Building Autonomous Line Followers with Arduino and PID!

By:

\Aneesh Vartakavi/

/ Krishna\

God…Why another one?

A lot of guides and webpages already cover autonomous line following robotics with an Arduino, but most of them just throw a sketch and a few pictures and a video of their effort at you. There is limited scope for experimentation, improvisation and 'tinkering' as Massimo Banzi calls it!

People have described their efforts at building their robot in detail, but the fact still remains that it is THEIR robot, and you will obviously not build the same robot as they did. This guide in that sense intends to help you build your own robot, crafting your own design as you read along. This is what I believe, makes this unique!

This is made with both a beginner and novice in mind, so don't fret if you find something too new (read up on it before continuing) or too easy (please bear with us and continue reading!).

Hope you like this, and find robotics as much fun as we did!

Please feel free to share this and post/upload it wherever you want to! Knowledge should be free and this will always be!

Before you start…

There are very few references to commercial parts/circuit diagrams. This is intentional! Once you know what a component is and what it does, you should be able to search around for the exact information you need. This is not another spoon-feed guide!

Do not expect circuit diagrams and part assembly instructions! We focus on how to choose the part, not how to fix it up.

You need not follow the same approach as we preach in the guide, but carefully modify whatever code or design you might need. It is encouraged that you try something new and different!

Some stuff is taken from websites and forums around the web, and some of it is our own experience in building robots and helping others build them. If you find any of your material in here and you wish it removed, edited or highlighted, do please contact us; details are at the end of this guide. We'll be happy to acknowledge your request!

If you decided to build a line following robot, you probably already know what it is…let's take a closer look at the various aspects.
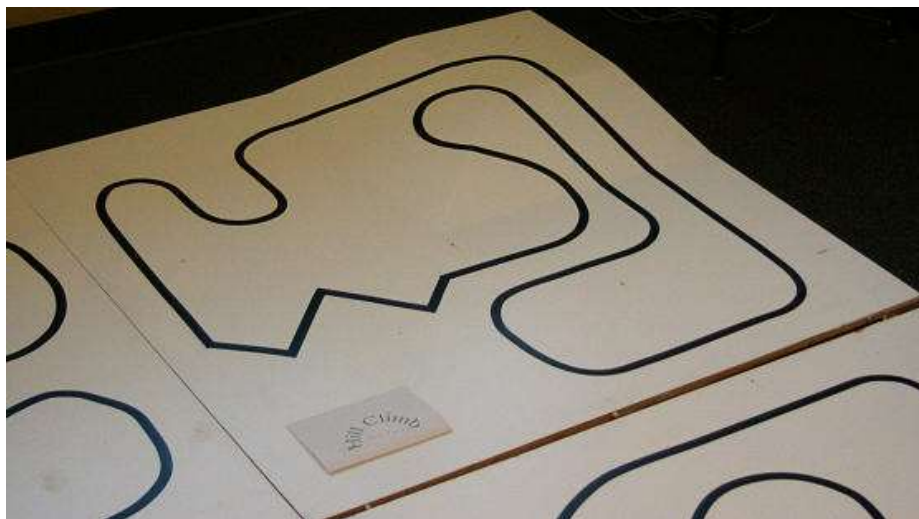
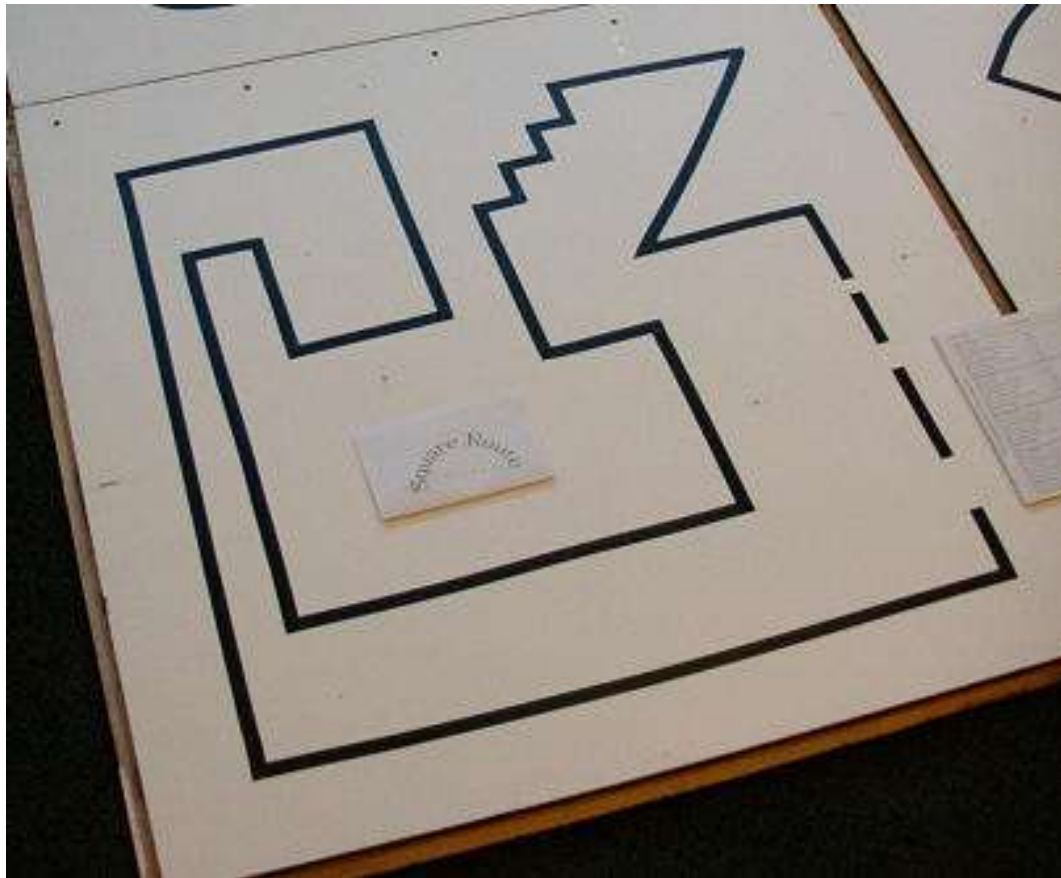# To Battle!!

The Arena

You should know what you're up against. Study these carefully!
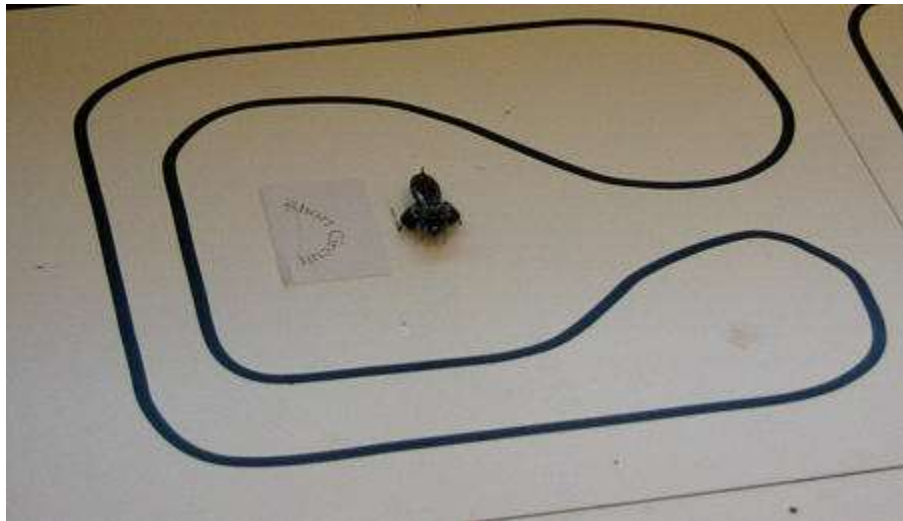
A Simple Arena

Hill Climb – An arena with varying gradient, and note the sharper turnings!

Let's spend a little more time with the "Square Route" as this arena is called.

- Notice all the turnings are acute or right angles.
- The spacing between the lines in some places (the parallel tracks on the left) is very narrow, so your sensors can read both lines and confuse your robot.
- There are gaps in the track on the right. They're not even equal in length.
- The 'stairs' in the top right are small!

Right angle turnings introduce different sensor readings. You'll understand more about these when we discuss sensor types and placement. The difficulty lies in differentiating a right and a left turn, which can be quite hard if your robot 'swings' on its track. This also means the algorithm for turning has to be sharper than a gentle turning (we'll come to handling soon). The gaps in the track can be confusing because your sensors 'read dead'. You can both interpret this as the robot has strayed away from the line, and therefore program it to find the line again, or a gap in a straight track where you ignore this and move on.

A 'tight' track like this means your robot has to be smaller, and the handling tighter.
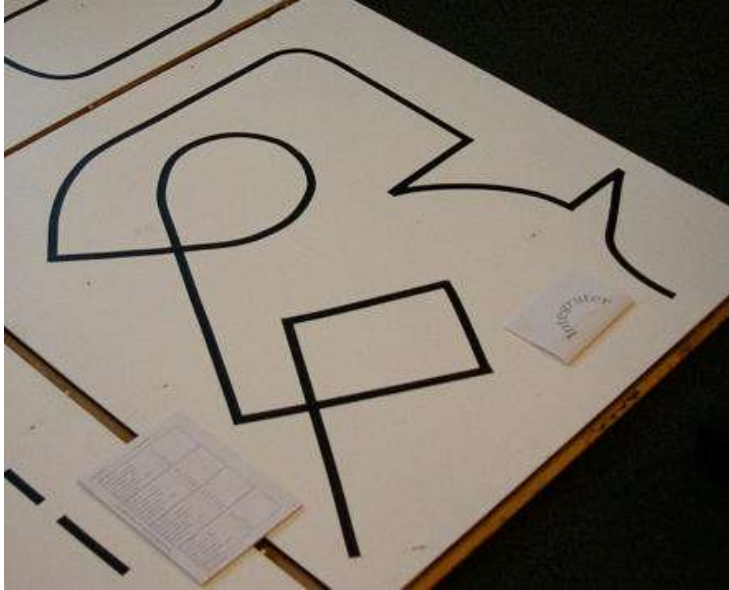
The Integrator

This is one beast of a track! Notice the track crosses back and the successive acute angle turns!
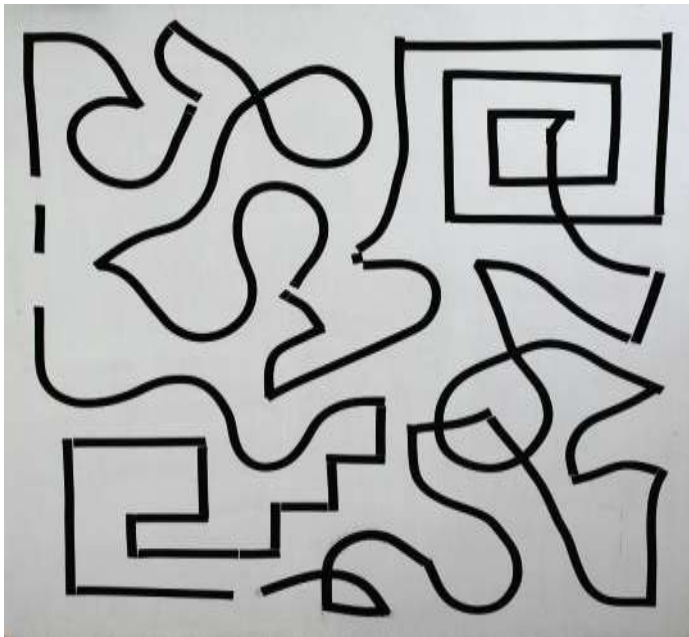
Possibly one of the hardest you'll face! Take note of its features.

Knowledge of the arena layout is important, because a robot designed for one **may** not work on the other. A general guideline is to keep your robot size as small as possible, and work from the basics to handle progressively more difficult arenas. Sensor placement is also an important aspect, as is the algorithms used to interpret these readings. We'll cover how to do the simplest tracks, and suggest how you can modify this to build better robots.
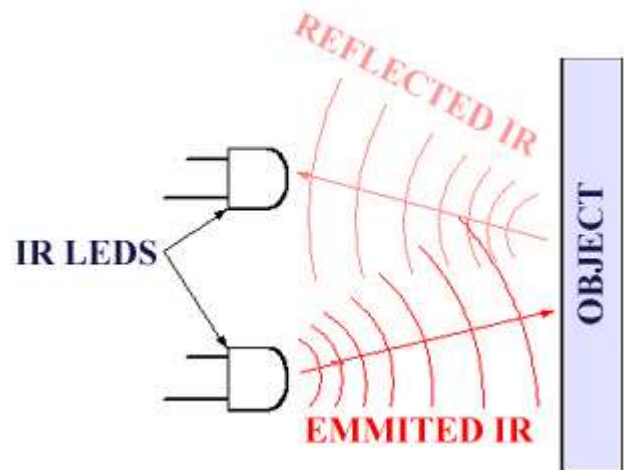
The Sensors

There are many kinds of sensors, and quite a few have been put to use in autonomous robotics. Do look up the types of sensors various other robots have used.

We chose Infra-Red Sensors, and advise you to do the same. The reasons are

- Ease of Implementation
- Cost
- Availability

IR sensors work on the principle illustrated by this diagram.

The physics which govern reflection of visible and infrared light are the same. IR LED's are easily available, and so are the sensors to detect IR radiation.

Unless shielded properly, ambient radiation may affect these readings. You can "smooth" the data after receiving, but proper placement and shielding of sensors will go a long way to help get accurate readings off the track.

You've two options now, to purchase a manufactured IR *sensormodule*, or to build one yourself.

A manufactured one has many benefits, like less susceptibility to noise and easy tuning and integration. The biggest disadvantage of a manufactured module is…it is not configurable. Exactly the opposite is true for a DIY module. If you're only just starting out, it will probably save you quite a bit of trouble if you purchase one, but it does have its limitations. Read this full section out before you make a decision.

Sensor modules in this sense refers to a circuit which can sense reflected light and convert it into a quantity which can be interpreted by a microcontroller…voltage.

We call it modules, because sometimes more than one sensor pair is integrated, forming an array of sensors!

Here are the different configurations in a module:

Single Sensor

A single sensor is enough to track a simple line, but it is comparatively very imprecise in its movements and is severely crippled in its abilities. In a way a single sensor robot does not become a line follower, it just tracks one of two edges of the line. People have posted videos and reports of their efforts on the web, check them out!

Sensor Arrays

Sensor arrays are usually configured in a straight line, and usually range from 2-8 sensor/receiver pairs in an array. They are called reflectance arrays, as they use the properties of reflection to determine distance of an object.

As a general rule, more sensors in an array giveyou

- More readings and therefore more control. On the other hand these may be hard to control and process if you're short on microcontroller performance (Don't worry about the performance just yet!).
- Readings from adjacent tracks as well(applicable only on tight tracks)
- The distances between the sensors are also fixed, and may not be the perfect for the given track width. This, as you will see later, affects the *set point* of our control algorithm.

Here's what the array from Pololu looks like. It's probably the best online retail store along with SparkFun and many others. You can save lots if you can find a similar thing with a local dealer instead of shipping them internationally if you need to.

Pololu Line Sensor Array

Selecting the Type

Most manufactured sensors give you little choice in deciding how many sensors in an array to use. Pololu has this special product where two sensor pairs at the extreme ends can be removed. You can decide how many sensors you need in your array based on:

- Availability – See what's available, and what's in your budget. Don't fantasize!
- Size – The module should be small, and the sensors should not read the adjacent track.
- Range – It is not critical, but it should be sensitive in the height you place it in your robot.

As long as these two aspects are satisfied, you should be safe…unless there is a rare, lurking exception!

There are, and always will be different opinions about the optimum number of sensor pairs in a module. The only thing common about all this is…they all worked! Don't give too much thought to how many sensors you have, as a working robot can be built from one to a dozen pairs. A personal suggestion -

Between 4 to 8 pairs in an array will serve you well, and are available easily. Remember an Arduino has only 5 analog input ports, so if you're using an Arduino and more than 5 sensors, you need to use a special IC or maybe a multiplexer to read the 'extra' sensors.

Making your own sensors

If you find a need to customize your sensors, or don't want to buy one; gear up to build your own arrays.

This is not advised for beginners. It requires you to know some analog circuitry like OPAMPS. Circuit implementations are available everywhere on the web.Remember analog electronics is not to everyone's taste!

Here's a helpful link: http://www.ikalogic.com/ir_prox_sensors.php

Look it up even if you don't intend to build one of your own!

Think twice before you venture this path, it is effort intensive compared to buying a prefabricated one! Don't get discouraged if you intended to build your own sensors though. Some of the best robot builders around have built their own sensors and swear by them!The ultimate choice lies with you and your resources!

Sensor Configurations

Most sensors come in a linear array. Interesting implementations with 'V' and inverted 'V' shaped arrays have also been made. Most of these shapes have been custom made, and are not available easily.These perform better in some advanced tracks, but we'll stick to the in-line configuration here in this guide. Search up on this in case you decide to upgrade your robot later!

Analog and Digital Arrays

There are two types of arrays based on the kind of voltage output, analog and digital.

 A digital voltage array gives out only 2 values. Logically, this will just tell you if you are over a line or not. A digital array cannot accurately predict color difference in a track, for example if you have a black track with a red discontinuity on a white background, there is a chance that the white and red may give out the same reading. This means when your robot is actually over a red track, the sensors tell the robot that it is on white, which means it has strayed off the path, and it'll perform whatever action you tell it to do when it's strayed off the path.

A digital array is actually just an analog array with its output through a 'darlington pair' of transistors; which converts the output into what we can now call 'digital'. Many manufacturers don't specify this directly in the name, so you might have to search their datasheet to get what you need.

An analog array requires the extra task of converting to digital using an ADC in the microcontroller. We assume you're using an Arduino, so this will be no extra effort at all.

If you're using AVR's or PIC's, you'll need to code the microcontroller's input through the ADC, and probably define a buffer at the input and output. Help is available online!

Given a choice, an analog array will serve you better, and can be implemented in a good, tight control algorithm coming up in the next few sections!

Microcontrollers

We need a brain for the robot, which reads values from the sensors and tells the motors what to do. A microcontroller, or μC for short, does this exact job.

An Arduino is one of the simplest and most easy to use microcontrollers out there.

We assume you are using the Arduino Duemilanovae, which should be directly compatible with Decimila and UNO. If you have any other version or clone, do ask for guidance on the Arduino forums.

Arduino comes with an IDE, which helps burn code onto the microcontroller from your PC. It is open source and the website has links and extensive documentation on this.

An Arduino Duemilanovae has everything you need to get the job done, and lots of computing power to spare. This is not the only microcontroller you can use, PIC's and AVR's are among the other choices, but this is the easiest to work with! Others may be more powerful or more efficient, but the brilliant brains behind the Arduino made it easy…and fun!

PID Control

PID is a control system algorithm, which predicts and minimizes errors. It stands for 'Proportional Integral and Differential Control'. It is a generic algorithm and has many implementations. Some have even written libraries (in Arduino) for it. We will just use a simple implementation, but you can possible improve it by using other's extensive work on this subject.

Why this control?

You would have seen line followers with jerky movements. A control algorithm aims to reduce these, so your robot can travel smoother and faster along the line!

Why only PID?

You can use other digital control algorithms as well. We chose PID because it is efficient and simple to implement. How good a controller performs is inversely proportional to its computational complexity. Choose whatever suits your purpose.

Can I do without a PID control algorithm?

Yes, you definitely can, but it's a small addition and will be a bonus to your design! If you have the hardware to handle it, it'll be worth your while!

The Mechanical Design...

Flesh and Bone!

The first aspect of your design is your chassis. The chassis should be strong, light and small. Manufactured chassis are available, but it's best if you make your own customized one.

A good material is important – Epoxy or polymer boards are probably the best, but any other material should do as well.Avoid wood as it gets heavy and splinters when you drill holes or cut to size.

Sizing it up – Most competitions specify the size to be under a limit. Keep this in mind if you wish to compete. Place all your components on your board, and cut out the extra bits out. Always cut the board a little larger than you estimated at first, you can always cut it precise once the entire robot is finished. You might be thankful for this later! Keep the size as small as possible! Popular shapes are the circle and square, but you needn't follow them.

Fixing onto the chassis – You'll have to fix components onto the board. One way is to glue them (not advised!), and another is to drill holes into the board and clamp the components using screws. The Arduino has convenient holes for this as well. The placement has to be precise, so be careful! Use of standard screw sizes is recommended!

How to choose a motor is covered in the next section, here only the mechanical aspects are covered. You'll have to get clamps for the motors to fix them onto your chassis. The shaft size should match the inner hole of the wheel (duh!). Some come with a premade hole in the shaft for fixing. They are worth it!
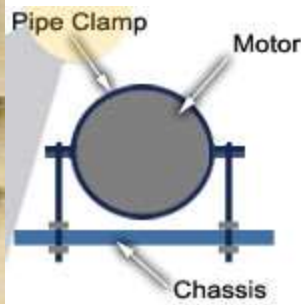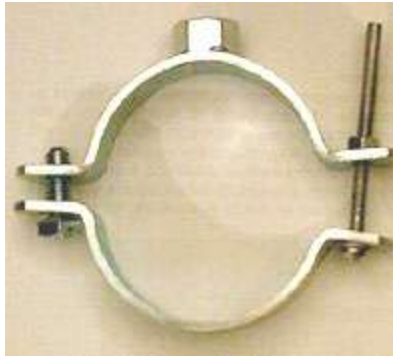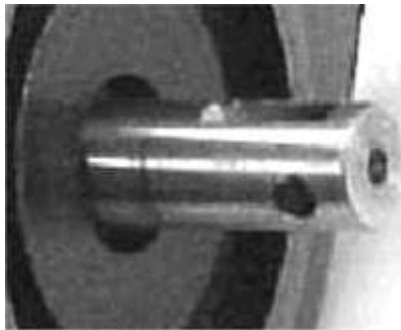
Pipe Clamps

You can also use rubber bands, or simple plastic fasteners the tie your bags with in supermarkets. Keep in mind some are temporary and some are permanent, so choose the best one for your purpose.

Plastic Fastener


A hole in the shaft…super convenient!

The Wheels

Wheel selection is very important. You can get them in any hobby robotics or toy store. Many online retailers sell them as well. The height of the center of the wheel should align with the motor shaft. It should have some degree of friction, which depends on how the surface of the arena is made. Some come with a readymade bushing so can be conveniently screwed in place. Look out for these! How you place your wheels and how many depend on your mechanical design.

If you have a wheel that's a little too small, you can always paste surgical tape (the cloth ones) or rubber bands around it to increase diameter and improve friction. This is also useful as a last minute addition, so keep some with you when you take your bot to battle!

The Drive System

The easiest and probably the best drive to use is called a differential drive. A lot has been written about the number of wheels and motors, and it is pointless replicating it here. Choose what seems best to you, and modify the rest of the design and code accordingly.

- Our code works if you're using something like this…
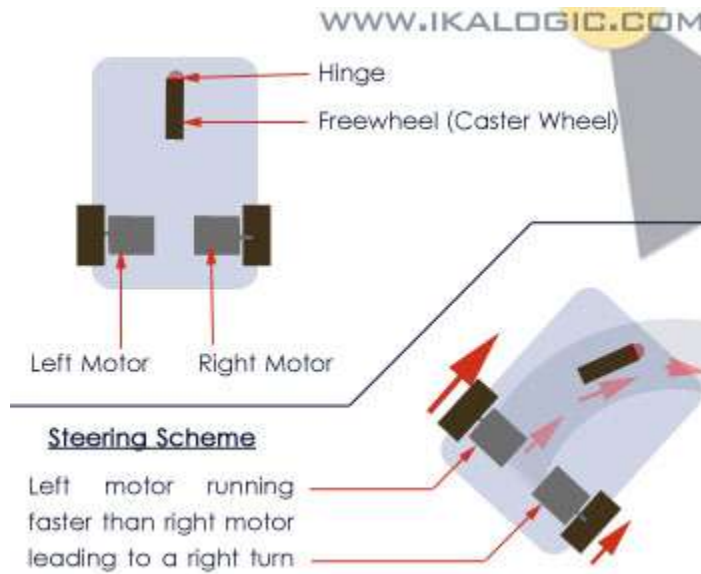
It's one of the easiest to implement and design, though other designs (like a tricycle drive) may perform better in a few aspects. Spend time on this area and make a careful decision!

Motor Selection

There are three types of motors used in small robots, and each has its own advantages and disadvantages. For what each motor does and how it works, research online. Here we only help you decide which one is best for your robot.

DC motors

They're cheap, and available in multiple variations of speed and torque. Some even come with gear sets, so you can customize it to best suit your robot. The disadvantage is you have no internal feedback control. That means you have no idea of the speed of the motor.

A word of caution here, the speed mentioned in the motor stats represents what is called the "no-load" speed of the motor. The motor will never run at that speed when you fix a chassis and all your components onto it.

A lot has been written about this, but I'll put in a few words because it is very critical.

- Pick a motor with a low voltage rating. This means, just for example, you should pick a 6V motor over a 24V motor. Motors with lower ratings are generally lighter.
- Some special motors with aluminum parts are available in a few places. They claim to be much lighter than the standard ones. I've never used them, so read up on the web before you buy them if you decide to.
- Some come with a complete gear assembly, meaning you can change the gear ratios and the torque of a motor. This can come in handy if you find yourself stuck in a rare situation where you're non geared DC motor has insufficient torque or speed.

The disadvantage mentioned above, means you have no way to "map" or remember the arena. This is not critical for a plain line following robot, but if you need to map the arena and you're using DC motors, you'll need an additional something like *optosensors*.

The code section assumes you're using DC motors. You'll need to modify the code if you want to use another kind of motor.

Stepper Motors

Stepper motors have the advantage of precise angular movement. You can tell a motor to rotate by, say, 45 degrees. Then knowing the diameter of the wheel, you can easily calculate the distance your bot has moved! All you need to do is to remember the signals you gave to the motor, and your track will be mapped.

The disadvantage of stepper motors are

- Cost – They're more expensive than simple DC motors
- Uneven movement – Some motors do not move smoothly, unlike DC motors which have a constant rotational speed.
- Torque/speed – A stepper motor which has high torque and speed ratings is hard to get, and will be expensive if available. You might suddenly find out the stepper does not have enough to power your robot!

Servo Motors

Servo motors are simply DC motors with feedback.

Servo's can be used to map arenas, and they also come in high torque/speed ratings. They've overcome the defects of DC motors and Steppers, but still remain expensive! It's rare if they're as easy to use as DC motors, like come with a shaft hole to mount a wheel! They're also more difficult to control!

A Practical Perspective...

There are some online stores which sell a complete drive assembly, including geared motors and wheels. They're convenient to use since they've been made to be compatible, so you don't have to worry about whether they fit each other!

There are contradicting opinions about what speed and torque is ideal. True they have an inverse relationship, but for robotics you can add gears to your motor and make them "high speed and high torque" as well. This is an important aspect, so read up on speed and torque if you need to. A bit of advice –High speed is pointless without torque, as the motor will never have enough time to accelerate to its maximum speed. High torque without much speed is equally pointless!

We'll be using a concept called PWM (Pulse Width Modulation) to drive our robot using the PID control algorithm. PWM is a technique where you control the speed of a motor by controlling how long current is supplied to it. If you power a motor for half the time, it runs at half the speed. Arduino does this with a single line of code. That's why it's easy!

Motor Drivers

Motor Drivers are circuits which allow control of your motors. You need them because you can't power a motor with just a microcontroller's supply. Motor Drivers are available in IC form, implemented in the form of an "*H-Bridge*". They allow you to switch on and off a motor using an output from a microcontroller, and the best feature is they allow you to run the motor in both forward and reverse!

This means each motor has 2 dedicated inputs from a microcontroller, one for direction control and one as an ON-OFF switch. If you're using PWM (we are!) you need a pin capable of producing a PWM output to control the ON-OFF switch. The direction control can be connected to any other normal pin.

You need not require the programming of the motors in reverse. If so, remember not to leave the direction control pin 'floating', and connect it to ground or high depending on your direction.

Motor Drivers are available in both IC form as well as a 'module' or 'breakout board'. The breakout boards are handy, as the pins would have been extracted out for easy access. It's worth the little extra to keep the circuit on your bot clean! You can see in the above picture the inputs and outputs have neat screw holders too!

Motor Drivers get extremely hot sometimes, and proper dissipation is necessary. Notice the large 'heat sink' in the module above.

Another important factor many overlook is the maximum current the motor driver IC allows to pass through it. It should be high if we need more current to drive our motors. This physical limitation of the IC can also result in the slow movement of your bot. Remember to allow for ample current in your

motors, and select a module/IC which has a motor driver capable of handling that current! Just for example, a commonly used IC is l293D, which allows for 500mA of current for each motor. The better option would be its bigger brother the L293DNE, which allows double that.

Power Supply

How you power your robot is important too! You have one of two choices

- Internal or on-board Supply
- External Supply

There is no fixed stipulation (unless specified for a competition) of which one you should use, just a matter of preference and resources available.

The voltage should be equal to the rating of the motor (which is assumed to be the highest in your circuit) and should be stepped down to power the other circuit components.

Internal or on-board supply requires batteries, which have to be rechargeable because you'll run through alkaline batteries faster than you know it. If you get rechargeable ones, you'll need a charger too. The current capacity, the weight and the rating of the battery also has to be accounted for. If you're using alkaline batteries you can make a pack out of them by connecting them in series and parallel. New technologies like Lithium Polymer and old ones like Nickel Metal Hydride are easily available, though may seem a little expensive. Remember PWM will make the motors draw currents in bursts of a few milliseconds apart. Some battery technologies cannot handle drawing so much current in such little time. This 'abuse' can even damage your battery! Consider all these factors in designing your on-board power system.
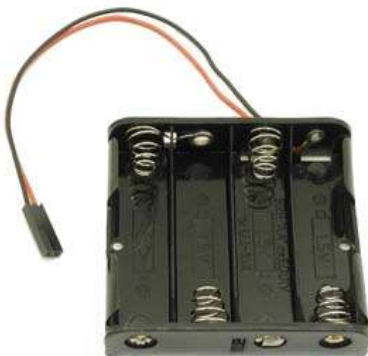
Consider buying or salvaging a battery holder from a gadget like an old toy car. They are helpful and neat! Check for sizes and shapes though, as 9v batteries are sometimes more useful than 'AA' sized ones and may have higher current ratings.

A voltage regulator IC connected to a 9v battery would look something like the diagram below.
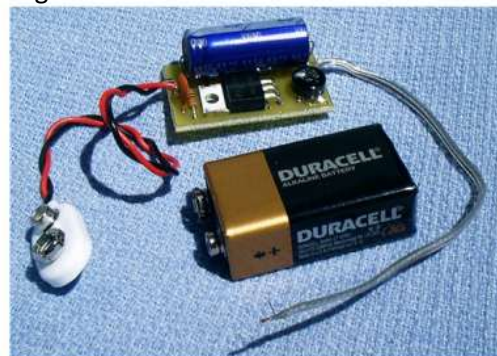
An External supply will consist of an AC to DC adapter. These come in various current and voltage ratings, and some even contain a variable regulator. You'll need to make a long cable to connect this to your robot, which can be cumbersome at times. You'll obviously need a regulator on board because different components run at different voltages.

A word of caution, do be very careful with voltages in your circuit. You could damage many components with excessive voltage supply! A Digital Multi-Meter is a handy instrument to help you in this. They are many excellent YouTube video tutorials on how to use them.

The Code!

All the hardware in the world is pointless without a good algorithm to put it to use. In the next section we'll learn how to build the code necessary to run a line follower. You should have completed all mechanical aspects by now!

Developing the PID algorithm

The PID algorithm uses three constants, Kp, Ki and Kd to function. They are shorthand notations for proportionality, integral and differential constants respectively. These three constants have to be set by you after testing, and define how good your control works.

Now let's look at how to develop a simple PID control algorithm.

Use the Arduino function *analogread()* to retrieve sensor values. You'll need an array to store these variables. Use Serial.print() to display these sensor values and observe them through the serial monitor in the Arduino IDE.

If you're using five sensors connected at analog pins 0-4, your code would look somewhat like this:

```
void setup()
{ Serial.begin(9600);                 //Necessary to set up Serial port
}


void loop()
{ Serial.print(analogRead(0));
Serial.print(' ');
Serial.print(analogRead(1));
Serial.print(' ');
Serial.print(analogRead(2));
Serial.print(' ');
Serial.print(analogRead(3));
Serial.print(' ');
Serial.print(analogRead(4));
Serial.println(' ');
delay(1500);                 //Set the number to change frequency of readings.

}
```

Burn this code onto the Arduino and place your bot over a sample track you've made. Move it across the line and observe the sensor readings. The reading will be a number between 0 and 1023. Find out what reading comes up for white and black, and any other colors you may want.

Important – The sensors connected from analog pins 0-4 should be in order from left to right. If you're using 5 sensors, the leftmost should be connected to pin 0, the middle one connected to pin 2, and the rightmost one to pin 4.

Now let's find the 'set point'. It is the position the robot is stable in, in our case in dead center of the line. This is the position the control algorithm strives to achieve. Familiarize yourself with the Arduino language!

We use two new quantities in our algorithm, an average of the sensor readings and sum of the sensor readings. You'll understand why we need these two when you move a little further in the algorithm development.

The below code displays the sensor average, sensor sum and position via the serial port. Use the serial monitor to view the readings.

```
longsensors_average;

intsensors_sum;

int position;

long sensors[] = {0, 0, 0, 0, 0};                 // Array used to store 5 readings for 5
sensors.

void setup()

{ Serial.begin(9600);

}

void loop()

{ sensors_average = 0;

sensors_sum = 0;


for (int i = 0; i < 5; i++)

   {sensors[i] = analogRead(i);

sensors_average += sensors[i] * i * 1000;   //Calculating the weighted mean

sensors_sum += int(sensors[i]);}           //Calculating sum of sensor readings

position = int(sensors_average / sensors_sum);

Serial.print(sensors_average);

Serial.print(' ');

Serial.print(sensors_sum);

Serial.print(' ');

Serial.print(position);

Serial.println();

delay(2000);

  }
```

I hope you have enough knowledge of mathematics and kinematics to understand the calculations of the weighted mean and position! It's not something I can explain really...

Now on to finding the set point, place your bot on the dead center of the line. The position or the third column of the readings above will give you your '*set point*'. Note it down separately. If you have time, verify the set point before your final run, more on this will come up in the tuning section.

Read up wiki's page on PID control before you continue. (I really mean it!)

Now we'll build the complete PID algorithm.

We start with calculation of the sensor sum and average similar to the above code. Note the next few segments of code are not complete. You'll have to finish them yourself!

```
sensors_average = 0;

sensors_sum = 0;

for (int i = 0; i < 5; i++)

  {

sensors[i] = analogRead(i);

sensors_average += sensors[i] * i * 1000;    //Calculating the weighted mean of the sensor
readings

sensors_sum += int(sensors[i]);              //Calculating sum of sensor readings

  }


voidpid_calc()

{   position = int(sensors_average / sensors_sum);

proportional = position - set_point;       // Replace set_point by your set point

integral = integral + proportional;

derivative = proportional - last_proportional;

last_proportional = proportional;


error_value = int(proportional * Kp + integral * Ki + derivative * Kd);

}
```

The above formula for calculation of error value is the functional definition of PID control. Notice you have to define the values of Kp, Ki and Kd in the code somewhere. After calculating the value of error, we need to tell the motor to move such that the error is minimized.

```
voidcalc_turn()

{   //Restricting the error value between +256.

if (error_value< -256)

   {

error_value = -256;

   }



if (error_value> 256)

   {

error_value = 256;

   }



   // If error_value is less than zero calculate right turn speed values

if (error_value< 0)

   {

right_speed = max_speed + error_value;

left_speed = max_speed;

   }

   // Iferror_value is greater than zero calculate left turn values

else

   {

right_speed = max_speed;

left_speed = max_speed - error_value;

   }

}
```

The above code snippet assumes you're using the differential drive system, where you execute a left turn if you reduce the speed of your left motor and a right turn if you reduce the speed of the right motor. We use a value max_speed that has to be defined by you right in the beginning to control the

speed of the motor. The maximum value of this is 256, which corresponds to the maximum output of the 8 bit DAC converter on the Arduino.

Now we have only one job left to do, to run the motors! So let's define a function which does exactly that. 'motor_right' and 'motor_left' are the pin numbers at which your motors are connected via the motor driver. Remember to use PWM pins!

```
voidmotor_drive(intright_speed, intleft_speed)

{     // Drive motors according to the calculated values for a turn

analogWrite(motor_right, right_speed);

analogWrite(motor_left, left_speed);

delay(50);          // Optional

}
```

IMPORTANT – If you're motors don't run at the same speed, meaning it veers to a side even with both motors get the same power, add a line of code to correct it. This is a rare case, but it may happen nevertheless!

Say your left motor moves faster than your right motor, add this before the analogWrite().

```
Left_speed = left_speed - 20
```

The '20' is a random number, and you should set it depending on your motors.Just set the values and do a test run if its still not approximately same add subtract accordingly till you have a more or less syncd motors(i.e they run at approx. the same speed).

Put the functions together, and use this statement in the loop section.

```
void loop()

{ sensors_read();   //Reads sensor values and computes sensor sum and weighted average

pid_calc();        //Calculates position[set point] and computes Kp, Ki and Kd

calc_turn();       //Computes the error to be corrected
```

```
motor_drive(right_speed, left_speed);  //Sends PWM signals to the motors

    }
```

That's it! You've built your PID algorithm for your line follower! You've yet to complete the code with the definitions of all the variables and the setup() segment.

The Next Steps…

Tuning

The most important parts of your algorithm are the three PID constants, Kp, Ki and Kd. They control the calculation of error and therefore affect the speed of the motors. PID is widely used industrially, and there are many techniques to tune PID. Here we'll use the trial and error method, though painstaking, is the easiest to use.

We need to manually set the values of the PID constants, so double check your robot and make sure everything is working and your batteries are charged. Your bot has a lot of track to cover!

1. Set all three constants to zero. Run the robot and see how it handles.
2. Vary the values of Kp, Ki and Kd in that order, one at a time and test your robot.
3. Do step 2 over and over again to get your bot perfectly tuned. (duh!)

This also involves burning the code into the Arduino many, many times! I've tried to send values to the Arduino while the bot is running via the serial port. I'd even asked for help in the forums, but I could not implement it in the short time I had. If you do manage a simpler method please let me know!

Fast tuning is important because each track may require different PID constants, and to tune it differently each time is lengthy. You may not even get the time needed if you're competing in a competition. What I finally managed to implement was a push-button entry system and a LCD display to help faster tuning.

Here's the segment of code responsible for this. This runs only once in the beginning because I called the function in the setup(). Pressing the reset button on the Arduino will call this again. Read up the Arduino tutorial on LCD display integration first. This uses some functions for display manipulations you need to understand.

The LCD will need many digital output pins. You can use any of the ones, leaving 2 PWM pins for motor control and 2 digital pins for direction control (if you're using it).

```
#include <LiquidCrystal.h>    // Built in library for LCD operation

void setup()

{ lcd.begin(16, 2);           // Set cursor to the bottom line of the LCD display

lcd.print("WHADDUP???");

delay(3000);

  // Setup input pushbutton pins

pinMode(select_switch, INPUT);   // replace the first parameters by the pins you're using

pinMode(up_switch, INPUT);

pinMode(down_switch, INPUT);


push_button_set();  // calling the function to set values of constants

    }



void loop()

{ if (start==true)    // start is a flag which is set on completion of input

   {sensors_read();         //Reads sensor values and computes sensor sum and weighted average

pid_calc();        //Calculates position[set point] and computes Kp,Ki and Kd

calc_turn();               //Computes the error to be corrected

motor_drive(right_speed, left_speed);

  }

else

motor_drive(0,0);  }


float set( float a)

{   delay(100);
```

```
lcd.setCursor(0,1);

lcd.print(a);

while(digitalRead(select_switch)==LOW)        //LCD print KI or KP or KD on one line
//dependant on select_count and display current value

    {   delay(100);

if(digitalRead(up_switch)==HIGH)                //If the button has been pressed

{ delay(100);

a*=10;               // increment by .1    you can change this if you need to.

a++;

a/=10;

lcd.setCursor(0,1);

lcd.print(a);

    }


if(digitalRead(down_switch)==HIGH)

{ delay(100);

a*=10;

        a--;

a/=10;

lcd.setCursor(0,1);

lcd.print(a);

    }

delay(70);// Delay to prevent debounce

    }

delay(100);

return(a);
```

```
}


voidpush_button_set()              //This is for changing the values of Ki,kp and kd

{

   ++select_count;

if (select_count==1)

{ lcd.clear();

lcd.print("Set Kp - ");

Kp=set(Kp);

       ++select_count;

     }

delay(100);

if(select_count==2)

{ lcd.clear();

lcd.print("Set Ki - ");

     Ki=set(Ki);

   ++select_count;

     }

delay(100);

if(select_count==3)

{ lcd.clear();

lcd.print("Set Kd - ");

Kd=set(Kd);

       ++select_count;

     }

delay(100);
```

```
if(select_count>3)

      {start=true;

lcd.clear();

lcd.print("BRING IT ON!!!");

delay(3000);

      }//LCD write push start button to start

}
```

I came across a funny problem when pins 0 and 1 dedicated to TX and RX on the Arduino were used. The Arduino Software would not allow me to burn code while these pins were connected to some hardware, and give me an error. This was solved when I removed the connections to these pins. I still don't know why this happened, but if you face a similar problem you can try this and hopefully you'll get past it!

Pushbutton switches are susceptible to this problem called de-bounce, where the output voltage spikes because of improper contact. You can eliminate this by introducing a delay, but remember you have to hold the switch in position for a longer time for the code to register this as a change. It's always good to add some delay in the code, but make sure it's not too less or too much to slow down process.

Sensor Thresholds

Another way you can achieve better control of your robot is to introduce a *threshold value*. Now say you measured the reading on black to be about 500. Accounting for a little noise, you can assume any reading in the range of 480 to 520 as black. The rest of the readings will be assumed as noise, and set to zero. A code to implement this can run like this:

```
voidsensors_read()

{ sensors_average = 0;

sensors_sum = 0;


for (int i = 0; i < 5; i++)

  {

sensors[i] = analogRead(i);


    // Readings less than threshold are filtered out for noise
```

```
if (sensors[i] < threshold)

sensors[i] = 0;

sensors_average += sensors[i] * i * 1000;   //Calculating the weighted mean of the sensor
readings

sensors_sum += int(sensors[i]);              //Calculating sum of sensor readings

}

}
```

The sensors[i]<threshold can be modified to sensors[i]<high_threshold&&sensors[i]>low_threshold.

This threshold concept may help if your sensors do not give consistent readings. If you implement this and find your bot does not work, vary the acceptable range a little. This is useful only if your robot moves on a track of the same color throughout.

Detecting Discontinuities

Sometimes you may need to count the number of color discontinuities in the track. For example, a black track with red breaks.

To count this, first note down the sensor readings when the bot is over red. This is an important calibration step. Add a little threshold if you wish, and use flags to detect a transition. The code can run like this :

```
if (sensors[i] <red_threshold+threshold_offset&& sensors[i]>red_threshold-threshold_offset)
//setting flags for counting

previous_red=true;

else

present_red=false;


if(previous_red== true &&present_red== false)   //When the bot has moved from red to black,
count increments.

{ red_count++; digitalWrite(13,HIGH);

delay(50);

    }

digitalWrite(13,LOW);

prev_red=pres_red;
```

}

Here we count the number of times the bot has moved from red to black. Counting the number of times the bot moves from black to red requires a simple manipulation. I've included an LED attached at pin 13 to flash whenever the transition is made. Once you test your robot and see this flash properly, you can add a count variable and display it on the LCD panel if you used one.

Thinking Ahead

There are still things to do after you've come this far. Things like right angle turns, discontinuities and intersections are still left to be done! You can also use the bi-directional motor control feature too. Maybe your current hardware is not enough to handle these, but that's for you to find out! You can re-orient the sensors and make a wall-follower or maze solver robot too!

We hope you've enjoyed learning, and found all this useful! This took much longer than we expected to write out, and we don't know if we'll be writing more stuff like this again.

Contact Us!

We hope you like this, and would really like your feedback! We'd love to know if you used some of this and scaled greater heights or even if you really liked it! It's a good feeling to know a hell lot of work has been appreciated! ;-)

 Do mail us at **barahir.bregor@gmail.com**

And yeah…do keep your questions to the forums, and don't bombard us with them! ;-)

Peace!!!