“LOCALISATION OF BOT”

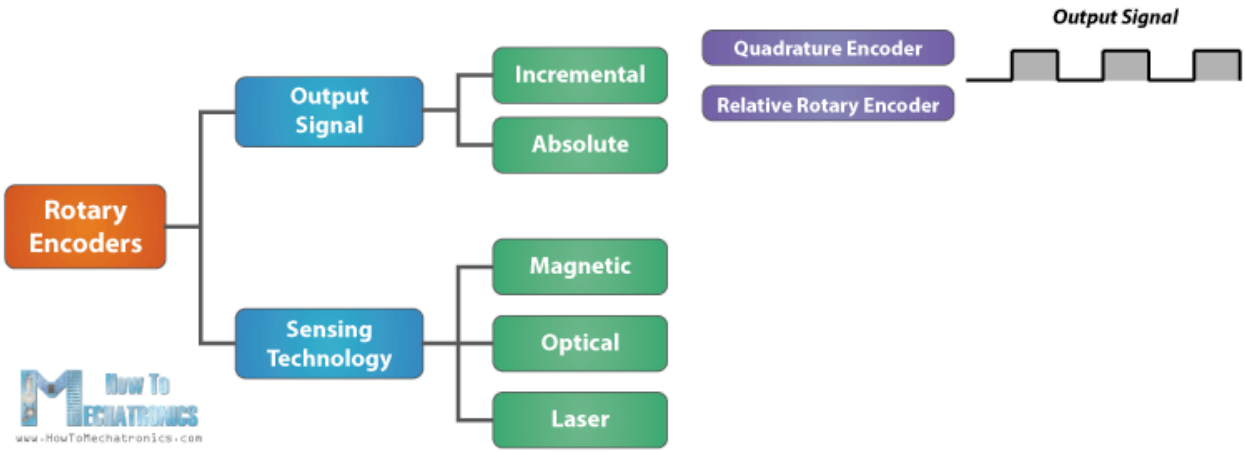
Problem statement : To estimate the position of a differential drive robot using motion sensors.
Localisation is simply computing the position of robot with respect to its reference frame. To make robot autonomous one of the key information the robot needs is its "position".With the use of this information different methods can be used for navigating the robot and perform desired tasks

Modules
Rotary encoder interfacing on atmega128, PID, Differential drive kinematics, interrupts, uart and motor driver.

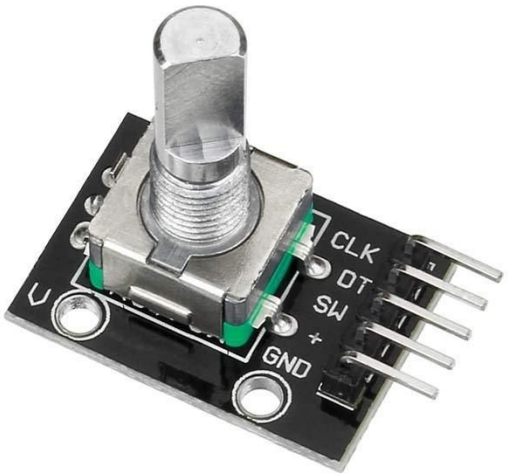
Things used
Atmega128, motor driver L293D, Voltage regulator 7805, rotary encoder, motors.

Basics required
Need to learn the basics of AVR coding like setting pin high or low , using pin as input or output , bit manipulation. You must be able to read datasheet.

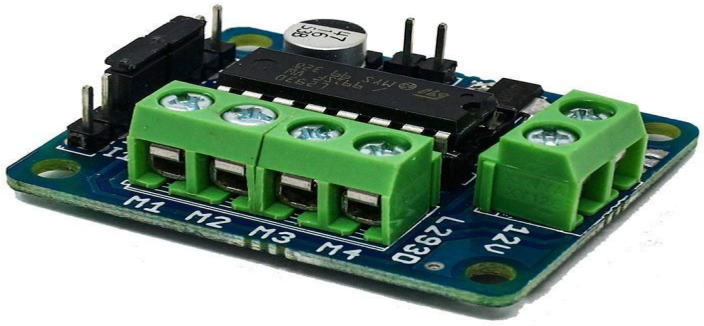
Describing the materials using the project.
Rotary encoder (ky-040)
A rotary encoder is a type of position sensor which is used for determining the angular position of a rotating shaft. It generates an electrical signal, either analog or digital, according to the rotational movement.There are many different types of rotary encoders which are classified by either Output Signal or Sensing Technology. The particular rotary encoder that we will use in this tutorial is an incremental rotary encoder and it’s the simplest position sensor to measure rotation.



To learn more about encoder visit :
https://howtomechatronics.com/tutorials/arduino/rotary-encoder-works-use-arduino/
Youtube video : https://www.youtube.com/watch?time_continue=2&v=v4BbSzJ-hz4

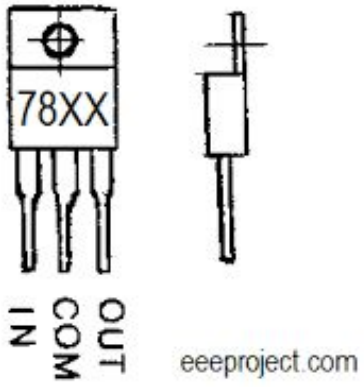


Motor driver
A motor driver is a small Current Amplifier whose function is to take a low-current control signal and then **turn** it into a higher-current signal that can drive a motor. The L293D is a typical Motor Driver which can drive 2 DC motors simultaneously. Motor Driver ICs are primarily used in **autonomous** robotics only.



7805 IC :

It converts 7-12 volt input to 5 volt output.



STEPS FOLLOWED BY OUR TEAM:

STEP 1: Creating basic circuit for testing the use external interrupt.

First of all we created a simple circuit which basically did was to switch led on and off with the use of switch to create external interrupt. Whenever the switch was pressed it gave a high voltage to the pin it was connected. Then we used the rising edge at PIN D0 to call the interrupt function.

Use of datasheet to get the values of specific registers to set the interrupt at Rising Edge.

Table 48. Interrupt Sense Control[1]

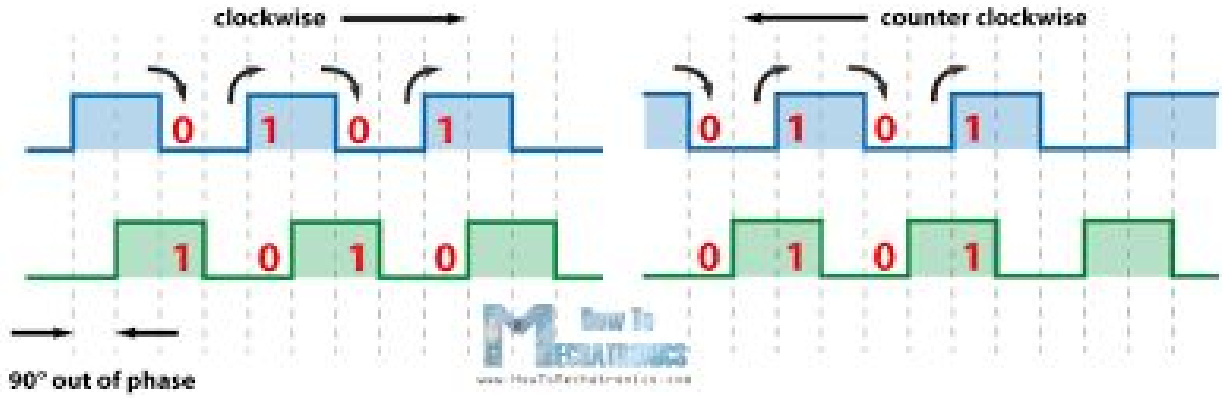
ISCn1	ISCn0	Description
0	0	The low level of INTn generates an interrupt request.
0	1	Reserved
1	0	The falling edge of INTn generates asynchronously an interrupt request.
1	1	The rising edge of INTn generates asynchronously an interrupt request.

STEP 2: Creating a circuit to test interrupts sent by rotatory encoder.

We made a circuit similar to that we made for checking interrupt by using switch the only difference was that we used our encoder instead of switch. The functioning of encoder is stated above. We used OUTPUT A for sending interrupts in PIN D1 and OUTPUT B's signals were received at PIN D2.

Here also we used Rising Edge of the signal provided by OUTPUT A to call interrupt function and the state of the OUTPUT B was used to check whether the rotation of encoder was clockwise or anticlockwise.

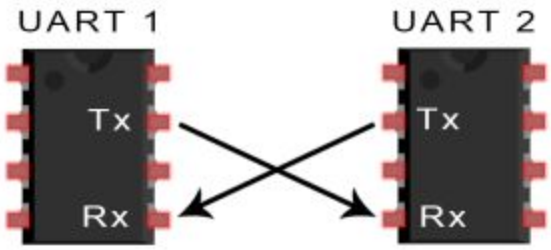
The final direction of our rotation was shown using leds.



Step 3 : Use of UART communication for checking outputs by encoder.

UART stands for Universal Asynchronous Receiver/Transmitter. It's not a communication protocol like SPI and I2C, but a physical circuit in a microcontroller, or a stand-alone IC. A UART's main purpose is to transmit and receive serial data

In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART:

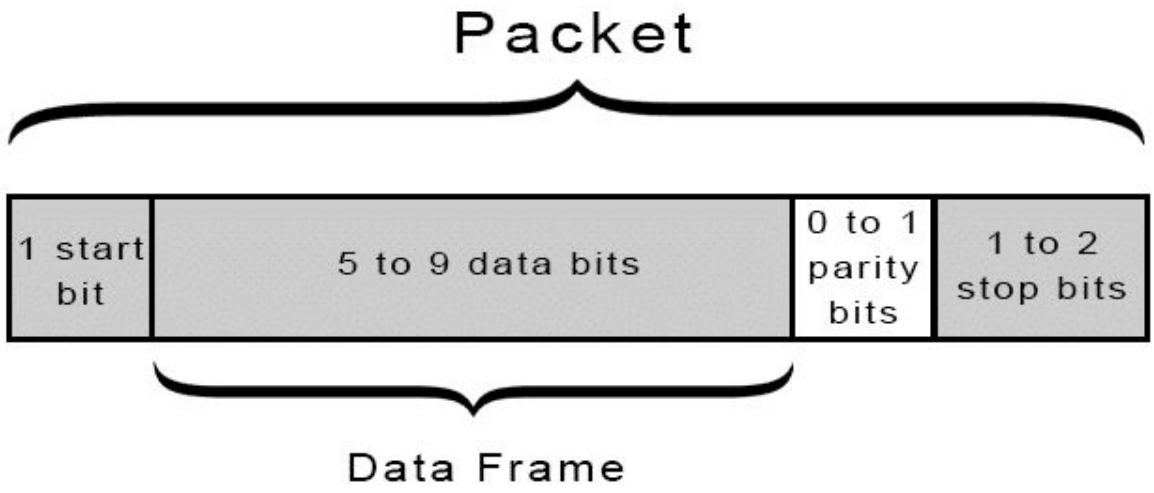


UARTs transmit data *asynchronously*, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.
When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate.* Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate. The baud rate between the transmitting and receiving UARTs can only differ by about 10% before the timing of bits gets too far off.

Both UARTs must also must be configured to transmit and receive the same data packet structure.

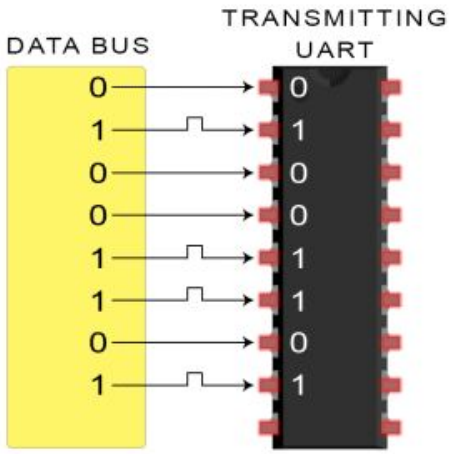
Wires Used	2
Maximum Speed	Any speed up to 115200 baud, usually 9600 baud
Synchronous or Asynchronous?	Asynchronous
Serial or Parallel?	Serial
Max # of Masters	1
Max # of Slaves	1

Data in serial format:

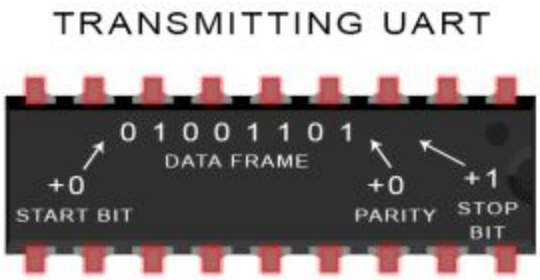


STEPS OF UART TRANSMISSION

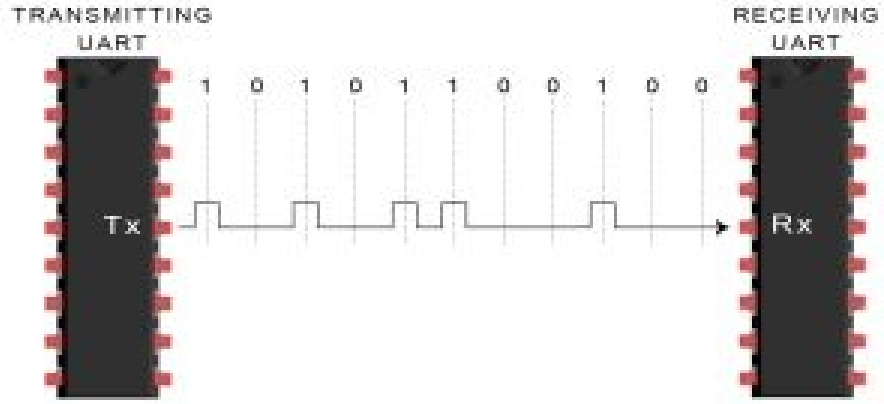
1. The transmitting UART receives data in parallel from the data bus:



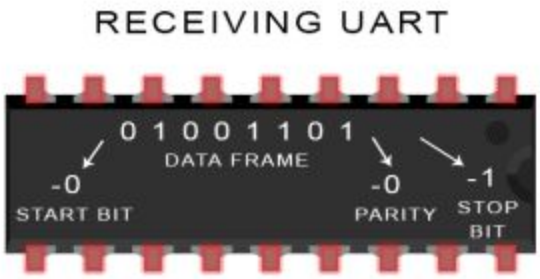
2. The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame:



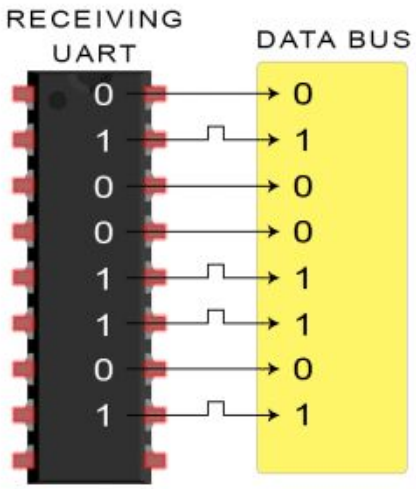
3. The entire packet is sent serially from the transmitting UART to the receiving UART. The receiving UART samples the data line at the pre-configured baud rate:



4. The receiving UART discards the start bit, parity bit, and stop bit from the data frame:



5. The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end:



UARTn Control and Status Register A – UCSRnA:

RXCn: UART Receive Complete. This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty (i.e., does not contain any unread data).

UDREn: USART Data Register Empty .The UDREn flag indicates if the transmit buffer (UDRn) is ready to receive new data. If UDREn is one, the buffer is empty, and therefore ready to be written.

UARTn Control and Status Register B – UCSRnB:

RXENn: Receiver Enable, writing this bit to one enables the USARTn Receive.

TXENn: **Transmit** Enable, Writing this bit to one enables the USARTn to transmit.

UCSZn: Character Size. UCSZn0,UCSZn1,UCSZn2 , these bits are used to set size of data bits in a frame the Receiver and Transmitter use.

ISSUE: UART was not able to transmit our number of counts in integer form so we had to use itoa function which converts counts to string and displays it in decimal form on screen. For that we had to include library string.h and used function as follows: -- Itoa(count, name_of_string, 10)

Step 4: Understanding the motion of our bot

The motion of our robot completely depend on the differential drive kinetics.
The differential drive kinetics can be understood using the following link:

http://rossum.sourceforge.net/papers/DiffSteer/?fbclid=IwAR3-p_x1Ys_f6PeGC9Tg5Vroq_sVpvDhdNJxFnwylu37I_sBooGQ9wje-4M#d6

Many mobile robots use a drive mechanism known as differential drive. It consists of 2 drive wheels mounted on a common axis, and each wheel can independently being driven either forward or backward. While we can vary the velocity of each wheel, for the robot to perform rolling motion, the robot must rotate about a point that lies along their common left and right wheel axis. The point that the robot rotates about is known as the ICC

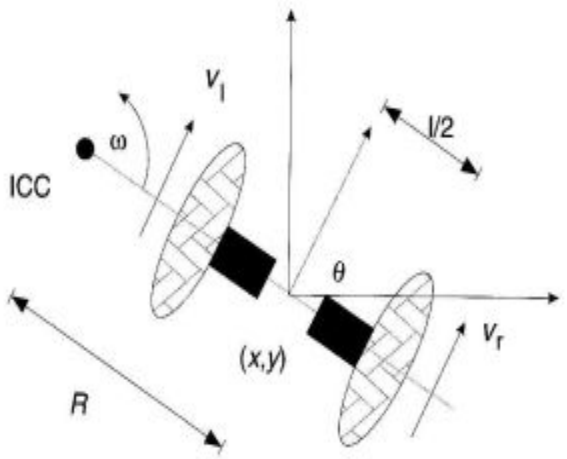


Figure 1: Differential Drive kinematics (from Dudek and Jenkin, *Computational Principles of Mobile Robotics*.

By varying the velocities of the two wheels, we can vary the trajectories that the robot takes. Because the rate of rotation ω about the ICC must be the same for both wheels, we can write the following equations:

$$\omega\,(R + l/2) = V_r \quad (1)$$
$$\omega\,(R - l/2) = V_l \quad (2)$$

where l is the distance between the centers of the two wheels, V_r , V_l are the right and left wheel velocities along the ground , and R is the signed distance from the ICC to the midpoint between the wheels. At any instance in time we can solve for R and ω:

$$R = \frac{l}{2}\frac{V_l + V_r}{V_r - V_l};\ \omega = \frac{V_r - V_l}{l}; \quad (3)$$

There are three interesting cases with these kinds of drives.

1. If $V_l = V_r$, then we have forward linear motion in a straight line. R becomes infinite, and there is effectively no rotation - ω is zero.
2. If $V_l = -V_r$, then $R = 0$, and we have rotation about the midpoint of the wheel axis - we rotate in place.
3. If $V_l = 0$, then we have rotation about the left wheel. In this case $R = \frac{l}{2}$. Same is true if $V_r = 0$.

Note that a differential drive robot cannot move in the direction along the axis - this is a singularity. Differential drive vehicles are very sensitive to slight changes in velocity in each of the wheels. Small errors in the relative velocities between the wheels can affect the robot trajectory. They are also very sensitive to small variations in the ground plane, and may need extra wheels (castor wheels) for support.

2 Forward Kinematics for Differential Drive Robots

In figure 1, assume the robot is at some positon (x, y), headed in a direction making an angle θ with the X axis. We assume the robot is centered at a point midway along the wheel axle. By manipulating the control parameters V_l , V_r, we can get the robot to move to different positions and orientations. (note: V_l , V_r) are wheel velocities along the ground).

Knowing velocities V_l, V_r and using equation 3, we can find the ICC location:

$$ICC = [x - R\,sin(\theta)\,,\; y + R\,cos(\theta)] \tag{4}$$

and at time $t + \delta t$ the robot's pose will be:

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} cos(\omega\delta t) & -sin(\omega\delta t) & 0 \\ sin(\omega\delta t) & cos(\omega\delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICC_x \\ y - ICC_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega\delta t \end{bmatrix} \tag{5}$$

This equation simply describes the motion of a robot rotating a distance R about its ICC with an angular velocity of ω.

STEP 5: Carving out logic behind our main calculation using differential drive kinetics.

Our project's main application is self-localization so we wanted some mathematical equations which could be used to deduce the approx. distance travelled by our robot in the X-Y plane. In short we wanted the final coordinates of our robot with reference to our initial position taken as origin. Thus to achieve what we needed was to use some equations to deduce x and y after receiving the count values by each wheel every moment i.e. we had kept our calculation in the main while loop.

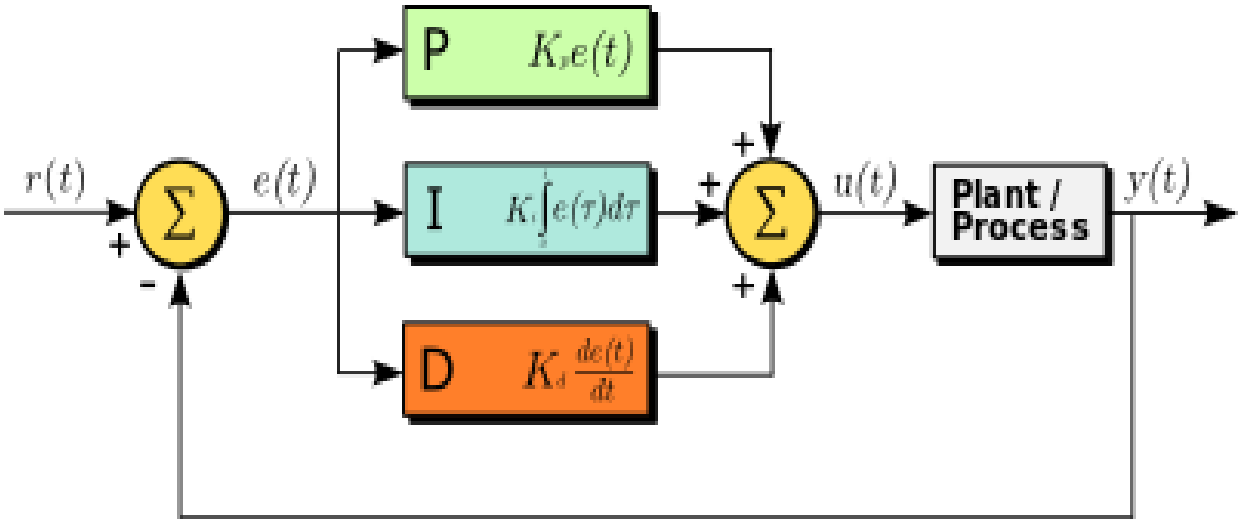
This approach was a complete failure because there was very high risk of skipping a lot of counts. As the rate at which the counts were changing could have exceeded the rate at which the main loop code was executed.

To solve this problem we thought of putting timer interrupt so that we can do calculations on a bulk of reading and not after every count received.

We did the main calculation in the main while loop itself and we just received the reading given by encoders in timer interrupt. Calculations were necessary in main while loop so that we don't skip some counts while our calculation are done in timer interrupt. Thus we did is save the counts received by each wheel in some other variables and set the value of original variables to 0, thus our bot neither skips the counts and nor it changes the values of variables between the ongoing calculations.

This was our final approach with which we knew the current location of our bot.

Step 6 : implementing PID



In this model:

- Term **P** is proportional to the current value of the error. For example, if the error is large and positive, the control output will be proportionately large and positive, taking into account the gain factor "K". Using proportional control alone will result in an error between the setpoint and the actual process value, because it requires an error to generate the proportional response. If there is no error, there is no corrective response.
- Term **I** accounts for past values of the error and integrates them over time to produce the **I** term. For example, if there is a residual error after the application of proportional control, the integral term seeks to eliminate the residual error by adding a control effect due to the historic cumulative value of the error. When the error is eliminated, the integral term will cease to grow. This will result in the proportional effect diminishing as the error decreases, but this is compensated for by the growing integral effect.

- Term **D** is a best estimate of the future trend of the error, based on its current rate of change. It is sometimes called "anticipatory control", as it is effectively seeking to reduce the effect of the error by exerting a control influence generated by the rate of error change. The more rapid the change, the greater the controlling or dampening effect.[1]

Tuning – The balance of these effects is achieved by loop tuning to produce the optimal control function. The tuning constants are shown below as "K" and must be derived for each control application, as they depend on the response characteristics of the complete loop external to the controller. These are dependent on the behaviour of the measuring sensor, the final control element (such as a control valve), any control signal delays and the process itself. Approximate values of constants can usually be initially entered knowing the type of application, but they are normally refined, or tuned, by "bumping" the process in practice by introducing a setpoint change and observing the system response.

We have basically used PID to move our bot on required angle.

So error = required angle – present angle.

And total error = Kp * error + Kd * (previous error – error) and Kp, Kd can be calculated by tuning them practically.

Leftrpm = basevelocity + total error

Rightrpm = basevelocity - total error

Using PID we are determining speeds of our both wheels so that it can move on required angle. Suppose the error is 0 so both wheels will run with same RPM and it will follow straight path with base velocity.

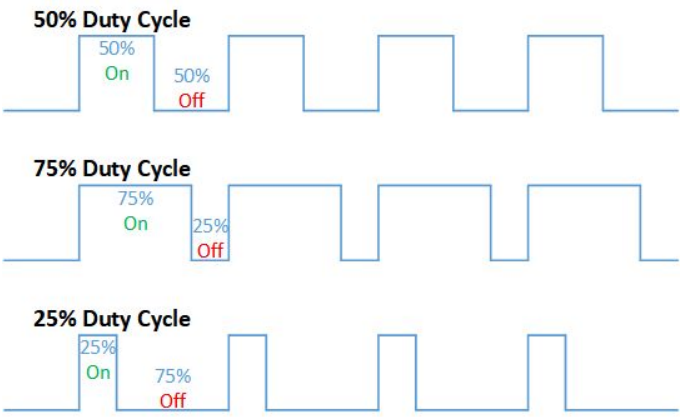
STEP 7 : Implementing PWM

Pulse width modulation (**PWM)** is a method of reducing the average power delivered by an electrical signal, by effectively chopping it up into discrete parts. The average value of voltage (and current) fed to the load is controlled by turning the switch between supply and load on and off at a fast rate. The longer the switch is on compared to the off periods, the higher the total power supplied to the load.

DUTY CYCLE :

X% : of duty cycle means that our motors will run at X % of our max RPM.

Hence, 100% duty cycle means that our motors will run at max RPM.



We have used fast PWM in non-inverting mode. And hence the following registers have to be set as shown:

1) We have used the following Bits of Registers for non-inverting mode of PWM

Table 59. Compare Output Mode, Fast PWM

COMnA1/COMnB1/ COMnC1	COMnA0/COMnB0/ COMnC0	Description
0	0	Normal port operation, OCnA/OCnB/OCnC disconnected.
0	1	WGMn3:0 = 15: Toggle OCnA on Compare Match, OCnB/OCnC disconnected (normal port operation). For all other WGMn settings, normal port operation, OCnA/OCnB/OCnC disconnected.
1	0	Clear OCnA/OCnB/OCnC on compare match, set OCnA/OCnB/OCnC at BOTTOM, (non-inverting mode)
1	1	Set OCnA/OCnB/OCnC on compare match, clear OCnA/OCnB/OCnC at BOTTOM, (inverting mode)

Note: A special case occurs when OCRnA/OCRnB/OCRnC equals TOP

2) We have used 14th mode as shown in figure for fast PWM mode:

Mode	WGMn3	WGMn2 (CTCn)	WGMn1 (PWMn1)	WGMn0 (PWMn0)	Timer/Counter Mode of Operation[1]	TOP	Update of OCRnx at	TOVn Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCRnA	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICRn	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCRnA	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICRn	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCRnA	TOP	BOTTOM
12	1	1	0	0	CTC	ICRn	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICRn	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCRnA	BOTTOM	TOP

Note: 1. The CTCn and PWMn1:0 bit definition names are obsolete. Use the WGMn2:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

Top value in some modes of operation can be defined by OCRnA register , ICRn register or by set of fixed values. When OCRnA as top value in PWM, OCRnA register cannot be used for generating PWM output.

As we are using OCRnA register for generating PWM output, we are using ICRn as top value in PWM mode.

ICRn = X this defines the max value of our signal

OCRnA = velocity * X / 200;

Thus we can achieve the number of counts for our required velocity by the use of above formula.

PROBLEMS FACED:

Initially we were using encoder pins without pullups. So were using RC filter so that we don't miss the counts and obtain a continuous signal.As output from encoder is open drain , so to increase the open drain output to +5v we used pull up.