross product to find the forward velocity
5. Compute the correction vector
6. Compute the final control command
7. Send the control command to the robot

TODO: Today

1. Get `findMinDist()` function working
   - Used in both the 2D control in-class activity and P1.2
2. [*Optional*] Finish 2D control activity from Wednesday
3. Work on Project 1
   1. P1.1 (Drive Square) should be finished and pushed to GitHub
   2. P1.2 (Drive Safe) can be finished once `findMinDist()` is working
   3. For P1.3 (Wall Following), start with the cross product and driving parallel to the wall. Then add the correction.