Joystick

Thumb Joystick

The joystick as previously discussed in the Manifest in *Introduction Robotics* is the analog 2-axis thumb joystick with a button. It has two analog *10k* potentiometers.

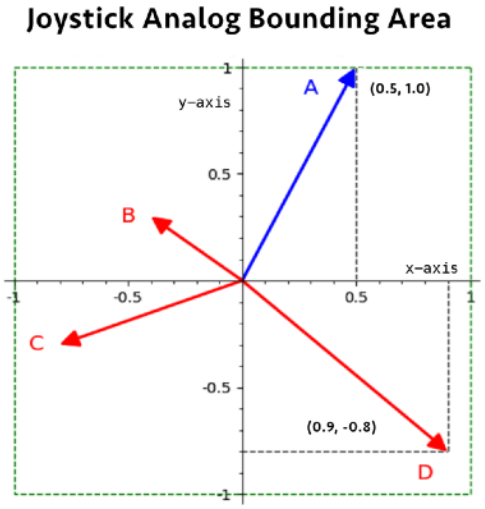


Image 1

Each potentiometer circuit might be connected to a MCU analog pin where the analog signal is converted when programming would output a digital reading as shown as the x-axis and the y-axis. The two axis (x, y) give the joystick analog bounding area shown within the dotted green line in Image 1. As the joystick is moved from the center to the final position, for example the resultant vector would produce the letter A the blue arrow in Image 2. The same vector matches that in Image 1. The x-component in vector A is 0.5 where the joystick moved half the distance along the

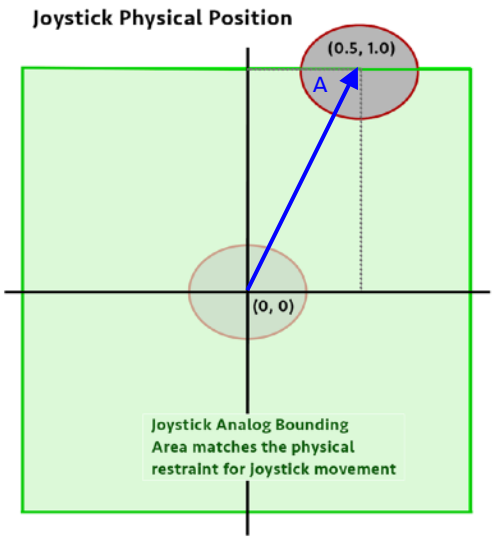
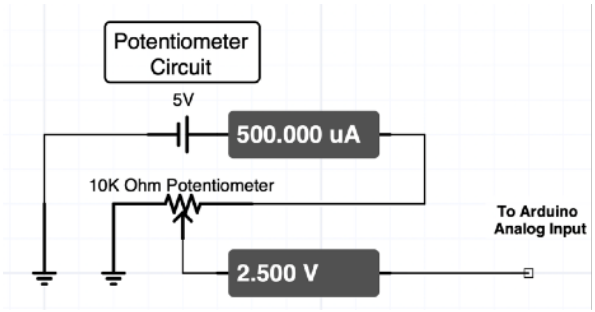


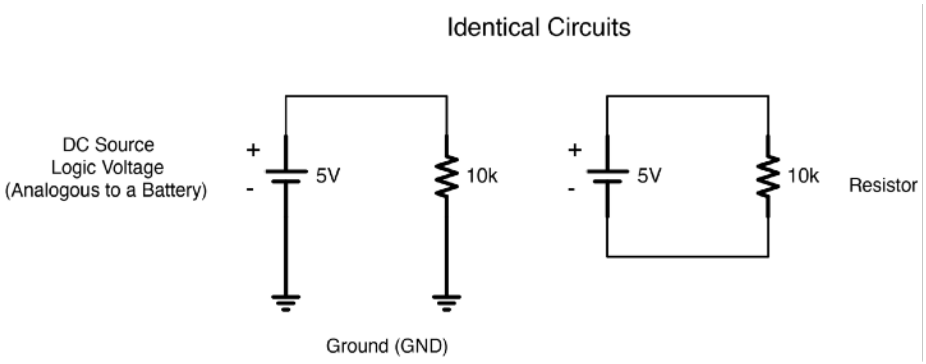
Image 2

x-axis in the positive direction. The y-component is 1.0 where the joystick moved to the end of the bounding distance along the y-axis. Computationally, the vectors illustrate the behavior of the joystick but as briefly mentioned, programming the ADC reading output is required.



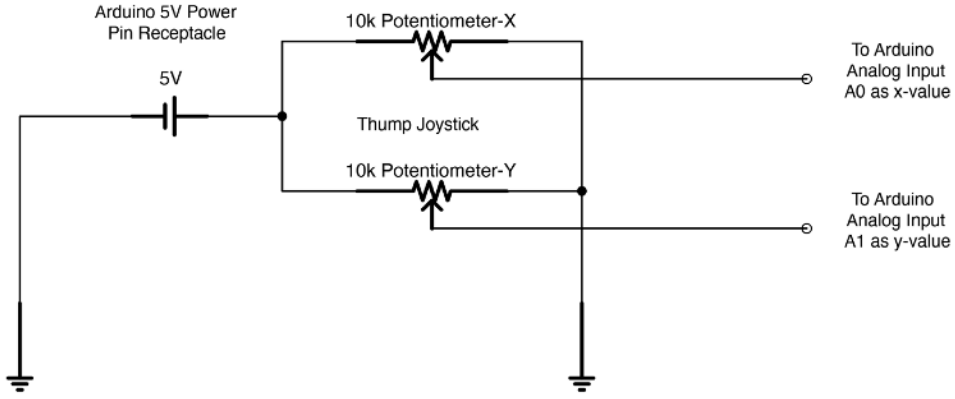
Circuit 1

How does the joystick function? The potentiometer inside the joystick is a three-terminal resistor with a *sliding contact* that forms an adjustable voltage divider. When the joystick is in the center position, the potentiometer voltage is 2.5V as configured in Circuit 1.



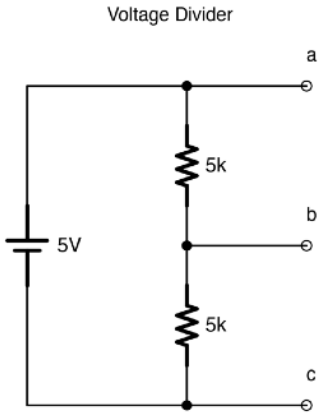
Circuit 2

The two electronic circuits shown in Circuit 2 are identical. Circuit 1 can be redrawn with two potentiometers which is what the joystick has keeping in mind that the joystick grip automatically returns to the center position once released as shown in Circuit 3.



Circuit 3

If the potentiometer is viewed as an adjustable voltage divider, it can be simplified as shown in Circuit 4.



Circuit 4

The voltage divider contains two or more resistors in *series* as opposed to being in *parallel*. Using a voltmeter, measuring the voltage across terminals *a* and *b* is 2.5 volts. The voltage across the terminals *b* and *c* is also 2.5 volts. The voltage across *a* and *c* is 5.0 volts, the same voltage as the power supply.

If the resistance across *a* and *b* is replaced with a *2.5k ohm resistor* instead of the 5k, then the voltage is about 1.7 volts. To do the calculation, use Ohm's Law.

$$Ohm's \quad Law$$

$$Voltage = Current \times Resistance$$

$$V = IR$$

Equation 1

The total resistance of a *voltage divider* is mathematically shown as:

$$R_{total} = R_1 + R_2 + \cdots$$

Equation 2

Using Equation 2, the total resistance (*2.5k*) + (*5k*) is (*7.5k*) ohm where *k* is for *kilo* which is a *SI Multiple* prefix for the value of 1000. So 2.5k is 2,500. The total resistance is then 7,500 ohm.

To calculate the voltage, the *1.7 volts* across *a* and *b*, *Kirchhoff's Current Law* states the current is equal throughout the entire circuit if the resistor's are in *series*.

$$I = \frac{V}{R}; \quad \frac{V_{total}}{R_{total}} = \frac{V_{ab}}{R_{ab}}; \quad V_{ab} = \frac{R_{ab}V_{total}}{R_{total}}; \quad 1.7V \approx \frac{(2500\Omega)(5.0V)}{7500\Omega}$$

When the sliding contact moves across the resistor in the potentiometer, the voltage varies directly proportional to the resistance. The total resistance remains constant at 10k ohm (Ω). The sliding contact along the resistor, varies the resistance on either sides of the contact.

$$R_{Potentiometer} = R_{Ground} + R_{Logic\ Voltage} \qquad \text{Equation 3}$$

If the sliding contact moves towards the voltage supply, the voltage becomes 5 volts. If the contact moves towards the ground, then the voltage becomes zero (0) volts, all the while, the total resistance for the potentiometer remains a constant.

$$R_{Ground} = R_{Poteniometer} - R_{Logic\ Voltage} \qquad \text{Equation 4}$$

If the resistance on the side towards the logic voltage is $1.45k\ \Omega$ and knowing the resistance for the potentiometer, then the resistance on the side towards the ground connection of the circuit is $8.55k\ \Omega$. Use Ohm's Law to calculate the voltages. Simply use Equation 3 to confirm the resistance of the potentiometer.

Computation

Potentiometer

Voltage ranges from the ground (0V) to the logic voltage (5V) where the potentiometer is connected to the MCU analog pin

10-bit Conversion

ADC Readings: from zero (0) to 1023 coming from the MCU analog pin.

Linear Mapping using the map() function

x-component = map(x, 0, 1023, -1.0, 1.0)
y-component = map(y, 0, 1023, -1.0, 1.0)

The joystick is built with two potentiometers that varies the voltage from the voltage supply where the variable voltage is called an analog signal. The MCU is a digital device that can handle zeros

(0) and ones (1). The MCU like the Arduino Uno cannot handle analog signals but it can convert analog signals to digital values by using an analog-to-digital converter (ADC).

Arduino Uno has a 10-bit ADC converter that gives 1024 digital values. If the analog voltage reference is 5 volts, then an analog signal of zero (0) volts gives a digital value of zero (0). If the analog signal is 5 volts, then the digital conversion is 1023.

The map() function as described under Arduino Reference - Functions - Math: states that the map() function re-maps a number from one range to another. In basic algebra, it is a linear equation mapping x to f(x) where:

$$f(x) = a\,x + b \qquad \text{Equation 5}$$

To fully describe the map() function, the map() function is derived from four different forms of the same linear equation.

A. The common slope-intercept form for the equation of a line with slope m and y-intercept:

$$y = m\,x + b \qquad \text{Equation 6}$$

B. The slope m of the line through two points P1(x1, y1) and P2(x2, y2):

$$m = \frac{(y2 - y1)}{(x2 - x1)} \qquad \text{Equation 7}$$

C. The point-slope form for the equation of a line through P1(x1, y1) with slope m:

$$y - y1 = m(x - x1) \qquad \text{Equation 8}$$

D. The equation for the map() function is derived from both the slope m Equation 7 and the point-slope form Equation 8:

$$y = (y2 - y1)\left(\frac{x - x1}{x2 - x1}\right) + y1 \qquad \text{Equation 9}$$

The problem, when programming using the map() function, is the ordered pair (x, y) where the (x, y) values within P1(x1, y1) and P2(x2, y2) are mismatched. As an analogy when using Equation 7, a test for beginners try to convert Fahrenheit value to Celsius value (or Celsius to Fahrenheit) by matching the boiling-points and the freezing-points of water thus creating the two points P1 and P2. The mismatching comes about when mixing apples with oranges or precisely, mixing a boiling-point value with a freezing-point value of water as a single point in an order pair P(x, y).

Matching the values in an order pair should be for example the temperature of boiling water as P(212, 100) or as P(100, 212) which is an equal temperature with different units. That is to say that water boils at 212 degrees Fahrenheit and also at 100 degrees Celsius. With the boiling and freezing-points of water, it gives the two points necessary to do a conversion from a known value using the map() function to an unknown value. Try to convert 75 degrees Fahrenheit to Celsius using equation (4). The arguments necessary for the inputs for the map(x, x1, x2, y1, y2) function including a known value (75) to convert to y (degrees Celsius) ought to look like the following:

Map function: *y* = map(*x, x1, x2, y1, y2*) with its *parameters*

y = map(75, 32, 212, 0, 100) where the ordered pairs are (x1, y1) = (32, 0) and (x2, y2) = (212, 100).

So what is the relationship between a 10-bit number and the value 1024? The range of values, from zero (0) to one (1), as discussed in Images 1 & 2, were arbitrarily chosen but chosen to convey the concept of normalization. Adjusting values to unity (1) for example the 10-bit ADC readings with its values from zero (0) to 1023 can be scaled by dividing the values by 1024 giving a range of values from zero (0) up to one (1). Fortunately, one can convert back by simply multiplying the normalized values by 1024. Referring back to the computation, the linear mapping when using the map() function can be used to normalize the range of values. The 10-bit number is 1024 which means that the 10-bit number contains 1024 values starting from zero (0).

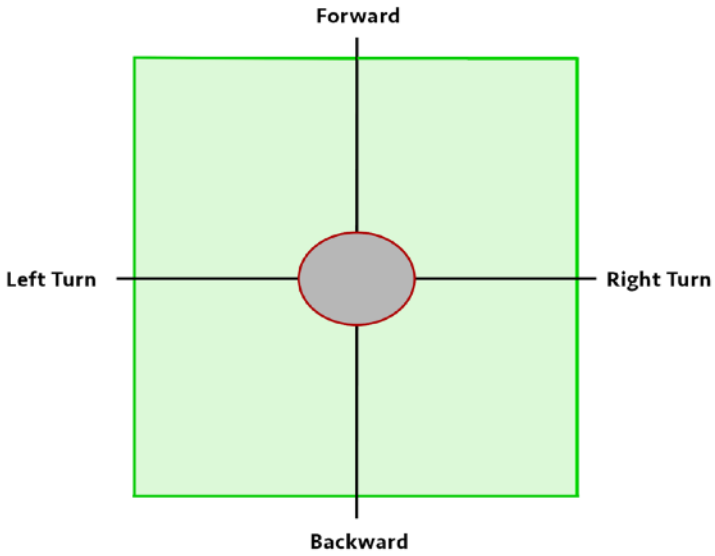
Keeping in mind the computational sequence, (1) potentiometer voltages, (2) 10-bit conversion, and (3) linear mapping in relation with arrow A in Images 1 & 2, the x-component is 0.5 which means that the joystick moved the half the distance. The value 0.5 is a normalized value. The potentiometer relative to the x-axis had to supply three-quarters of the logic voltage (3.75V). The 10-bit analog to digital converter (ADC) reading would be 768, three-fourths of 1024. This also goes for the y-component with a value of 1.0 also normalized. The joystick moved the entire allowable distance from its center position. The potentiometer for the y-axis supplied all its logic voltage (5.0V). An ADC reading here would be 1023. As soon as the joystick is released, it returns automatically back to the center with the ADC readings at 511. Circuit 1 shows a single potentiometer circuit but the joystick has two potentiometers therefore two identical circuits. See Circuit 3. These circuits as shown are connected to Arduino analog inputs. As discussed, the logic voltage in Circuit 1 is 5V, but its output is one-half which means that one of the joystick axis still remains in the center position

Joystick Algorithm

The joystick algorithm is simple. It is based on eight conditions related to the joystick's position from the center to any one of the eight octants within the analog bounding area, Image 1. As discussed, the components are the x and y values. Although the x and y are independent of each other, they can be considered a vector.

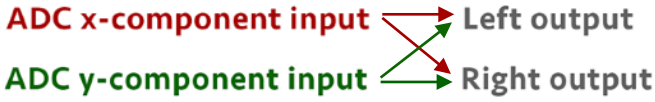
Although simple, it can output results that at best may seem correct until tested. The convention for the x-axis in relation with the joystick is used along the horizontal axis analogous to a steering wheel of a vehicle where it is used for turning the robot *left* or *right*. Notice the indicated turns in Image 3. For the y-axis again in relation with the joystick is used along the vertical axis analogous to an accelerator of a vehicle for moving the robot forward and backward. The the grip of the joystick is always centered to automatically stop the robot. To move the robot, move the grip of the joystick in any direction away form the center within the area or along one of the axis. The robot can simultaneously move forward and turn when the joystick is positioned in any one of the areas away from the axes. When the joystick is moved to make a *left* turn along the axis, the robot turns in place. When the joystick is moved along the vertical axis, robot moves forward or backward only.

Image 3



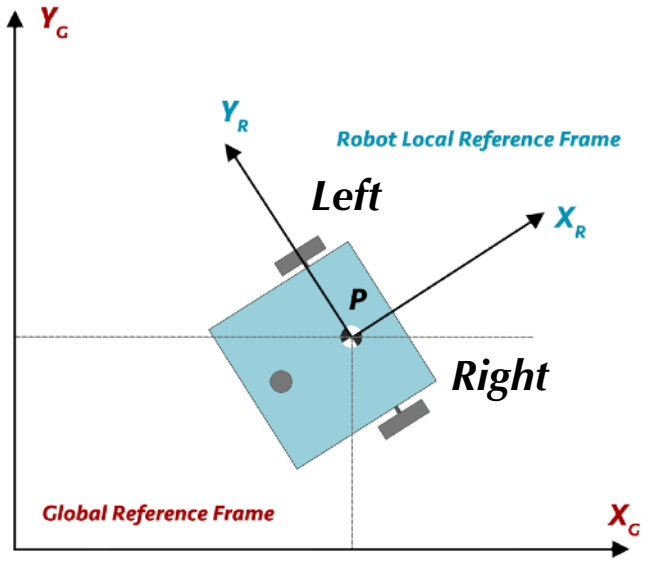
The joystick algorithm converts the ADC digital reading inputs to the *Left* and *Right* outputs for each of the motor directions. See mapping the joystick algorithm below. This is not to say that a single joystick coordinate value is mapping a single ADC reading for example to a single output like the *left* motor. Unfortunately each of the *Left* and the *Right* outputs require both ADC inputs.

Mapping the Joystick Algorithm
from the Inputs to the Outputs



The decisions made within the joystick algorithm and as one might observe, changes the mapping. Each mapping has to be tested to verify a rational outcome not for one but for 8 conditionals relative to the joystick's movement within everyone one of the 8 octants. To reiterate it differently, it is based on eight conditions related to the joystick's position from the center to any one of the eight octants. There is the total number of eleven boolean conditions including the center position and both the x-axis and the y-axis.

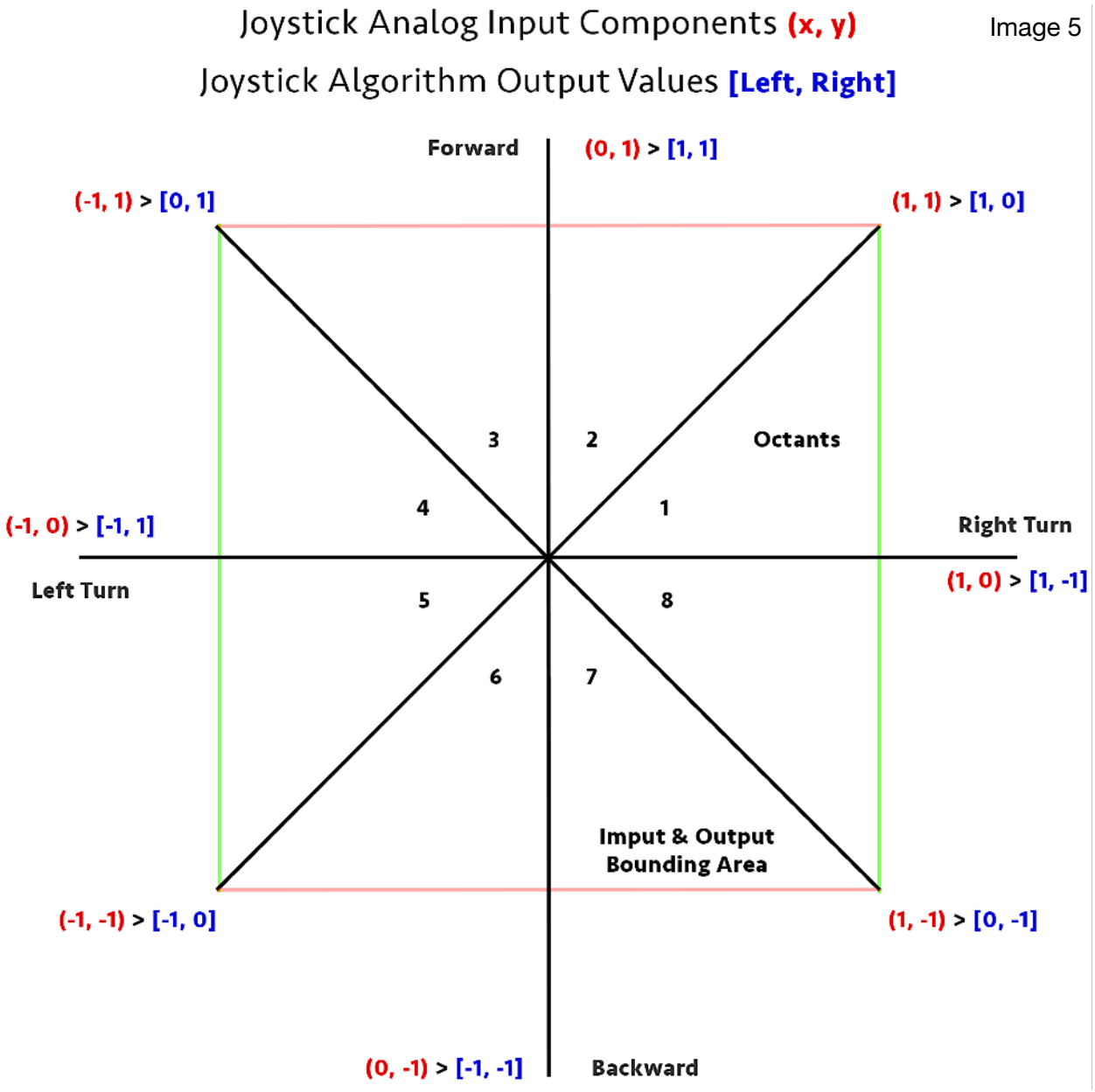
Image 4



When speaking about the *Left* and the *Right*, the Differential Drive Mobile Robot has two independently driven wheels, the *Left* wheel and the *Right* wheel as shown in Image 4. The trick when mapping an entire area would have to convert the input values to the *Left* and *Right* based on these boolean conditions.

The first level of conditions are simple:

(1) If the joystick lies within any of the octant areas which includes the forward or backward movement and the *left* and *right* turns, then the movements depend on the second level of conditions.
(2) If the joystick lies on the vertical axis then the forward and backward movement equals to y-input only.
(3) If the joystick lies on the the horizontal axis, then the *left* and *right* turns equals the x-input only.
(4) If the joystick lies in the center, then the movement is zero.



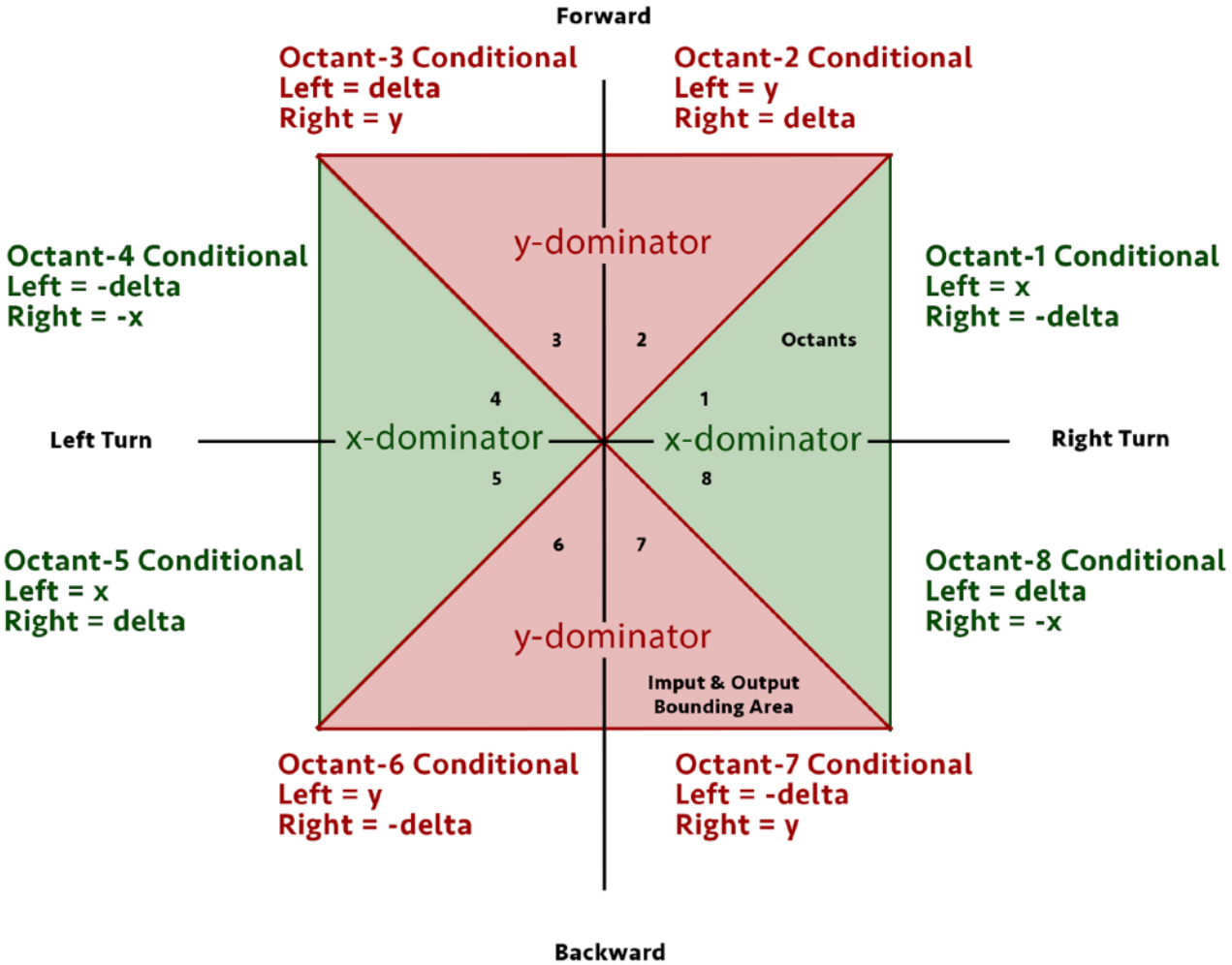
The second level of conditions embedded within the first level of condition (1) depends where the joystick is positioned in which one of the eight octants. The ADC reading inputs for the components (x, y) are used to determine which octant is used to assign the *Left* and *Right* outputs in the Joystick Algorithm. From Image 6 in the red areas along the vertical axis shows that the y-input is the dominating value. The green area along the horizontal axis shows that the x-input is the dominating value. The resultants for each octant are shown in Image 6.

Joystick Analog Input Components **(x, y)**

Joystick Algorithm Output Values **[Left, Right]**

Image 6

$$delta = 0 \leq (\, ||y| - |x|| \,) < 1$$



The only mathematical equation in the Joystick Algorithm is the delta equation. As previously discussed depending on the conditional, the delta value is simply taking the absolute values of each component x & y of the joystick inputs. Subtracting one from the other and then taking the absolute value of the difference. The final result is a normalized positive value greater than or equal to zero (0) and less than one (1).

$$delta = ||x| - |y||, \quad where\ (0 \leq delta < 1)$$ Equation 10

Image 5 is used for testing the second level of conditions. Once the joystick moves within one of areas of the octants, the value for either the component x or y is no longer equal to *0* or *1* but somewhere in-between. In Image 5 where the inputs for octants *1* & 2 are (0, 1), (1, 1) and (1, 0) shows that the outputs for *Left* is 1 but *Right* varies from *1* to *0* then from *0* to *-1* respectively. When comparing the variants with Image 6, notice the *Right* varies also with the variable *delta*. The testing should match the inputs as the joystick transitions around the bounding area. Many times depending how the joystick is physically oriented and how either the joystick or the L298N module is connected to the MCU, the mathematical signs for the values *Left* and *Right* shown in Image 6 may vary as well. The wiring among the three devices, joystick, Uno, and L298N is the reason why testing is necessary for obtaining the correct Left and Right output values. These output values are not intuitive. Each output has to be tested. Tested against what is expected. As the joystick moves from one octant area to another it becomes vitally important to ensure that the Joystick Algorithm gives rational resultants comparable to the output values in Image 5.

The C++ source for the Joystick Algorithm is listed for your convenience. The ***Joystick-Uno-L298N*** depository with the ***Joystick Algorithm*** can be downloaded from ***Github***.[1]

1. Github: https://github.com/MageMCU/Joystick_Uno_L298N

Joystick Algorithm taken from the Class Joystick

```
void Joystick::_joystick()
{
  double const _tol = 0.04;

  double const _inputX = (double)_processX / (double)_MIDPOINT_HI;
  double const _inputY = (double)_processY / (double)_MIDPOINT_HI;

  double _x = 0;
  double _y = 0;

  double delta = abs(abs(_inputX) - abs(_inputY));
  if (abs(_inputX) > _tol && abs(_inputY) > _tol)
  {
    if (abs(_inputX) <= abs(_inputY))
    {
      if (_inputX > _ZERO && _inputY > _ZERO)
      {
        _x = _inputY;
        _y = delta;
      }
      else if (_inputX > _ZERO && _inputY < _ZERO)
      {
        _x = -delta;
        _y = _inputY;
      }
      else if (_inputX < _ZERO && _inputY > _ZERO)
      {
        _x = delta;
        _y = _inputY;
      }
      else if (_inputX < _ZERO && _inputY < _ZERO)
      {
        _x = _inputY;
        _y = -delta;
      }
    }
    else if (abs(_inputX) > abs(_inputY))
    {
      if (_inputX > _ZERO && _inputY > _ZERO)
      {
        _x = _inputX;
        _y = -delta;
      }
      else if (_inputX > _ZERO && _inputY < _ZERO)
      {
        _x = delta;
        _y = -_inputX;
      }
      else if (_inputX < _ZERO && _inputY > _ZERO)
      {
        _x = -delta;
        _y = -_inputX;
      }
      else if (_inputX < _ZERO && _inputY < _ZERO)
      {
        _x = _inputX;
        _y = delta;
      }
    }
  }
```

```
61    else if (abs(_inputX) > _tol && abs(_inputY) < _tol)
62    {
63      _x = _inputX;
64      _y = -_inputX;
65    }
66    else if (abs(_inputX) < _tol && abs(_inputY) > _tol)
67    {
68      _x = _inputY;
69      _y = _inputY;
70    }
71    else if (abs(_inputX) < _tol && abs(_inputY) < _tol)
72    {
73      _x = 0;
74      _y = 0;
75    }
76
77    _normalLeft = _x;
78    _normalRight = _y;
79  }
80
```

The order of the conditionals were not placed for the most efficient performance but to convey the octant areas in a clear and representative fashion.

Article 1001: Joystick Algorithm
Was: *Joystick Algorithm & Compass Code*
The compass code was moved to another article.
By Jesse Carpenter
Version 20190817
Carpenter Software LC
https://carpentersoftware.com