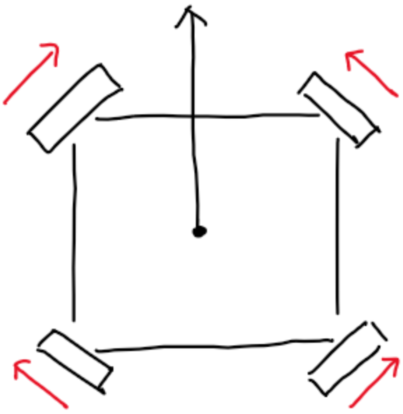
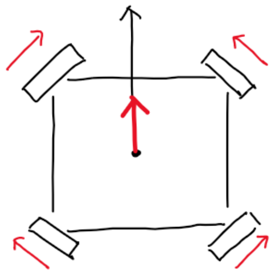
The content below is part of a VEXU team's notebook draft for the 2020 - 2021 season
To contact the former competitor that wrote this:
vexforum @sarah_97963A, discord @Sarah 97963A#2509

Move To Point Function (Holonomic)

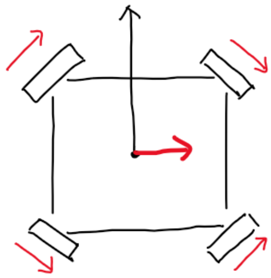


For the X drive we use, we defined the forward direction of the motor to be the direction that brings the robot in the positive Y direction. In that case, certain motors are reversed so that the red arrows in the picture on the left indicate the fwd direction of the motors, when the robot has the heading indicated by the black arrow.

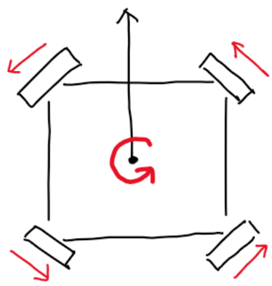
When the robot moves in the positive Y direction in its own frame, all four wheels spin in the forward direction.



When the robot moves in the positive X direction in its own frame, the left front wheel and the right back wheel spin in the forward direction, while the left back wheel and the right front wheel spin in the reverse direction.



When the robot spins in the direction of increasing heading, the right front and back wheels spin in the forward direction, while the left front and back wheels spin in the reverse direction.

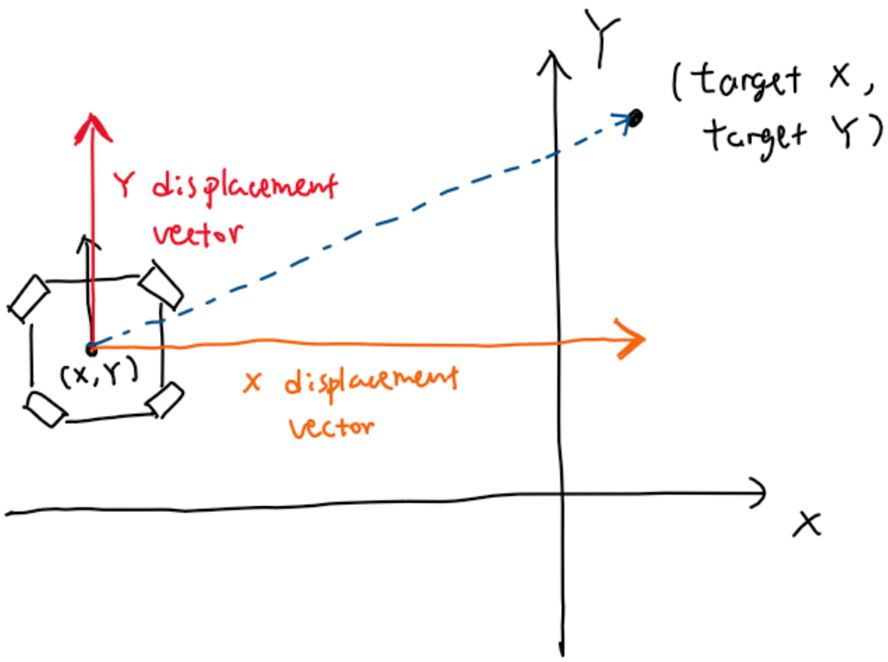


Therefore, if we want the robot to move in *its own frame* with `XPower` in the positive X direction, `YPower` in the positive Y direction, and `turnPower` in the positive heading direction, the combined power for each motor would be the following:

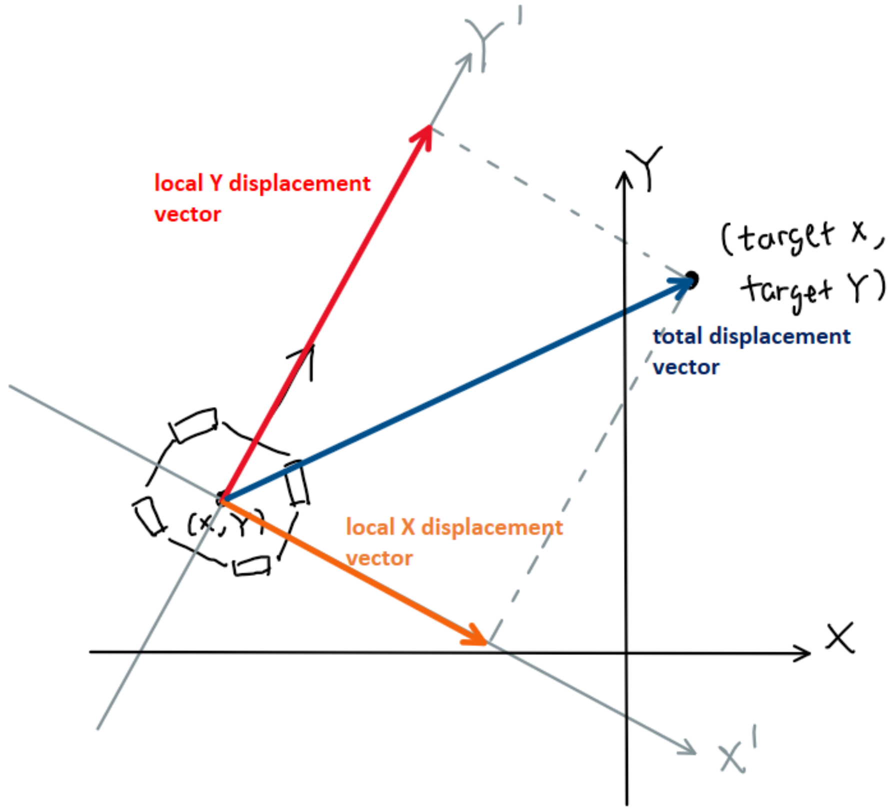
Motor\Action	Positive Y direction	Positive X direction	Positive heading direction	Combined power
Left front	+ YPower	+ XPower	- turnPower	YPower + XPower - turnPower
Left back	+ YPower	- XPower	- turnPower	YPower - XPower - turnPower
Right front	+ YPower	- XPower	+ turnPower	YPower - XPower + turnPower
Right back	+ YPower	+ XPower	+ turnPower	YPower + XPower + turnPower

Table 1

When the robot is facing forward and we want it to move from its initial position (X, Y) to target position (target X, target Y), it's appropriate to break down the total displacement vector into a Y displacement vector and an X displacement vector as shown in the figure below. Then, we can define an X direction error and a Y direction error, and calculate the `XPower` and the `YPower` using control loops such as PID.

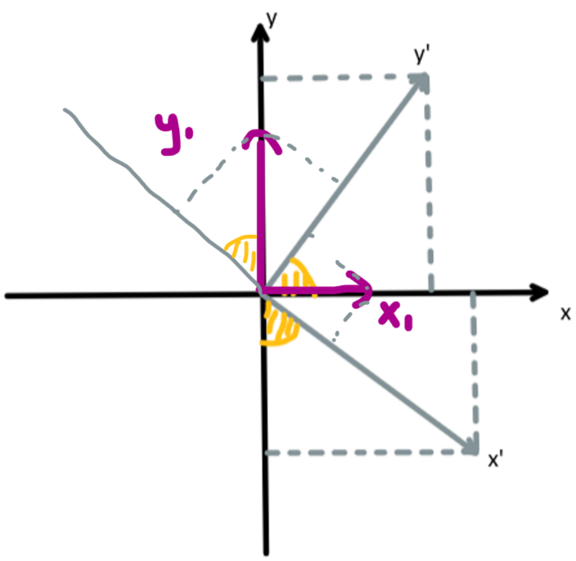


However, the method described above is only valid when the robot is facing forward. That is, when the robot's local frame is aligned with the global frame. When the robot is facing other directions, the X displacement vector and the Y displacement vector, which are calculated in the global coordinate system, will not stay the same when translated into the robot's local coordinate system.



As shown above, instead of finding the total displacement vector's projection onto the global X and Y axis, we should find its projection onto the local X and Y axis (denoted as X' and Y').

The math of finding the local X and Y axis projection is very similar to what we have done in the odometry section, when converting from local coordinate system into global coordinate system. What we have done previously is conversion from local to global, what we need now is conversion from global to local.



In the figure on the left, black axes X and Y represent the global system, gray axes X' and Y' represent the local coordinate system.

If we know the global X and Y displacement vector, x1 and y1, and the angle between the local Y axis and the global X axis, which is the robot's heading, the local X and Y displacement vectors (x1' and y1') can be calculated in the following way:

x1' = x1*sin(theta) + y1*cos(pi - theta)
y1' = y1*sin(pi - theta) + x1*cos(theta)

Again, the `XPower` and `YPower` can be calculated using any control loops and the local displacement vectors.

Now that we have a way to move the robot facing any direction from any position to a certain target position, we can make the movement more efficient by making the robot turn toward a target heading during its movement toward the target point. The calculation of the heading error is fairly easy, we can simply define `turnError = targetHeading - currentHeading`. The only small issue is that since we have limited the heading range to [0, 360), the heading error calculated by the above equation can be unreasonable sometimes. For example, the robot current has a heading of 45 degrees and needs to turn to 315 degrees, the equation `turnError = targetHeading - currentHeading` would give a heading error of 270 degrees, which means the robot should turn counterclockwise for 270 degrees. This is not very efficient since turning 90 degrees clockwise would achieve the same thing.

Therefore, we created a function to calculate the minimum angle the robot has to turn to arrive at the target heading:

```
double find_min_angle (double firstAngle, double secondAngle){
 double minimunAngle = firstAngle - secondAngle;
 if(minimunAngle > 180){
   minimunAngle = firstAngle - 360 - secondAngle;
 }

 if(minimunAngle < -180){
   minimunAngle = firstAngle + (360 - secondAngle);
 }
 return minimunAngle;
}
```

After `turnError` is calculated, control loops can be applied to it to calculate the robot's turn speed.

To summarize the above calculations and derivations, the Move to Point Function we came up with consists of the following steps:

1. Calculate the global X and Y displacement vectors
2. Convert them into the robot's local coordinate system (its own frame)
3. Calculate turn error

4. Apply control laws to the X and Y displacement vectors and the turn error to obtain XPower, YPower, and turnPower
5. Calculate the power of each motor by combining the three powers correctly

Our code (can be simplified but I'm lazy):

```
void move_to_point (double targetX, double targetY, double preferredAngle, double linearMax, double
turnMax){

 Brain.Screen.clearScreen();

 // calculate absolute angle, which is the where the displacement vector is pointing in global
coordinate system
 // in radian
 double absoluteTargetAngle = atan2((targetY - currentY), (targetX - currentX)) * 180 / M_PI;
 // make sure it goes from 0-360 to match the global coordinate
 if (absoluteTargetAngle < 0){
   absoluteTargetAngle += 360;
 }

 double distanceToTarget = sqrt(pow((targetY - currentY), 2) + pow((targetX - currentX), 2));
 // for pid
 double last_distanceToTarget = distanceToTarget;

 double displacementXComponent = targetX - currentX;
 double displacementYComponent = targetY - currentY;

 // calculate the angle the robot needs to turn
 // might be different from relative angle to target depending on which type of motion you want
 // -1 --> always line up with target; other values --> turn to that specific value
 double turnAngle = 0;
 if (preferredAngle == -1){
   //if want the robot to always line up with target point
   turnAngle = find_min_angle(absoluteTargetAngle, currentHeading);
 }
 else{
   //if want the robot to stay facing the assigned direction during movements
   turnAngle = find_min_angle(preferredAngle, currentHeading);
 }
 // for pid
 double last_turnAngle = turnAngle;

 while((fabs(displacementXComponent) > 0.02) || (fabs(displacementYComponent) > 0.02) ||
(fabs(turnAngle) > 0.5)){

   // update the absolute angle
   absoluteTargetAngle = atan2((targetY - currentY), (targetX - currentX)) * 180 / M_PI;
   if (absoluteTargetAngle < 0){
     absoluteTargetAngle += 360;
   }

   // update angle to turn
   if (preferredAngle == -1){
     //if want the robot to always turn toward target point
     turnAngle = find_min_angle(absoluteTargetAngle, currentHeading);
   }
   else{
     //if want the robot to stay facing the assigned direction during movements
     turnAngle = find_min_angle(preferredAngle, currentHeading);
```

```
    }

    // get pid output
    double turnSpeed = pid_turn(turnAngle, last_turnAngle);

    if(turnSpeed > turnMax){
      turnSpeed = turnMax;
    }
    if(turnSpeed < -turnMax){
      turnSpeed = -turnMax;
    }

    // update distance and x y displacement component
    distanceToTarget = sqrt(pow((targetY - currentY), 2) + pow((targetX - currentX), 2));
    // get pid output
    double movementSpeed = pid(distanceToTarget, last_distanceToTarget);
    // limiting speed
    if(movementSpeed > linearMax){
      movementSpeed = linearMax;
    }

    displacementXComponent = targetX - currentX;
    displacementYComponent = targetY - currentY;

    // using the robot's current heading as y axis, set a local coordinate system
    // calculate the x y displacement component (vector) in the robot's local coordinate system
    /*
     * global to local:
     * x' = - y*cos(theta) + x*sin(theta)
     * y' = y*sin(theta) + x*cos(theta)
     */
    double displacementXComponent_local = - displacementYComponent * cos(currentHeading / 180 * M_PI)
+ displacementXComponent * sin(currentHeading / 180 * M_PI);
    double displacementYComponent_local = displacementYComponent * sin(currentHeading / 180 * M_PI) +
displacementXComponent * cos(currentHeading / 180 * M_PI);

    double movementXRatio = displacementXComponent_local / (fabs(displacementXComponent_local) +
fabs(displacementYComponent_local));
    double movementYRatio = displacementYComponent_local / (fabs(displacementXComponent_local) +
fabs(displacementYComponent_local));

    move_with_assigned_power(movementXRatio * movementSpeed, movementYRatio * movementSpeed,
turnSpeed);

    // updates
    last_distanceToTarget = distanceToTarget;
    last_turnAngle = turnAngle;

    task::sleep(10);
  }

  Brain.Screen.clearScreen();
  group_stop_hold();

}
```