*1516 TCON * Motor Identification and Control*

November 24, 2015

Contents

Introduction

The lab objective is to control the EMG30 DC motor from an Arduino UNO board (center blue in figure) using an arduino motor shield equiped witha L298N driver (right blue).

Basic specifications

The EMG30 is a 12v motor equipped with an 2 channel encoder (of 360pulses/revolution), a 30:1 reduction gearbox, and a maximum speed of 217rpm without load (see for example `http://www.robot-electronics.co.uk/htm/emg30.htm`). The connector pin-out is

Color	Description
Purple	Hall sensor B out, open collector type
Blue	Hall sensor A out, open collector type
Green	Hall sensor GND
Brown	Hall sensor PWR (5V)
Red	Power MOTOR
Black	GND MOTOR

The Arduino UNO (`http://arduino.cc/en/Main/ArduinoBoardUno`) is a micro-controller board based on the ATmega328 (data-sheet). It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, an ICSP header, and a reset button. The pin-out is shown in fiigure **??**

The arduino motor shield has a L298N chip which is is a dual full-bridge driver that permits to drive two DC motors, controlling the speed and direction of each one independently (see `https://www.arduino.cc/en/Main/ArduinoMotorShieldR3` for an introduction). The connection pin-out is shown in figure 7

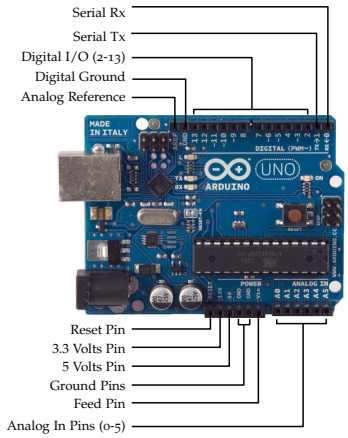

Figure 1: Arduino board pinout

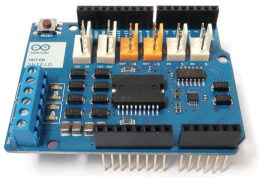

Figure 2: Arduino motor shield

Before we start... some technical aspects related to motor control

Encoders

Rotary encoders are often used to track the position of the motor shaft on permanent magnet brushless motors, which are commonly used on CNC machines, robots, and other industrial equipment. Incremental (Quadrature) encoders are used on Induction Motor type servomotors, but absolute encoders are used in Permanent Magnet Brushless Motors, where applicable.

Figure 3 reveals the nature of a hall-effect encoder. A permanent magnet rotates with the motor shaft. Two hall-effect sensors detect the polarity change and some electronics perform the right calculations to activate and deactivate the open collector output.

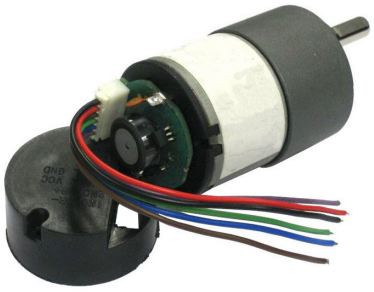

Figure 3: Hall-effect quadrature encoder, close to the one located in the EMG30 motor. It is possible to see 2 hall-effect sensors in a 120 degree shape.

The open collector encoder

An open collector is a common type of output found on many integrated circuits (IC), see figure 4.

Instead of outputting a signal of a specific voltage or current, the output signal is applied to the base of an internal NPN transistor whose collector is externalized (open) on a pin of the IC. The emitter of the transistor is connected internally to the ground pin. If the output device is a MOSFET the output is called open drain and it functions in a similar way.

Because the pull-up resistor is external and need not be connected to the chip supply voltage, a lower or higher voltage can be used instead. Open collector circuits are therefore sometimes used to interface different families of devices that have different operating voltage levels. The open-collector transistor can be rated to withstand a higher voltage than the chip supply voltage. Such devices are commonly used to drive devices such as Nixie tubes, and vacuum fluorescent displays, relays or motors which require higher operating voltages than the usual 5-volt logic supply.

Another advantage is that more than one open-collector output can connect to a single line. If all outputs attached to the line are in the high-impedance state, the pull-up resistor will hold the wire in a high voltage (logic 1) state. If one or more device outputs are in the logic 0 (ground) state, they will sink current and pull the line voltage toward ground. This wired logic connection has several uses.

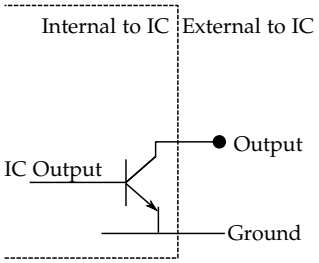

Figure 4: A simple schematic of an open collector of an integrated circuit (IC).

Pull-Up resistor

In electronic logic circuits, a pull-up resistor is a resistor connected between a signal conductor and a positive power supply voltage to

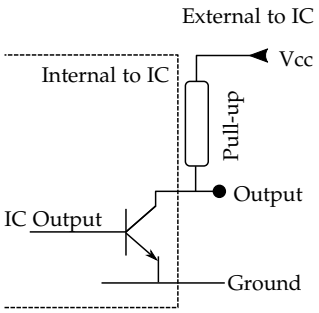

Figure 5: pull-up resistor in a open collector sensor

ensure that the signal will be a valid logic level if external devices are disconnected or high-impedance is introduced. They may also be used at the interface between two different types of logic devices, possibly operating at different logic levels and power supply voltages.

A pull-up resistor pulls the voltage of the signal it is connected to towards its voltage source level. When the other components associated with the signal are inactive, the voltage supplied by the pull up prevails and brings the signal up to a logical high level. When another component on the line goes active, it overrides the pull-up resistor. The pull-up resistor ensures that the wire is at a defined logic level even if no active devices are connected to it.

So when using arduino, the right voltage level for the digital inputs should be 5v, we have to set a pull-up resistor from the digital input to the 5v pin, but this may be done internally on the chip.

Arduino's internal pull-up

Arduino (Atmega) pins default to inputs, so they don't need to be explicitly declared as inputs with `pinMode()` when you're using them as inputs. Pins configured this way are said to be in a high-impedance state. Input pins make extremely small demands on the circuit that they are sampling, equivalent to a series resistor of 100 megohm in front of the pin. This means that it takes very little current to move the input pin from one state to another, and can make the pins useful for such tasks as implementing a capacitive touch sensor, reading an LED as a photodiode, or reading an analog sensor with a scheme such as RCTime.

This also means however, that pins configured as `pinMode(pin,INPUT)` with nothing connected to them, or with wires connected to them that are not connected to other circuits, will report seemingly random changes in pin state, picking up electrical noise from the environment, or capacitively coupling the state of a nearby pin.

There are 20K pullup resistors built into the Atmega chip that can be accessed from software. These built-in pullup resistors are accessed by setting the `pinMode()` as `INPUT_PULLUP`. This effectively inverts the behavior of the INPUT mode, where HIGH means the sensor is off, and LOW means the sensor is on.

When connecting a sensor to a pin configured with `INPUT_PULLUP`,the other end should be connected to ground. In the case of a simple switch, this causes the pin to read `HIGH`when the switch is open, and `LOW`when the switch is pressed.

The pullup resistors are controlled by the same registers (internal chip memory locations) that control whether a pin is `HIGH` or

LOW.Consequently, a pin that is configured to have pullup resistors turned on when the pin is an INPUT, will have the pin configured as HIGH if the pin is then switched to an OUTPUT with pinMode(). This works in the other direction as well, and an output pin that is left in a HIGH state will have the pullup resistors set if switched to an input with pinMode() [1].

Arduino code: Set pin as input with pull-up resistor

```
pinMode(pin, INPUT); // set pin to input
digitalWrite(pin, HIGH); // turn on pullup resistors
```

[1] Digital pin 13 is harder to use as a digital input than the other digital pins because it has an LED and resistor attached to it that's soldered to the board on most boards. If you enable its internal 20k pull-up resistor, it will hang at around 1.7V instead of the expected 5V because the onboard LED and series resistor pull the voltage level down, meaning it always returns LOW. If you must use pin 13 as a digital input, set its pinMode() to INPUT and use an external pull down resistor.

Moving the motor

To be able to move the motor we requiere power, but arduino provides 5v without a proper current intensity. To overcome this problem we use a power stage composed by a *H*-bridge.

H-bridges are available as integrated circuits, or can be built from discrete components.

The term *H*-bridge is derived from the typical graphical representation of such a circuit. An H bridge is built with four switches (solid-state or mechanical). When the switches S1 and S4 (according to figure 6) are closed (and S2 and S3 are open) a positive voltage will be applied across the motor. By opening S1 and S4 switches and closing S2 and S3 switches, this voltage is reversed, allowing reverse operation of the motor.

Using the nomenclature above, the switches S1 and S2 should never be closed at the same time, as this would cause a short circuit on the input voltage source. The same applies to the switches S3 and S4. This condition is known as shoot-through.

In our case the *H*-bridge is controlled by an integrated circuit located in the motor control shield, so we have to take no care about the switching sequence.

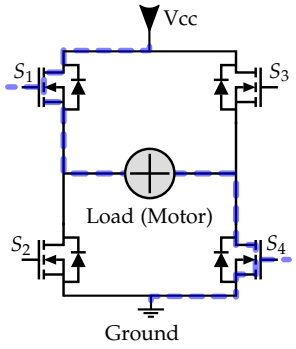

Figure 6: *H*-bridge working

The motor control shield

The Arduino Motor Shield is based on the L298 (http://www.st.com/web/en/catalog/sense_power/FM142/CL851/SC1790/SS1555/PF63147), which is a dual full-bridge driver designed to drive inductive loads such as relays, solenoids, DC and stepping motors. It lets you drive two DC motors with your Arduino board, controlling the speed and direction of each one independently. You can also measure the motor current absorption of each motor, among other features.

This shield has two separate channels, called A and B, that each

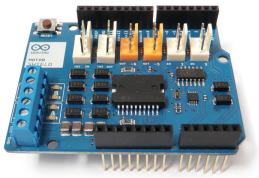

Figure 7: Arduino motor shield

use 4 of the Arduino pins to drive or sense the motor. In total there are 8 pins in use on this shield. You can use each channel separately to drive two DC motors or combine them to drive one bipolar stepper motor.

Function	Chanel A oins	Chanel B Pins
Direction	D12	D13
PWM	D3	D11
Brake	D9	D8
Current sensing	A0	A1

Figure 9 shows the right orientation and position of the shield on top of the arduino board

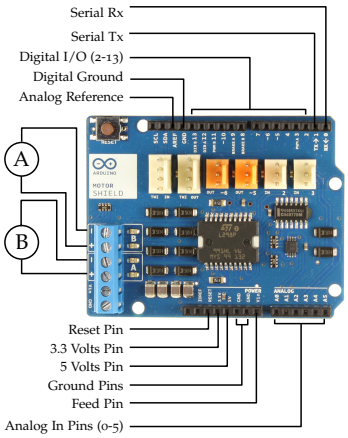

Figure 8: Motor Shield pionout

Connections

Motor-power To power the motor, the RED (power) and BLACK (ground) cables of the motor must be connected to the motorA pins of the driver.

Encoder-shield To measure speed we are going to connect only one of the two encoder channels, so channel A or channel B from the encoder should be connected to digital pin 2 in the shield

Encoder feed To allow the encoder to work properly we must provide power supply to it, this is accomplished connecting Vcc pin and Ground pin to the brown and green wires.

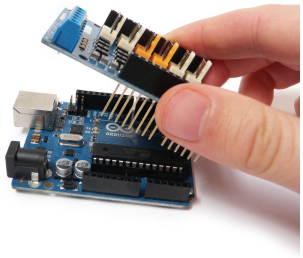

Figure 9: Right way to connect the shield to the arduino board

Figure 10 shows the way we connect the motor to the driver, which yelds a closed loop control.

Moving the motor for the first time

Given the previous connections, a basic program that moves the motor is

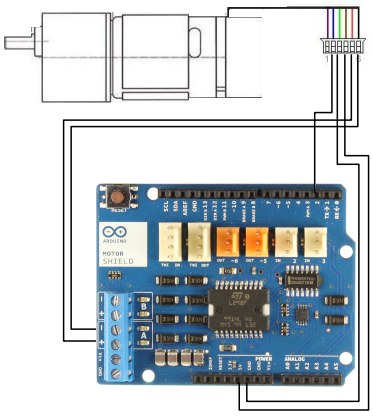

Figure 10: Connections between the motor and the driver

Arduino code: Basic motor code

```
int PWM=3;/*connected to Arduino's port 3(output pwm)*/
int IN2=2;/*Encoder reading*/
void setup()
{
      pinMode(PWM,OUTPUT);
      pinMode(IN2, INPUT);/* set pin to input*/
      digitalWrite(IN2, HIGH);/* turn on pullup resistors*/
}

void loop()
{
      analogWrite(PWM,150);
}
```

It is interesting to note that the PWM programming becomes extremely easy compared to other microprocessor-based architectures.

Reading the motor speed

Encoders measure motor's rotation speed. Encoders output are trains of pulses whose frequency depends on motor rotational velocity. The arduino board will power the encoder embedded in the motor (GREEN and BROWN cables) and will read the encoder outputs (PURPLE and BLUE cables).

A basic program that moves the motor and reads information coming from the sensor is given next.

Arduino code: Basic motor code

```
int PWM=3;/*connected to Arduino's port 3(output pwm)*/
int IN2=2;/*Encoder reading*/
int pulseBlength=0;//encoder pulse length in microseconds
void setup()
{
      pinMode(PWM,OUTPUT);
      pinMode(IN2, INPUT);/* set pin to input*/
      digitalWrite(IN2, HIGH);/* turn on pullup resistors*/
      analogWrite(PWM,150)/*Start the Motor*/
      Serial.begin(115200);
      Serial.println("Starting data acquisition...")
}

void loop()
{
      pulseBlength=pulseIn(IN2,HIGH);
      Serial.println(pulseBlength);
}
```

It also uses the serial line at 115200bps to printout the sensor information. This information can be displayed in the Serial Port Monitor available from the Arduino IDE, in the tools menu. After executing the code, the serial port monitor should display something like the data shown in figure 11

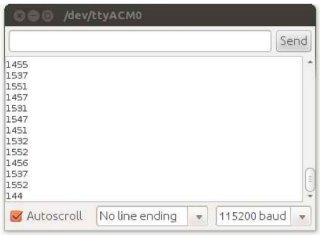

Figure 11: Serial port output with the motor speed

DC-Motor speed model

The electric equivalent circuit of the armature and the free-body diagram of the rotor are shown in the following figure **??**. For this example, we will assume that the input of the system is the voltage source (V) applied to the motor's armature, while the output is the rotational speed of the shaft $\frac{d\theta}{dt}$. The rotor and shaft are assumed to be rigid. We further assume a viscous friction model, that is, the friction torque is proportional to shaft angular velocity.

In general, the torque generated by a DC motor is proportional to the armature current and the strength of the magnetic field. In this example we will assume that the magnetic field is constant and, therefore, that the motor torque τ is proportional to only the armature current i by a constant factor K_τ as shown in the equation below. This is referred to as an armature-controlled motor.

$$\tau = K_\tau i$$

The back emf, e, is proportional to the angular velocity of the shaft

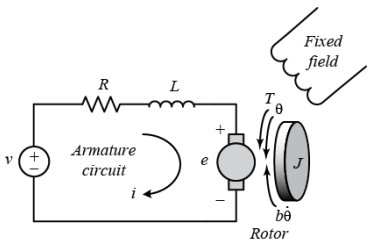

Figure 12: Phisical description of a DC-Motor

by a constant factor K_e.

$$e = K_e\dot{\theta}$$

In SI units, the motor torque and back emf constants are equal, that is, $K_t = K_e$; therefore, we will use K to represent both the motor torque constant and the back emf constant.

From the figure above, we can derive the following governing equations based on Newton's 2nd law and Kirchhoff's voltage law.

$$J\ddot{\theta} + b\dot{\theta} = K_i$$
$$L\frac{di}{dt} + Ri = V - K\dot{\theta}$$

In state-space form, the governing equations above can be expressed by choosing the rotational speed and electric current as the state variables. Again the armature voltage is treated as the input and the rotational speed is chosen as the output.

$$\frac{d}{dt}\begin{bmatrix} \dot{\theta} \\ \\ i \end{bmatrix} = \begin{bmatrix} -\frac{b}{J} & \frac{K}{J} \\ \\ -\frac{K}{L} & -\frac{R}{L} \end{bmatrix}\begin{bmatrix} \dot{\theta} \\ \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ \\ \frac{1}{L} \end{bmatrix} V$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix}\begin{bmatrix} \dot{\theta} \\ \\ i \end{bmatrix}$$

Parameter Estimation

Estimating the system parameters may become a hard problem, to overcome it you can use any toolbox from matlab, but in this case we will be using fminsearch function. The plan is to give an input to the system, square signal changing every 1 seconds, and to record the motor speed. Once we have this data we will create a function in matlab[2] to compute how far we are from the real model, this function takes the input, the parameters and computes an output, then it compares the real output with the estimated one to provide a positive number which is the distance to the real output. Next we provide this code:

[2] A function that we will name as distance

Matlab code: System response

```
function r=distance(params)
   % (J)     moment of inertia of the rotor
   % (b)     motor viscous friction constant
   % (K)     electromotive force constant  and  motor torque constant
   % (R)     electric resistance
```

```
% (L)     electric inductance
J=params(1); b=params(2); K=params(3);
R=params(4); L=params(5);
A=[-b/J K/J; -K/L R/L];
B=[0; 1/L];
C=[1,0];
D=0;
sys=ss(A,B,C,D)
%Now we discretize the systsem with the same
%rate the data was provided
h=0.02;
sysd=c2d(sys,h);
%And now we simulate the response
global u %system input, recorded with arduino
global t %timming of the input
global y %Real output recorded with arduino
[y_model,t]=lsim(sysd,u,t)
r=norm(y-y_model)
```

Now, to use this function you have to do a couple of things:

1. Load into the workspace the values of the real system output, real system input and its timings
2. Declare these variables as global
3. Call `fminsearch` function to estimate the parameters

Once you have the model is a matter of standard work to design the controller.

TODO list

1. check whether the given codes work: start by the BLINK, continue with the moving the motor and with the encoder reading, and finish with the kst osciloscope
2. check the speed sensor quality, think a way to improve the readings and implement it[3]
3. Check the static gain of the motor to see whether it is linear or not, graph speed against pwm
4. Find a new variable (fakePWM) such that the static gain against this variable is linear
5. Obtain the motor model
6. Design and implement a controller in state space to control the motor [4]

[3] Use interrupts to accomplish this point

[4] The sampling period is up to you

7. Design and implement a controller that works regardless the load of the motor
8. Design and implement a controller that follows a sinusoidal signal regardless the applied load to the motor

If there is time left:

1. Change motor to port B and attach both encoder channles to the arduino.
2. Make a position control of the Motor.

Interrupt code for Arduino

Arduino code: Set up interrupt on port 2

```
long m1,m2;
long up_times[3]={0,0,0};
int index=0;
int pin = 2;
//Interrupt for encoder
void int0()
{
    m1=micros();
    up_times[index]=m1-m2;
    index=(index+1)%3;
    m2=m1;
}
void setup() {
    pinMode(pin, INPUT);
    digitalWrite(pin,HIGH);
    attachInterrupt(digitalPinToInterrupt(pin), int0, RISING);
}
void loop() {
    float speed;
    long t_min=0;
    t_min=up_times[0]+up_times[1]+up_times[2];
    speed=60E6/t_min/30;
    Serial.println(speed);
    delay(100);
    /*TO DO*/
}
```

Increasing PWM and measuring speed

In order to model the static gain of the motor[5] we must plot the gain to any input to he system. To do so we apply an increasing PWM signal to the motor and we measure it's speed. Once we have this data we will be able to perform a plot of speed against PWM.

[5] This is a non constant value, the motor saturates at some speed, so as we give more pwm signal to the motor it accelerates less and less up to a point where it is unable to go faster. This brakes the linearity rule "twice the input twice the output"

Arduino code: Get speed for a range of PWM values

```
long m1,m2; //Store time in interrupts
float speed;//Sepeed calculation
long t_tot=0;//Mean value of interrupts
long up_times[3]={0,0,0};//Interrupts timing
int index=0;   //Intex for interrupts timing
int pin = 2;  // Encoder pin
int PWM = 3; //PWM pin
//Interrupt for encoder
void int0()
{
    m1=micros();
    up_times[index]=m1-m2;
    index=(index+1)%3;
    m2=m1;
}
void setup() {
    pinMode(pin, INPUT);
    digitalWrite(pin,HIGH);
    attachInterrupt(digitalPinToInterrupt(pin), int0, RISING);
    //Initialize serial and wait for port to open:
    Serial.begin(115200);
    while (!Serial) {
        ; // wait for serial port to connect. Needed for native USB
    }
}
void loop() {
    Serial.println("----------------------------------------")
    for (inti=40;i<255;i++)
    {
        analogWrite(PWM,i); //set pwm
        delay(2000); //whait for 2 seconds
                     //to allow the ,motor to stabilize
        t_min=up_times[0]+up_times[1]+up_times[2];
        speed=60E6/t_min/30; //Compute speed
        Serial.print(i)
```

```
        Serial.print(",")
        Serial.println(speed); //print speed
    }
}
```

Arduino Code: demo

```
void setup()
{
    //Initialize serial and wait for port to open:
    Serial.begin(115200);
    while (!Serial)
    {
        ; // wait for serial port to connect. Needed for native USB
    }
}
void loop()
{
    Serial.println()
    code
}
```