UNIVERSIDAD DE SONORA

DIVISIÓN DE INGENIERÍA
Departamento de Ingeniería Industrial

CONTROL DE MOVIMIENTO DE UNA GUÍA LINEAL

TESIS

Que para obtener el título de:

INGENIERO EN MECATRÓNICA

Presenta:

BRUNO MARTÍNEZ ESCARCEGA

HERMOSILLO, SONORA. **Abril 2016**

Universidad de Sonora

Repositorio Institucional UNISON

Agradecimientos

Quiero agradecer a todos mis maestros ya que ellos me enseñaron a valorar los estudios y a superarme cada día, al igual que agradezco a mi madre y amigos porque ellos estuvieron en los días más difíciles de mi vida como estudiante.

Índice

Capítulo 1

Introducción

1.1 Problemática a resolver

En todas las industrias es necesario el uso de motores para transportar materiales o productos o en otros tipos de sistemas con funciones más específicas como controlar brazos robóticos. Para esto se han diseñado numerosos circuitos que logran controlar con éxito motores de corriente directa, pero muchos de estos controles son genéricos y no permiten un marco de opciones amplio para realizar modificaciones a su desempeño. Muchos solo permiten variar una s*eñal PWM* o funcionan utilizando controladores *PID*, incluso hay circuitos integrados con esta función, pero solamente pueden generar una señal trapezoidal o triangular para controlar los motores de corriente directa y no generan perfiles de velocidad con aumentos o decrementos de aceleración suaves y variables con el tiempo [1,2,3,4], lo anterior genera muchas limitantes al momento de controlar motores de cd debido a que no se dispone de un marco amplio de opciones que permita modificar los esquemas de dichos controladores, esto no quiere decir que sean malas opciones; el caso más común, probablemente es controlar un motor de corriente continua mediante un circuito temporizador 555 [5], que funciona como un *PWM*, formando una opción bastante económica.

En este trabajo se diseñará un sistema de control de motores de corriente continua que permita crear perfiles de velocidad a partir de las necesidades del usuario, necesidades como velocidad, posición, aceleración, tiempo y empuje. Para lograr esto, será necesario diseñar una

interfaz gráfica para poder representar los perfiles del motor que sean de interés. La interfaz es uno de los puntos de mayor interés en el trabajo. La idea es contar con un sistema con que sea posible controlar el motor al mismo tiempo que poder observar detalladamente los estados transitorios y permanentes en el tiempo, o poder crear perfiles para observarlos posteriormente cuando empiece a funcionar el motor.

Al adquirir un control de motor es común tener que fabricar un circuito electrónico o instalar algún sistema físico que cumpla con los requerimientos necesarios para su funcionamiento, por eso es primordial diseñar además del programa un circuito electrónico que trabaje en conjunto con el software que se utilizará, para así lograr un sistema completo, software y hardware que permita lograr todas las funciones antes mencionadas, además de cumplir con los objetivos planteados en los objetivos específicos y generales.

1.2 Objetivos

1.2.1 Objetivo General

El objetivo principal es el diseño de un *servomecanismo lineal* por medio de perfiles de posición, velocidad, aceleración y empuje que pueda ser operado por un usuario desde una computadora mediante el uso de un programa de código abierto diseñado para este fin. Dicho programa deberá contar con indicadores que muestren todos los datos concernientes al *servomecanismo*.

1.2.2 Objetivos Específicos

1. Análisis de las ecuaciones de movimiento para el control de variables mecánicas.
2. Implementación de las ecuaciones de movimiento por medio de *prosessing*.
3. Diseño de una interfaz gráfica en *processing*.
4. Diseño del control de movimiento en la plataforma *arduino*.

1.3 Hipótesis

Utilizando las plataformas de desarrollo *arduino* y *processing*, es posible diseñar un sistema de control de movimiento para un servomecanismo lineal.

1.4 Delimitaciones

El motor y el *encoder* utilizado en este trabajo no fueron adquiridos especialmente para el presente proyecto, por lo que es esperado que se presenten fallas mínimas o falta de precisión en algunos procesos. El programa diseñado solo será eficiente al trabajar con el modelo utilizado de motor de corriente continua, si se deseara conectar un modelos diferente será necesario adaptar los parámetros del programa a este nuevo motor.

Se llevarán a cabo pruebas físicas con un motor de *24 volts* de corriente directa y una guía lineal de tipo industrial donde se conectará al motor del servomecanismo y se realizarán pruebas virtuales en el programa generando perfiles de velocidad. Uno de los puntos de interés es la implementación de un sistema de ecuaciones capaz de generar *curvas de aceleración* que serán necesarias para construir los perfiles deseados.

Capítulo 2

Marco de referencia

2.1 Antecedentes

2.1.1 Estimación de velocidad

El problema más común en esta clase de sistemas es medir correctamente los cambios que se generan en el motor, ya sea de velocidad o posición, para esto se puede encontrar una gran cantidad de documentos e información, artículos especializados en la lectura de *encoders* para sistemas similares al aquí presentado. Sin embargo resulta muy fácil cometer errores en este ámbito, provocando malas lecturas y cálculos erróneos, lo que ocasionaría que el sistema sea inútil y poco fiable.

El primer paso para determinar la velocidad de un *encoder* es conocer su posición, esto se logra mediante 2 trenes de pulsos que puede emitir un *encoder incremental*, dichos trenes son llamados A y B, como se pueden apreciar en la figura 2.1. En conjunto, estos 2 canales permiten conocer la dirección en que gira el motor, esto último corresponde a un *encoder incremental* debido a que solo se manejan 2 canales.

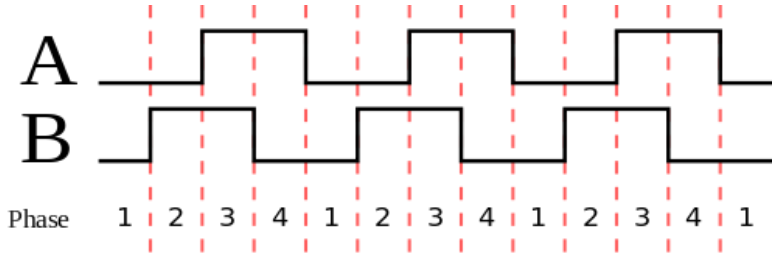

Figura 2.1. Trenes de pulso de un *encoder incremental, canal A y canal B.*

En este punto es donde existen 2 métodos diferentes que se basan en la misma ecuación para estimar velocidades. El primero es medir los cambios de posición obteniendo ΔX y calcular la velocidad en intervalos de tiempo fijos. Este método es mejor para velocidades moderadas. El segundo método funciona mejor con velocidades bajas, y consiste en establecer como base un diferencial de posición y medir el tiempo, en otras palabras obtener Δt [6].

2.1.2 Perfiles de velocidad.

Los perfiles de velocidad sirven para trazar la velocidad que deberá tomar un motor en un determinado tiempo, para propósitos del presente trabajo, dicho tiempo será establecido por las necesidades de algún usuario. En principio existen 2 perfiles básicos que se pueden tomar en cuenta, estos son el *perfil de velocidad triangular* y *perfil de velocidad trapezoidal* [7].

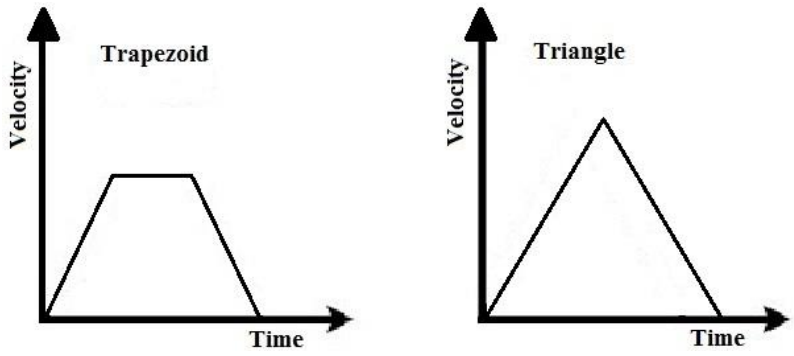

Figura 2.2. Perfil de velocidad trapezoidal y triangular.

Si se analiza un perfil de velocidad más elaborado que los vistos en la figura 2.2 se observan algunas diferencias, principalmente en las aceleraciones que no son constantes todo el tiempo, al analizar la figura 2.3 se observa esto. Un ejemplo es el primer segmento de *T0* y *T1*, conformado por varias partes como aceleración, desaceleración y aceleración constante. Primero el perfil alcanza su velocidad máxima en *T1*, para luego mantenerse así hasta *T2*, donde empezará a disminuir su aceleración hasta llegar al reposo en *T3*.

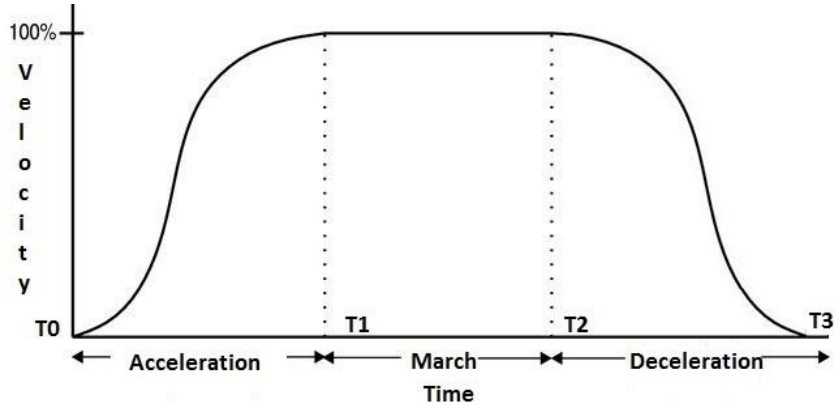

Figura 2.3. Perfil de velocidad.

Para este trabajo se utilizan ecuaciones para construir las partes de las que estará formado un perfil de velocidad. Observando la figura 2.4 es posible saber que el movimiento de aceleración está formado por tres partes donde la aceleración aumenta en el segmento "*a*", la aceleración es constante en el segmento "*b*" y finalmente la aceleración disminuye en el segmento "*c*", todo esto para formar la aceleración del perfil donde se busca alcanzar el punto máximo de velocidad.

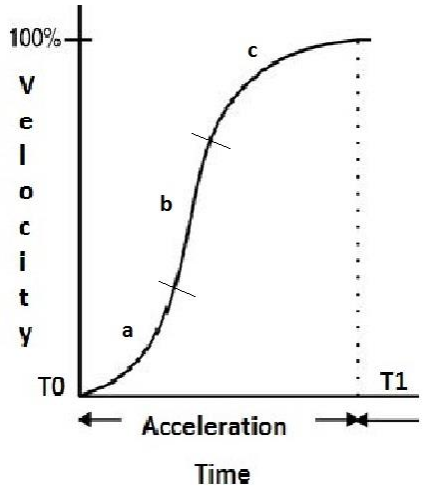

Figura 2.4. Curva de aceleración del perfil de velocidad de la figura 2.3.

En el momento en que se busca disminuir la velocidad es necesario iniciar un proceso de desaceleración, el cual también está formado por varios segmentos que se comportan de acuerdo a las mismas ecuaciones que cuando el perfil acelera, pero ahora utilizando aceleración negativa, en otras palabras desacelerando.

2.1.2.1 Perfil de velocidad triangular

El *perfil de velocidad triangular* tiene como característica que aumenta su velocidad con una aceleración constante en un determinado tiempo, y cuando alcanza su velocidad máxima (*Vmax*) comienza un decremento de velocidad igualmente con aceleración constante hasta llegar a una velocidad de cero.

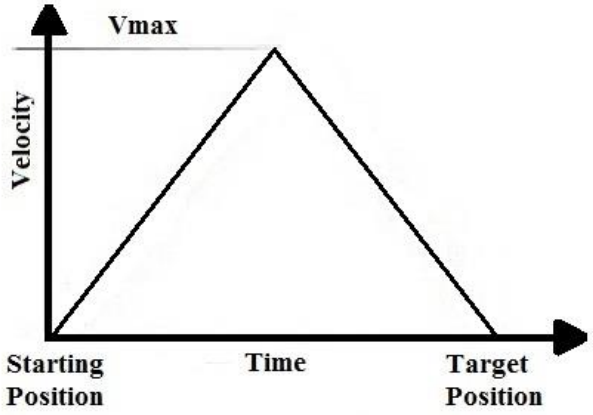

Figura 2.5. Perfil de velocidad triangular.

2.1.2.2 Perfil de velocidad trapezoidal

El *perfil de velocidad trapezoidal* es similar al *perfil de velocidad triangular*, pero cuando llega a su punto de velocidad máxima esta se mantienen constante durante un tiempo, al cumplir ese tiempo empieza un decremento de velocidad constante hasta descender a una velocidad cero o a la velocidad deseada.

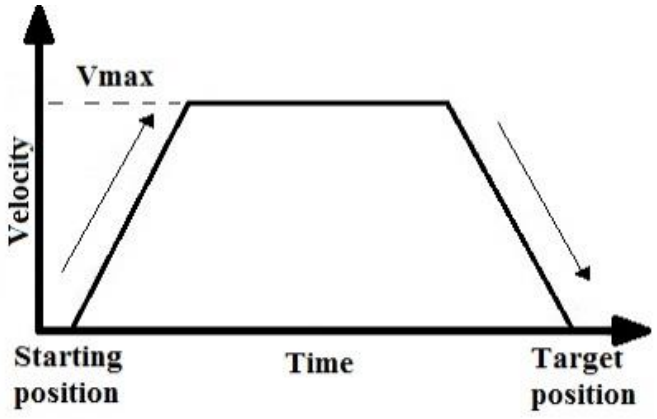

Figura 2.6. Perfil de velocidad trapezoidal.

Estos perfiles de velocidad se pueden manejar en intervalos de tiempo que denotan el estado de la velocidad en ese momento, por ejemplo en la figura 2.7, se puede apreciar como el tiempo 1 (*T1*) corresponde al periodo donde la velocidad se incrementa hasta alcanzar la velocidad deseada. Aquí se aprecia como en el tiempo 2 (*T2*) es el periodo donde la velocidad se mantiene constante, cabe señalar que el perfil de velocidad puede carecer de este periodo dando como resultado un perfil de velocidad triangular, al tener dicho periodo de tiempo el resultado es un perfil de velocidad trapezoidal. Al final se encuentra el tiempo 3 (*T3*), donde la velocidad empieza a decrecer para llegar a una velocidad de cero.

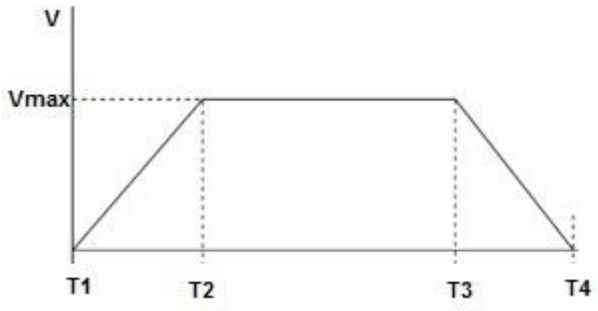

Figura 2.7. Tiempos del perfil de velocidad trapezoidal.

2.1.2.3 *Jerk* en los perfiles de velocidad

El *Jerk* es la suavidad que tendrá la curva de aceleración de un perfil de velocidad, el *jerk* no es más que el tiempo de aceleración o desaceleración que hay en el aumento de velocidad que el perfil usa para alcanzar su velocidad deseada.

El jerk propone eliminar el empuje mecánico cuando cambia la velocidad y esto también puede ser usado para optimizar la respuesta, reducir el derrame cuando se transportan líquidos y prevenir los movimientos bruscos al transportar cajas en una banda transportadora [7].

El *Jerk* es definido como el porcentaje de aceleración o desaceleración que existe en el aumento de velocidad hasta alcanzar su máximo, este porcentaje está basado en el tiempo que dura cada fase de la curva de aceleración (incremento de aceleración, aceleración constante, decremento de aceleración). En un *jerk* del 50% los tiempos de la etapa de incremento y decremento de aceleración son iguales y en conjunto son la mitad del tiempo total de la curva de aceleración, con un *jerk* de 0% no hay tiempo de aceleración incremental y decremento de aceleración, solo hay tiempo de aceleración constante, en otras palabras la aceleración se mantiene constante hasta que se alcanza la velocidad máxima, pero con un *jerk* del 100% no hay aceleración constante, la mitad del tiempo durante la curva hay un incremento de aceleración y la segunda mitad del tiempo hay un decremento de la aceleración hasta alcanzar la velocidad máxima.

2.1.3 Fundamentos del sistema de ecuaciones.

2.1.3.1 Ecuación de la pendiente

Una pregunta básica en el presente trabajo es, si es posible diseñar un sistema de ecuaciones que permita crear *perfiles de velocidad* a partir de variables que serán introducidas por un usuario. Dicho sistema de ecuaciones será utilizado en la interfaz gráfica aquí presentada para generar una curva de aceleración que permita tomar en cuenta variables tales como velocidad, posición, tiempo, aceleración y empuje. Estas ecuaciones deberán estar en función del tiempo y deben permitir relacionar las variables anteriormente mencionadas entre sí.

Para lograr lo anterior se parte de la ecuación de la pendiente en su forma general (2.1), [8].

$$m = \frac{y1-y}{x1-x} \tag{2.1}$$

en donde:

m es la pendiente de la recta.
y1 – y es la diferencia entre los puntos del eje Y.
x1 – x es la diferencia entre los puntos del eje X.

A demás también se utilizó la ecuación de la pendiente en su forma punto pendiente (2.2) [8].

$$y = mx + b \tag{2.2}$$

en donde:

y es las coordenadas del punto situado sobre el eje Y.
b es el punto donde la recta se intercepta con el eje Y.

2.1.3.2 Ecuación de aceleración.

Una vez que las ecuaciones de la pendiente sean integradas, se tendrá un conjunto de ecuaciones que podrán construir cualquier *curva de aceleración*. Dichas ecuaciones necesitan valores de aceleración y tiempo para poder trabajar, por tanto para incluir otros valores como posición o velocidad será necesario apoyarse de fórmulas y procedimientos con los que es posible determinar la aceleración que el sistema necesita para cumplir con los parámetros asignados. Una de estas ecuaciones es la fórmula para la aceleración (2.3), la cual requiere de una *velocidad final* y el tiempo en que el motor se mantendrá acelerando [7].

$$a = \frac{Vf}{\frac{t1}{2} + t2 + \frac{t3}{2}} \tag{2.3}$$

en donde:

a es la aceleración máxima del motor.
Vf es la velocidad máxima del sistema.
t1, t2 y t3 son los tiempos que corresponden a cada sección de la *curva de aceleracion*.

2.1.4 Processing y arduino.

2.1.4.1 Processing

Processing es un lenguaje de programación y entorno de desarrollo de código abierto, inicialmente creado para servir como medio para la enseñanza de los fundamentos de programación dentro de un contexto visual, *processing* evolucionó hasta convertirse en una herramienta de desarrollo para los profesionales. Hoy en día, hay decenas de miles de estudiantes, artistas, diseñadores, investigadores y aficionados que utilizan *processing* para el aprendizaje, la creación de prototipos y producción. [9]

Processing es el software perfecto para diseñar una interfaz gráfica que muestre los resultados de la evaluación del sistema de ecuaciones que se requiere para crear los perfiles de

velocidad. La interfaz gráfica diseñada puede presentar de manera visual todos los aspectos que son necesarios para entender la naturaleza del sistema.

2.1.4.2 Arduino

Arduino es una plataforma de hardware y software libre. *Arduino* es una placa con un *microcontrolador* de la familia *Atmega,* y con una serie de entradas y salidas analógicas y digitales, además contiene un entorno de desarrollo que facilita el uso de la electrónica. Debido a su sencillez cada vez es más utilizado en ambientes laborales [10].

Figura 2.8. Imagen de un *arduino UNO.*

El software consiste en un entorno de desarrollo que es capaz de implementar *processing*. En términos de este trabajo es de interés conocer especialmente la función para trabajar con *PWM* de *arduino,* función que cuenta con 6 salidas en la placa.

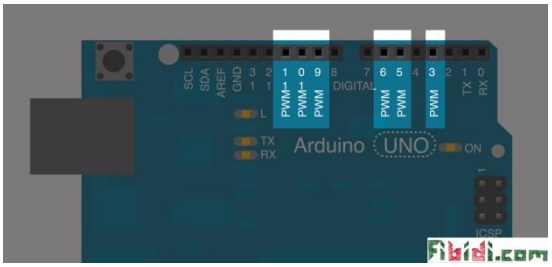

Figura 2.9. Salidas *PWM* de *arduino*

2.1.4.3 Comunicación entre *processing* y *arduino*

En algunos programas como *labview, C++* o *java* un método óptimo para la comunicación es la *comunicación serial,* donde la información es mandada en forma de un tren de datos. En el caso de *processing* en conjunto con *arduino* se utiliza este tipo de comunicación debido a la sencillez que presenta.

Para que su función se extienda a la *comunicación serial* durante el tiempo de ejecución, lo primero es abrir ese puerto serial en el programa *arduino* [11]. Para ello se utiliza la función "*Serial.begin (9600)"* a 9600 baudios, para que funcione óptimamente.

2.1.5 Modulación por ancho de pulsos (PWM).

La *modulación por ancho de pulsos* o *PWM* es una de las técnicas más usadas en ingeniería y básicamente se usa para modificar el ciclo de trabajo de una señal o fuente de energía. Uno de sus usos básicos es para controlar la velocidad de un motor de corriente continua como en este caso, esto sin reducir o limitar el torque del motor.

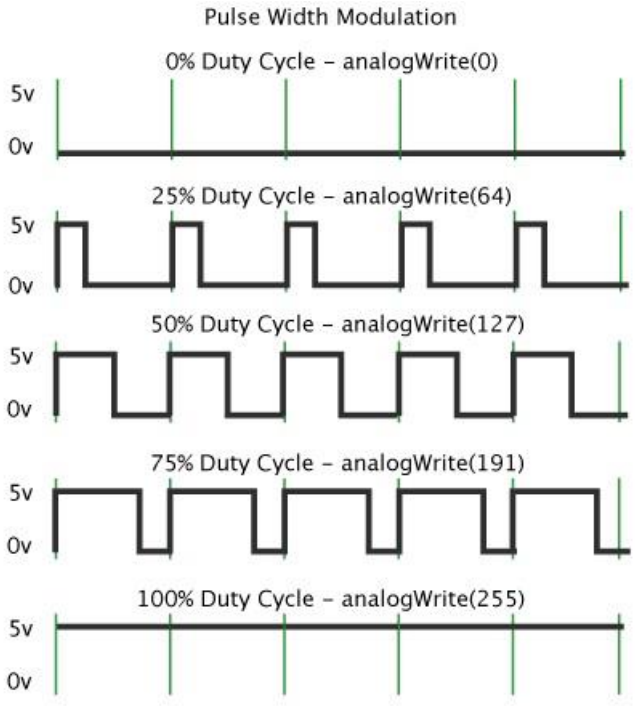

Figura 2.10. Imagen de una *señal PWM*.

Como se menciona en la referencia [12], en la actualidad existen muchos circuitos integrados en los que se implementa la *modulación PWM*, controles de motores y algunas otras aplicaciones.

2.1.6 Diseño electrónico

2.1.6.1 Etapa de potencia.

En los circuitos industriales es usual usar 3 etapas para garantizar la correcta funcionalidad del circuito, la etapa de control, etapa de aislamiento y etapa e potencia. [13,14] En este trabajo la etapa de control está conformada por un programa en *processing* en conjunto con un programa en *arduino*, el cual hace la función de puente con el hardware del *arduino* donde se encuentran las entradas y salidas. El programa en *processing* realiza los cálculos, las condiciones, ejecuta los algoritmos y muestra la interfaz gráfica.

En la *etapa de aislamiento* se puede encontrar *optoacopladores* que protegerán a la *etapa de control* de cualquier problema que se pueda suscitar en el circuito. En este caso se utilizará el circuito *4N35*.

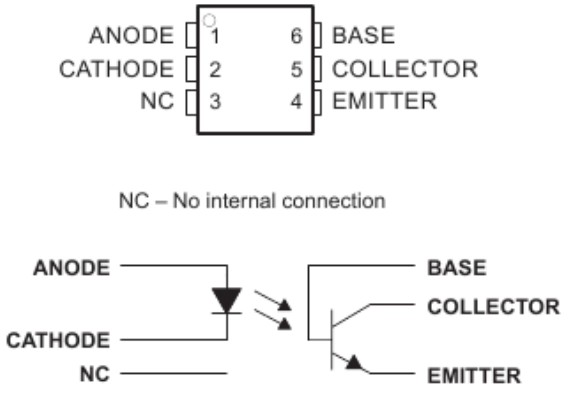

Figura 2.11. Estructura interna de un *optoacoplador*.

En la *etapa de potencia* se puede encontrar a un *tip 122* que alimentarán al motor con un voltaje de 24 v, debido a que la señal de *arduino* no tiene la suficiente potencia para activar al motor de corriente directa.

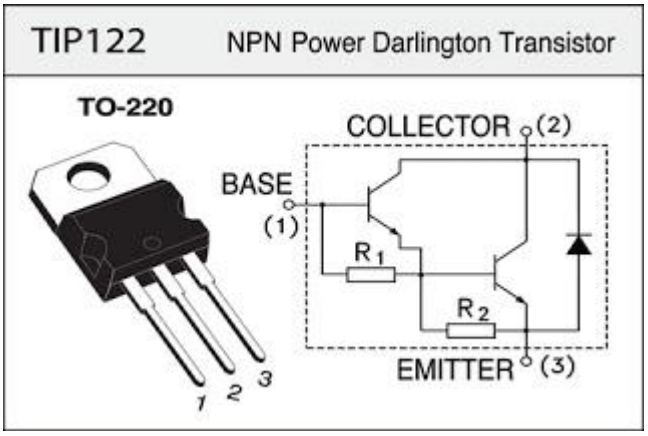

Figura 2.12. Circuito interno del *tip 122*.

2.1.6.2 Puente H.

El *puente H* es un circuito electrónico por el cual es posible controlar el sentido de giro de un motor (avance y retroceso), estos circuitos son de los más usados en la electrónica como se observa al ver la referencia [15], estos se usan comúnmente en robótica. Los puentes H están disponibles como *circuitos integrados*, pero también pueden ser construidos a partir de componentes discretos.

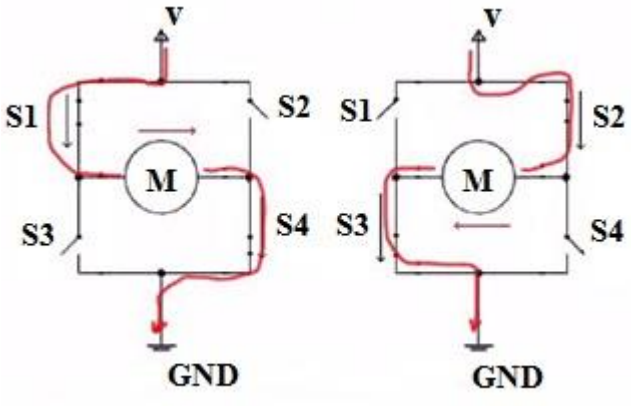

Figura 2.13. Circuito de un *puente H*.

Un *puente H* se construye con 4 interruptores (relevadores o mediante transistores). Para este trabajo se utilizaron relevadores.

2.1.6.3 Encoder digital.

Los *encoder* digitales son de los dispositivos más usados en la electrónica y el control, [16]. Los *encoders* son dispositivos electromecánicos que convierten la posición angular o movimiento de un eje de a un código analógico o digital.

Los *encoders rotativos* se utilizan en aplicaciones que requieren una medición precisa, incluyendo controles industriales, robótica, objetivos fotográficos especiales, etc. Un *encoder rotativo* está formado por un disco con marcas oscuras que se puedan detectar por medio de un sensor y una fuente de luz. En el libro *modeling and high-performance control of electric machines* [17] se observa cómo se puede deducir la distancia que ha avanzado el motor al que es necesario monitorear, esto por medio del número de marcas negras con los que cuenta el disco del *encoder*.

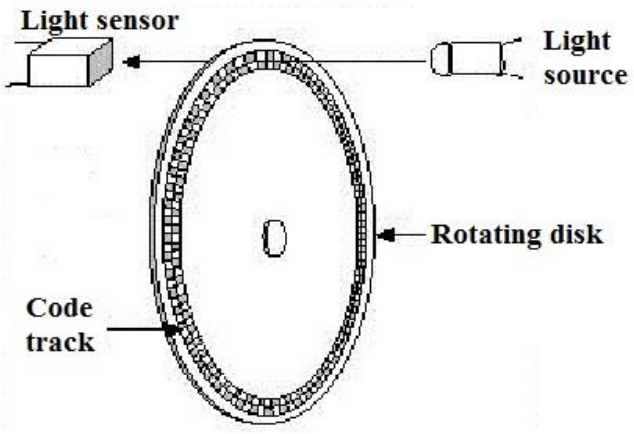

Figura 2.14. Imagen de un *encoder rotativo*.

Conociendo el número de marcas en el disco es posible estimar su avance en grados y conociendo el diámetro del eje del motor estimar su avance por centímetros.

Para el cálculo de la velocidad en revoluciones por minuto se utilizará la ecuación 2.4, [18], que relaciona la frecuencia obtenida del *encoder* y el número de pulsos que forman una revolución en el *encoder*.

$$RPM = \frac{(Hz)(60)}{No.pulsos} \tag{2.4}$$

en donde:

RPM se refiere a las revoluciones por minuto.

Hz es la frecuencia del encoder.

No. pulsos es el número de pulsos, en este caso será tratado como constante con valor de 120 y depende del modelo del encoder [23] .

2.2 Trabajos previos

En el ámbito del control de motores hay infinidad de formas de diseñar un control de velocidad, desde las formas más sencillas para los usos más rudimentarios hasta los controles más complejos para las aplicaciones y maquinas más avanzadas que requieren de los movimientos más precisos de un motor de corriente directa.

En [20], se demuestra cómo se puede diseñar una curva para generar un movimiento suave para controlar el arranque de un motor a través de cálculos matemáticos, más concretamente utilizando la curva de bézier. Su función es similar a las presentadas en este trabajo, se denotan 2 puntos en un plano cartesiano, el primer punto denota el inicio de la curva de aceleración y el segundo punto marca el final de la curva, es decir, las coordenadas donde el motor dejara de acelerar para llevar una velocidad constante. Al conocer estos 2 puntos se diseña una curva de aceleración con la que el motor podrá acelerar de la manera más suave y delicada posible.

Hay innumerables aplicaciones para el control de motores, una de las aplicaciones más comunes es el control de bandas transportadoras, debido a la variedad de objetos que se transportan en estas bandas (objetos sólidos, químicos inestables y toda clase de líquidos),

debido a la naturaleza de estos objetos muchos de ellos requieren de condiciones especiales, como un movimiento suave y lento al moverse. Para llevar esto a cabo se diseñan controles de velocidad para estas bandas transportadoras, como es posible observar en la referencia [21] donde se utiliza un *controlador PID* con ayuda de un *PLC* para construir un control de una banda transportadora.

Uno de los trabajos de interés para esta investigación es el trabajo de la referencia [22] que habla acerca de una interfaz gráfica diseñada en *labview* y que trabaja en conjunto con un *microcontrolador PIC16F877A* de la compañía *Microchip*. También se utilizaron procesadores que tenían la función de llevar registros de velocidad, posición y aceleración. Esto mediante una *señal PID* digital programable. En la señal que utilizaba era una señal trapezoidal, como antes observamos en nuestro trabajo esta señal es una de las señales básicas en los perfiles de velocidad. En el trabajo de la referencia [22] se observar cómo se utilizó un *encoder* para retroalimentar la señal de salida del sistema, esto es comparando la posición del *encoder* con la posición derivada de la señal trapezoidal del motor por medio de un *procesador LM628*.

El lenguaje usado en el *PIC* fue *lenguaje C* y la estructura del programa estaba constituida por un programa principal y varias funciones alternas que eran llamadas según se necesitaran, similares al presente programa en *processing*. El código del PIC se comunica mediante comunicación serial con la interfaz en *labview*, esto igual al trabajo aquí mostrado en *arduino* y *processing*.

Capítulo 3

Metodología

3.1 Análisis del sistema de ecuaciones.

El principal objetivo de la interfaz gráfica es poder funcionar correctamente y en conjunto con el sistema de ecuaciones que describen el movimiento de los perfiles de velocidad. Estas ecuaciones servirán para introducir parámetros tales como velocidad que se desea alcanzar, el tiempo y posición. Manipulando el tiempo en que se evalúa cada ecuación se pueden controlar parámetros tales como el *jerk* de la curva de aceleración. Estas ecuaciones serán la base para el resto del programa, debido a que si presentan alguna falla o no expresan los resultados correctamente el programa no funcionará y no se podrá controlar el motor correctamente.

Las ecuaciones para el control del perfil se obtendrán a partir de las ecuaciones de la recta. Tomando en cuenta como variables el tiempo y la aceleración. Para poder construir la curva de aceleración es necesario contar con un sistema de ecuaciones que permitan diseñar esta curva, para esto se ha partido de las ecuaciones de la recta para desarrollar dicho sistema de ecuaciones.

3.1.1 Incremento de aceleración.

Como se observa en la figura 3.1 la aceleración final está dada por la variable *Amax*, igual que el tiempo final está dado por *t1*. En este caso la aceleración inicial será cero y se representa como *Ain*, mientras que el tiempo inicial está dado por *t0*.

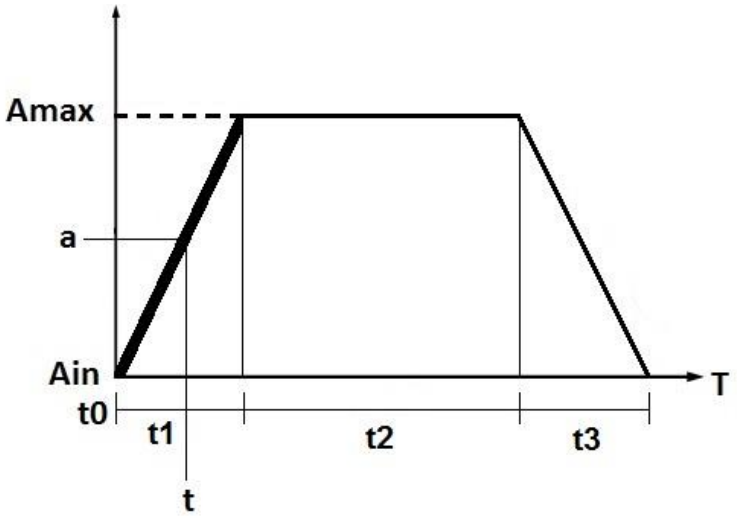

Figura 3.1. Incremento de aceleración

Partiendo de (2.1) que es la ecuación de la pendiente en su forma ordinaria, es posible desarrollar otras ecuaciones que servirán en el proceso.

Al sustituir las variables de (2.1) por las variables de aceleración y tiempo de la figura 3.1 da como resultado:

$$a = \frac{Amax - Ain}{t1 - t0} \tag{3.1}$$

en donde:

a es la aceleración instantánea de la curva.

Amax es la aceleración máxima que tomará la curva.

Ain es la aceleración inicial de la curva de aceleración

t1 es el tiempo que durara el segmento de la curva de aceleración.

t0 es el tiempo inicial de la curva de aceleración.

Al integrar (3.1) la aceleración instantánea de la curva (expresada como "*a*") se convierte en velocidad instantánea de la curva de aceleración. Dando como resultado la ecuación de velocidad (3.2).

$$V = \frac{Amax - Ain}{2(t1 - t0)}(t^2) \tag{3.2}$$

en donde:

V es la velocidad instantánea de la curva de aceleración.

t es el tiempo que se introducirá como dato, es la variable del sistema.

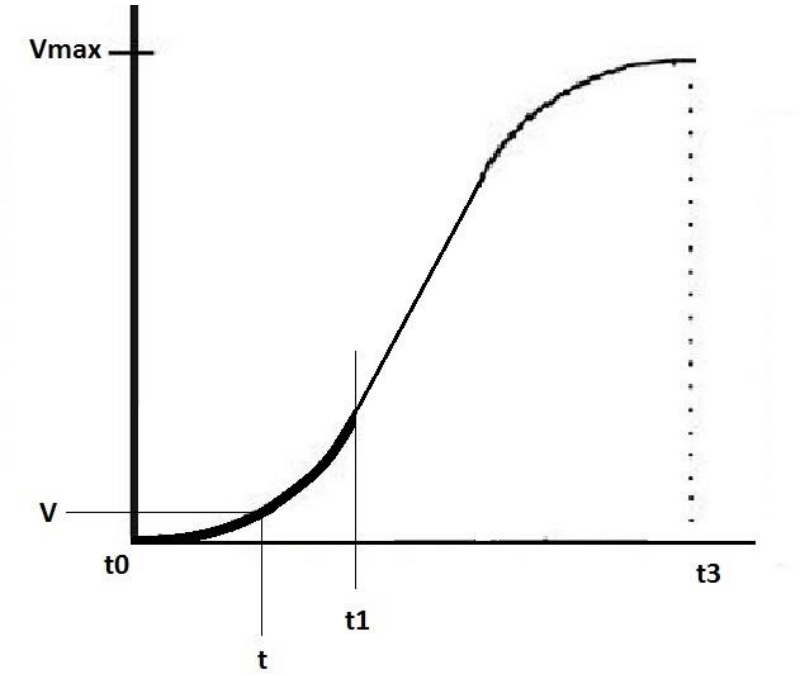

Figura 3.2. Curva de aceleración, enfatizando la ecuación 3.2.

La ecuación (3.2) será utilizada en el sistema para construir la primera sección de la curva de aceleración, también es posible simplificarla eliminando los valores iniciales como *t0* y *Ain*, dando como resultado:

$$V = \frac{Amax}{2\,t1}(t^2) \tag{3.3}$$

3.1.2 Aceleración constante.

En la figura 3.3 se observa cómo la aceleración se mantiene constante durante la curva, en este periodo es donde la velocidad adquiere un momento de aceleración constante. A diferencia del procedimiento anterior no es necesario integrar debido a que se trabaja con una recta donde el valor de aceleración inicial es *Amax* y el valor de aceleración final también será *Amax*, el tiempo estará dado por *t2*.

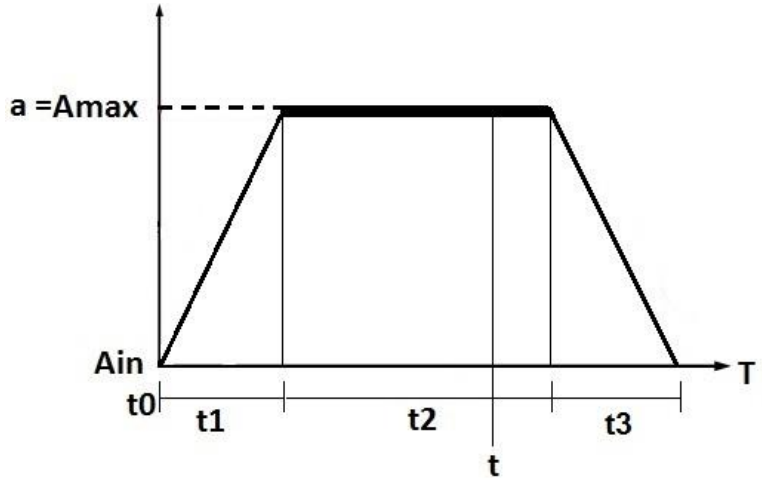

Figura 3.3. Aceleración constante.

Como se observa, al tratarse a la aceleración como una constante, el valor de la velocidad está dado por

$$V = (Amax)(t) \tag{3.4}$$

Para tener una ecuación implementable es necesario sumar la ecuación (3.4) con la ecuación (3.2), esto es para poder continuar con la segunda sección en el lugar donde terminó la primera parte. El resultado es la ecuación (3.5).

$$V = \frac{Amax - Ain}{2(t1 - t0)}({t_1}^2) + (Amax)(t) \tag{3.5}$$

Un cambio remarcable es que el tiempo *t* del segmento pasado de la curva ahora pasará a ser *t1*, debido a que en esta ecuación la sección de la curva que se evaluará es la sección de aceleración constante y el tiempo que se toma en el primer término de la ecuación es el tiempo que ya transcurrió al evaluarse la sección de incremento de aceleración, por lo tanto *t* del primer término pasa a tener su valor máximo, que equivaldría a *t1*.

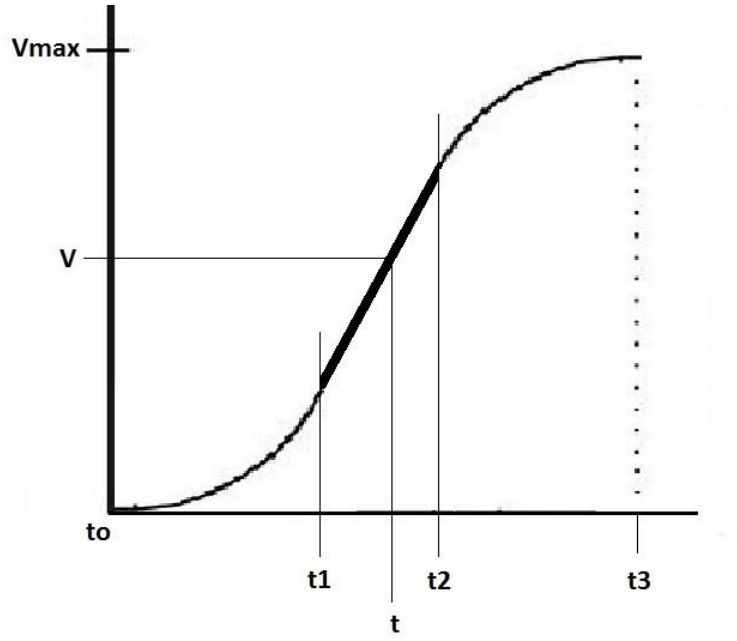

Figura 3.4. Curva de aceleración, enfatizando la ecuación 3.5.

La ecuación (3.5) servirá para evaluar la segunda sección de la curva de aceleración, es decir la aceleración constante y por medio de ella será posible conocer la velocidad instantánea en cualquier momento. Al igual que la ecuación (3.2) es viable simplificarla eliminando los valores iguales a cero.

$$V = \frac{Amax}{2(t1)}(t1^2) + (Amax)(t)$$

Al eliminar *Ain* y *t0* es posible dividir $t1^2$ entre $t1$.

$$V = \frac{Amax * t1}{2} + (Amax)(t)$$

Esta ecuación a su vez, es posible simplificarla aún más, dando como resultado la ecuación (3.6).

$$V = Amax\ \left(\frac{t1}{2} + t\right) \tag{3.6}$$

3.1.3 Decremento de aceleración.

Finalmente en la figura 3.5 se observa cómo la aceleración poco a poco empieza a disminuir hasta llegar a cero para mantener una velocidad constante durante el lapso de tiempo *t3*.

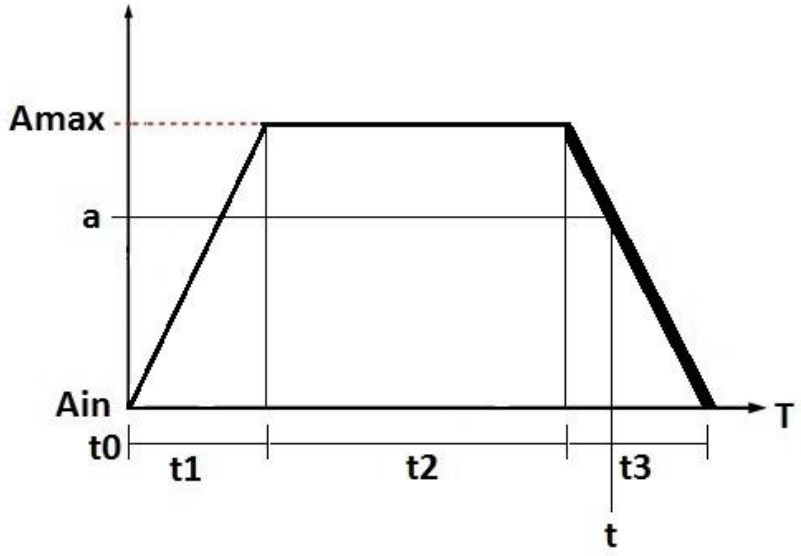

Figura 3.5. Decremento de aceleración.

Para desarrollar la parte final de la curva de aceleración es necesario utilizar la ecuación en su forma punto pendiente (2.2) en conjunto con la ecuación de la pendiente en su forma común (2.1). Lo mejor es partir de (3.1), debido a que en esta ecuación ya se han sustituido los valores de la gráfica.

Debido a que se ha utilizado la ecuación de la pendiente en su forma punto pendiente es necesario sustituirla en base a los valores de la gráfica lo que daría el siguiente resultado.

$$a = A(t) + Amax \tag{3.7}$$

en donde:

y se sustituyó por *a*, que es la aceleración instantánea que se calcula con la ecuación (2.2)

m se sustituyó por *A*, que es la aceleración instantánea que se calcula con la ecuación (2.1).

x se sustituyó por *t*, debido a que esta representa la variable del tiempo, en otras palabras eje X.

Finalmente *b* se sustituyó por *Amax*, debido a que es el valor en el que la recta intercepta al eje Y.

Una vez planteadas las variables utilizadas es posible continuar sustituyendo (3.1), en (3.7), para así poder trabajar con una sola ecuación, la cual queda como.

$$A = \frac{Amax - Ain}{t3 - t2}(t) + Amax \tag{3.8}$$

en donde:

t2 es el tiempo de aceleración constante

y *t3* es el tiempo del decremento de aceleración.

Integrando (3.8) se obtiene

$$V = \frac{Amax - Ain}{2(t3 - t2)}\, t^2 + Amax\,(t) \tag{3.9}$$

El último paso es sumar (3.9) y (3.5) para así tener la última ecuación, la cual describe la trayectoria que tomará el motor al reducir su aceleración. El resultado es

$$V = \frac{Amax-Ain}{2(t1-t0)}(t1^2) + (Amax)(t2) + \frac{Amax-Ain}{2(t3-t2)}\ t^2 + Amax\ (t) \qquad (3.10)$$

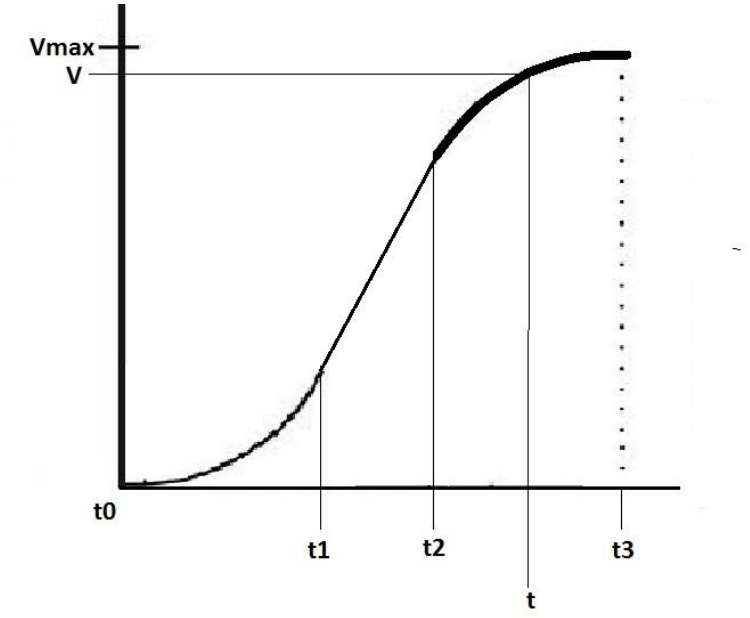

Figura 3.6. Curva de aceleración, enfatizando la ecuación 3.9.

Al igual que en las ecuaciones pasadas se eliminan los valores iniciales para poder simplificar la ecuación.

$$V = Amax\ \left(\frac{t1}{2} + t2 + \frac{t^2}{2(t3-t2)} + t\right) \qquad (3.11)$$

Teniendo el conjunto de ecuaciones, es posible pasar a graficarlas en *MATLAB* para poder observar la *curva de aceleración*.

Ecuaciones generales	Ecuaciones con valores iniciales iguales a 0	Sección de la curva
$V = \frac{Amax-Ain}{2(t1-t0)}(t^2)$ (3.2)	$V = \frac{Amax}{2\ t1}(t^2)$ (3.3)	Incremento de aceleración
$V = \frac{Amax - Ain}{2(t1 - t0)}(t1^2) + (Amax)(t)$ (3.5)	$V = Amax\ \left(\frac{t1}{2} + t\right)$ (3.6)	Aceleración constante
$V = \frac{Amax - Ain}{2(t1 - t0)}(t1^2)\ +\ (Amax)(t2) + \frac{Amax - Ain}{2(t3 - t2)}\ t^2 + Amax\ (t)$ (3.10)	$V = Amax\ \left(\frac{t1}{2} + t2 + \frac{t^2}{2(t3-t2)} + t\right)$ (3.11)	Decremento de aceleración

Tabla 3.1. Ecuaciones utilizadas en el programa.

3.2 Evaluación en MATLAB

Después de que se ha obtenido el sistema de ecuaciones que se implementarán en *processing* es necesario hacer una simulación para conocer el comportamiento de dichas ecuaciones al introducir las diferentes variables pertinentes para el presente trabajo, variables como velocidad, aceleración, tiempo y distancia, el mejor programa para hacer esto es *matlab* debido a la cantidad de funciones con las que cuenta, además de tener una amplia variedad de opciones al graficar ecuaciones matemáticas.

Para poder construir una gráfica de una *curva de aceleración*, las ecuaciones deben de evaluarse de manera secuencial, después de que se han cumplido los tiempos asignados. Estos

tiempos están declarados dependiendo del porcentaje de *jerk* o *empuje* que se requiera, estos cálculos se verán con más detalle en el capítulo 4, resultados.

Cada ecuación representa una parte en concreto de la *curva de aceleración*, la primera ecuación sirve para construir la parte de la aceleración incremental, en otras palabras esta ecuación tiene como función controlar la aceleración inicial que tendrá el motor de corriente continua.

Tomando como base a [7], en el siguiente ejemplo se aprecia cómo al introducir los datos se construirá la *curva de aceleración* en base a dichos valores. En el ejemplo se pretende alcanzar una velocidad de 1000 rpm por minuto, para esto se utilizan los siguientes datos:

- ✓ Tiempo de Aceleración = 100 segundos
- ✓ jerk = 50 %
- ✓ t1 = 25 segundos
- ✓ t2 = 50 segundos
- ✓ t3 = 25 segundos
- ✓ aceleración = 13.33 rev/seg^2

Al introducir los valores en la primera ecuación da como resultado la parte de la aceleración incremental de la curva. Para esto hay que tomar en cuenta que el tiempo se tomará desde cero para cada ecuación y la evaluación de la ecuación terminará hasta que el tiempo alcance su duración asignada. En este caso el tiempo durara 25 segundos, por lo tanto la ecuación (3.3) se valuará de 0 a 25 segundos, en intervalos de 2.5 unidades. En la ecuación (3.6) donde la aceleración es constante, el tiempo es de 50 segundos, al final de este segmento la curva habrá llegado a los 75 segundos (tomando en cuenta el primer segmento y el actual), pero en el programa la ecuación (3.6) se evaluará de 0 a 50 segundos, que es la duración de este segmento. El último intervalo donde empieza el decremento de aceleración dura 25 segundos, lo que hace que al terminar este segmento se alcancen los 100 segundos, pero al durar solo 25 segundos la ecuación se evaluará solo con esos 25 segundos.

El programa de *aceleración incremental* grafica la primera ecuación del sistema de ecuaciones (3.3), en la figura 3.7, se puede observar cómo la velocidad empieza en 0 y continua aumentando hasta llegar a 166 rpm, la figura 3.7 muestra claramente cómo es el incremento de velocidad del motor.

Aceleración incremental

```
clc, clear

t = 0:2.5:25;

Amax = 13.33;
Ao = 0;
to = 0;
t1=25;
t2=50;
t3=25;

v =((Amax-Ao)*(t.^2)) / (2*(t1-to));

plot(t,v);
grid on
```

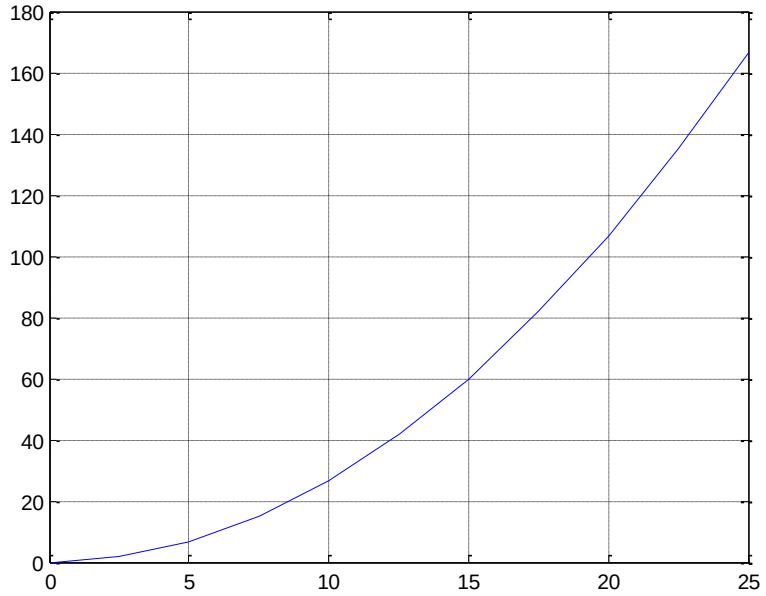

Figura 3.7. Grafica de la ecuación que controla el incremento de aceleración.

En el segundo programa se puede apreciar cómo la velocidad continúa con el último valor de la figura 3.7, se puede decir que este programa es la continuación del anterior, por eso esta ecuación se evaluará después de la primera parte del sistema, la ecuación (3.6) es la que se utiliza en este caso.

Aceleración constante

```
clc, clear

t = 0:2.5:50;

Amax = 13.33;
Ao = 0;
to = 0;
t1=25;
t2=50;
t3=25;

v = (((Amax-Ao)*(t1.^2))/(2*(t1-to))) + (Amax*t);

plot(t,v);
grid on
```

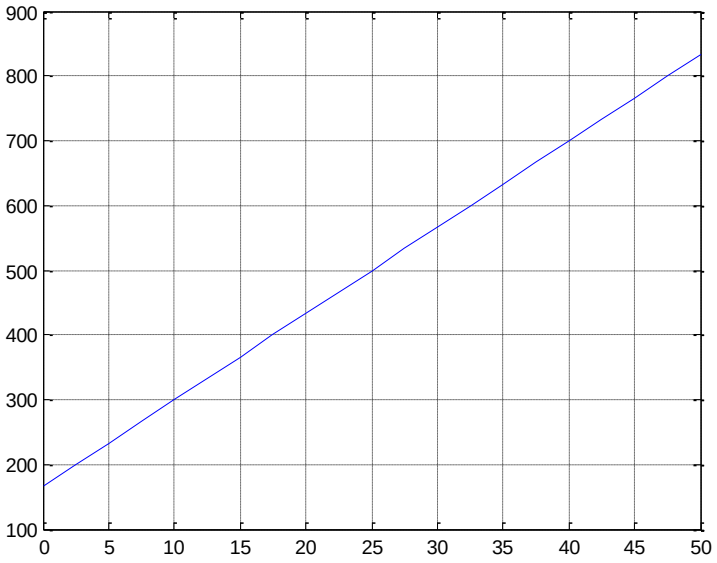

Figura 3.8. Grafica de la ecuación que controla la aceleración constante.

Al final se evalúa la tercera ecuación del sistema, la cual construye el decremento de aceleración mediante la ecuación (3.11), en otras palabras, construye el final de la curva de

aceleración, como se puede ver en la figura 3.9, el motor poco a poco empieza a disminuir su aceleración hasta llegar el punto donde la aceleración sea igual a 0, y la velocidad se mantenga constante.

Decremento de aceleración

```
clc, clear

t = 0:2.5:25;

Amax = 13.33;
Ao = 0;
to = 0;
t1=25;
t2=50;
t3=25;

v =(((Amax-Ao)*(t1.^2))/(2*(t1-to))) + (Amax*t2) + ( (Amax*t) + ( ((t.^2)*(Amax-Ao))/(2*(t3-t2))) );

plot(t,v);
grid on
```

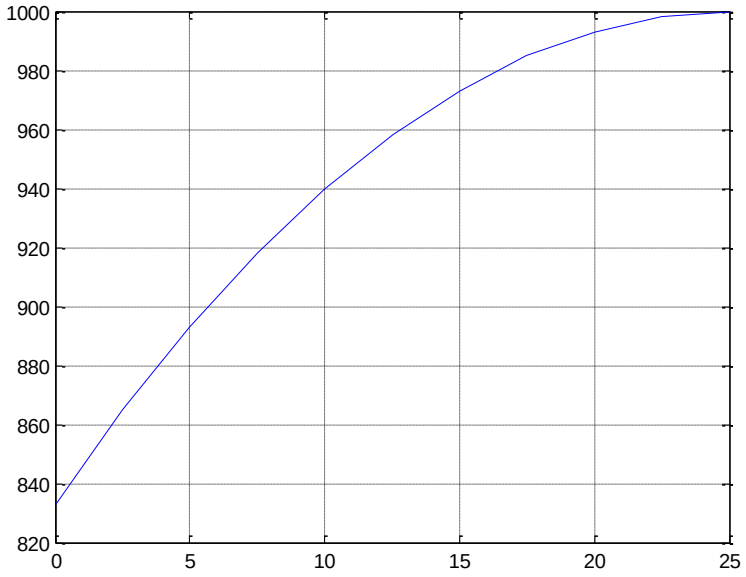

Figura 3.9. Grafica de la ecuación que controla el decremento de aceleración.

Al tener listas las ecuaciones, es posible construir la curva de aceleración al evaluar el sistema de manera secuencial. Al integrar el sistema de ecuaciones que se diseñó en un

programa de *matlab* el resultado será una *curva de aceleración* constituida por los segmentos que se han estudiado, (incremento de aceleración, aceleración constante y decremento de aceleración.)

```
clc, clear

ta = 0:2.5:25;
tb = 0:2.5:50;
tc = 0:2.5:25;

a = 13.33;
t1=25;
t2=50;
t3=25;

v1 = (a*(ta.^2))/(2*t1);
v2 = a*((t1/2)+tb);
v3 = (((-a * (tc.^2))/(2*t3))+(a*tc))+833  ;


plot(ta,v1);
hold on
plot(tb+25,v2,'r');
hold on
plot(tc+75,v3,'g');
grid on
```

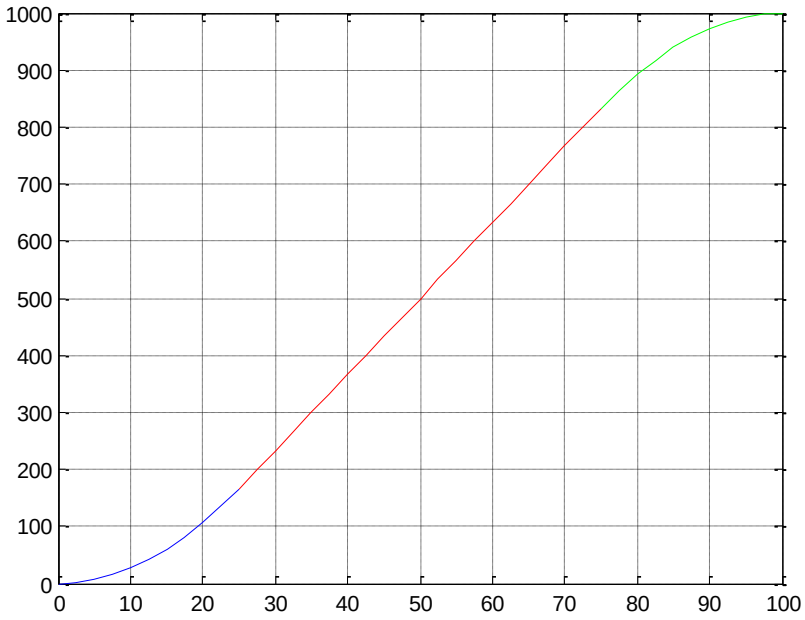

Figura 3.10. Imagen de una *curva de aceleración* de un perfil de velocidad.

Observando la figura 3.10 se aprecia que el primer segmento acaba con una velocidad de 166 rpm, mientras que el segundo segmento de aceleración constante acaba en 833 rpm dando inicio a la etapa final donde la aceleración se detiene empezando con un periodo de velocidad constante.

3.3 Diseño en processing

Debido al enorme potencial de *processing* se espera a diseñar una interfaz bastante detallada que exprese de manera clara y precisa la curva de aceleración, la velocidad y posición del sistema. Lo principal es introducir las ecuaciones y formulas necesarias para lograr generar una *curva de aceleración* que servirá para controlar el movimiento del motor. La interfaz constará de una pantalla principal con un indicador grafico de la posición del motor en la guía de metal, además de 3 graficas que mostrarán la aceleración, la posición del motor y la velocidad del motor en todo momento.

La interfaz en *processing* cuenta también con un manual de uso donde estarán todas las funciones y su correcto uso además de un panel de información. El programa está dividido en clases, donde cada una es una función específica del programa, y se activa dependiendo de las operaciones que se lleven a cabo. Esta estructura del programa es a razón de mantener organizado el código. Si en el futuro es necesario hacer modificaciones o actualizaciones seria relativamente sencillo identificar las partes del código.

La pantalla inicial del programa será la *pantalla principal*; es aquí donde se podrá seleccionar el tipo de control que se desee para posteriormente comenzar a trabajar. También es aquí donde se puede acceder al menú del programa. Al observar la figura 3.11 se observa cómo los 4 diferentes tipos de control se basan en el uso de la misma clase para generar una *curva de aceleración*, también se puede apreciar como los resultados se muestran en los diferentes tipos de indicadores del programa además de ser enviados al puerto serie, no sin antes tratarlos para ajustarlos a los parámetros de la función *PWM de arduino*. El sistema cuenta con un *encoder* que se utiliza como retroalimentación para que el motor mantenga la

velocidad deseada, además que los datos registrados por el *encoder* serán mostrados en la pantalla principal. En la clase de "*encoder*" es donde se calcula la velocidad del motor, y se cuentan los pulsos enviados desde *arduino*. También aquí se realizan las operaciones necesarias para convertir los pulsos en posición, (expresada en centímetros), la posición es actualizada constantemente para poder mostrar una lectura correcta del motor y es mostrada en la pantalla principal en forma de una gráfica y en el centro de la pantalla en la guía de metal.

Para controlar el motor se han diseñado varias funciones que hacen énfasis en modificar diferentes aspectos del movimiento del motor. Cada función contará con diferentes variables para modificar el comportamiento del éste, como la velocidad, el tiempo que le tomará recorrer determinada distancia o la aceleración que tendrá al aumentar la velocidad.

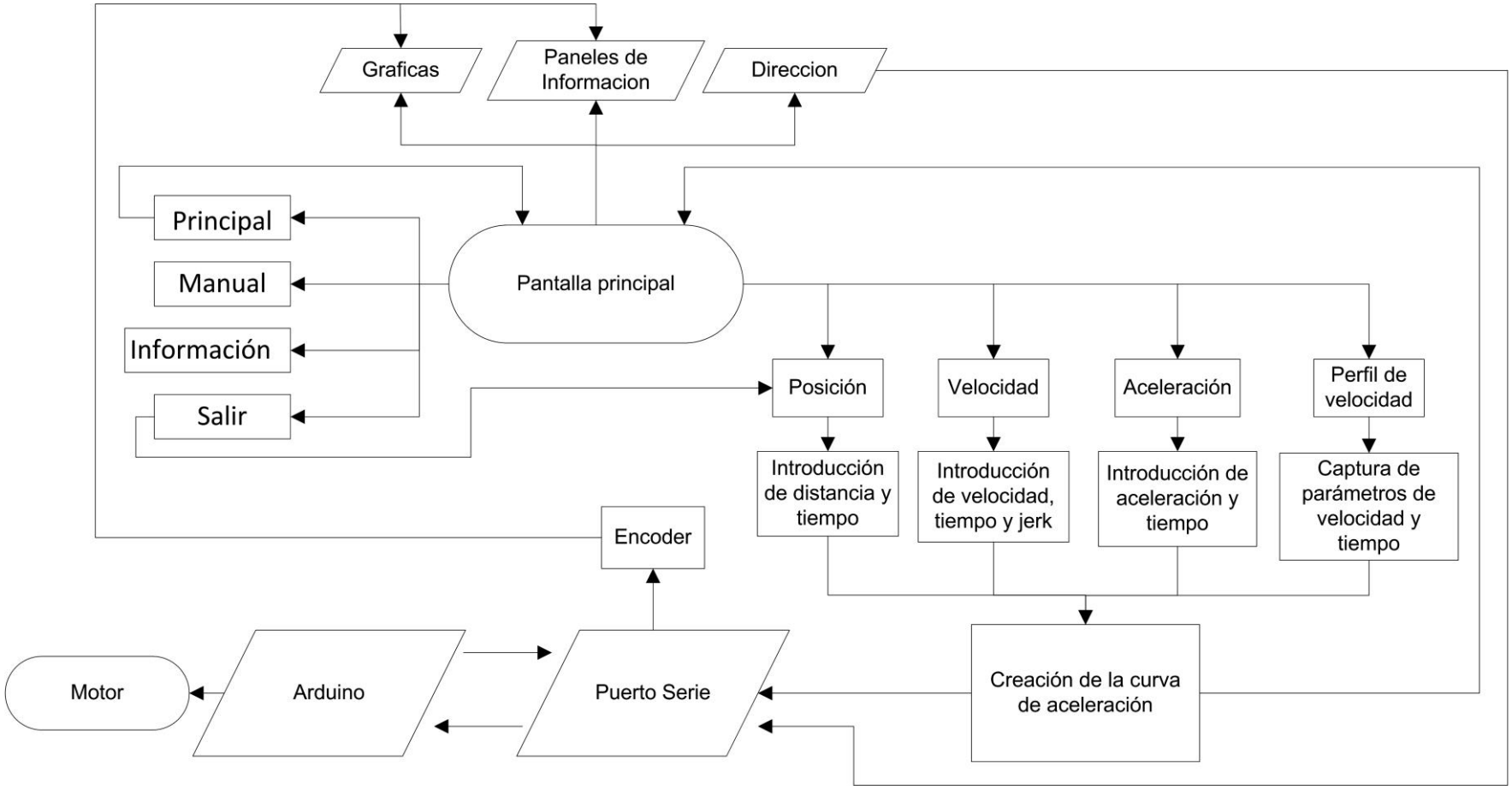

Figura 3.11. Diagrama de la estructura del programa en *processing*.

3.3.1 Algoritmo para el diseño de curvas de aceleración.

Mediante la clase de *curva de aceleración* se crean las curvas con las que se iniciarán y finalizarán los perfiles de velocidad. Éste algoritmo tiene como tarea evaluar secuencialmente cada ecuación que se necesite para crear una *curva de aceleración.* Empezando con la captura de datos de parte el usuario para posteriormente calcular los tiempos necesarios t1, t2 y t3.

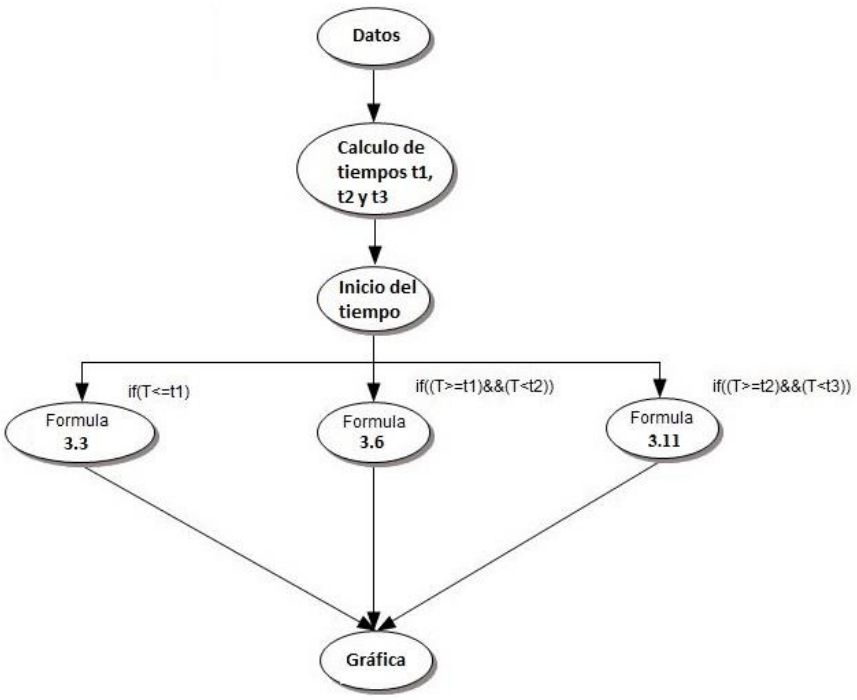

Figura 3.12. Algoritmo para diseñar curvas de aceleración.

Al contar con los datos necesarios listos se procede a iniciar un temporizador cuya función principal es ser usado para evaluar las ecuaciones debido a que será nombrado como "*t*" y será introducido en éstas. Mediante condiciones "*if"* se seleccionará la ecuación correspondiente de acuerdo al tiempo transcurrido como puede verse en la figura 3.12.

La captura de datos se realiza en varios controles, cada uno de éstos calcula la velocidad instantánea y la aceleración que el motor deberá adoptar en cada momento que dure el movimiento y manda dicho valor al indicador de la pantalla principal. Además de lo anterior

el valor de la posición es graficado en la *pantalla principal.* Estos 3 indicadores funcionan sin importar el tipo de control que se esté utilizando. El programa cuenta con un cuarto control el cual consiste en marcar puntos en un plano para generar un perfil de velocidad.

3.3.2 Control de velocidad

El control de velocidad genera curva de aceleración en base a valores introducidos, *velocidad final*, *tiempo*, y *empuje*. Al tener estos valores se calcula los tiempos que deberán tener cada sección de la curva, *aceleración incremental*, *aceleración constante y el decremento de aceleración*, dichos cálculos se verán a detalle en el capítulo 4, estos valores estarán a razón del *empuje* introducido. Al tener todos los valores necesarios para generar la curva se manda a llamar a la clase *curva de aceleración*.

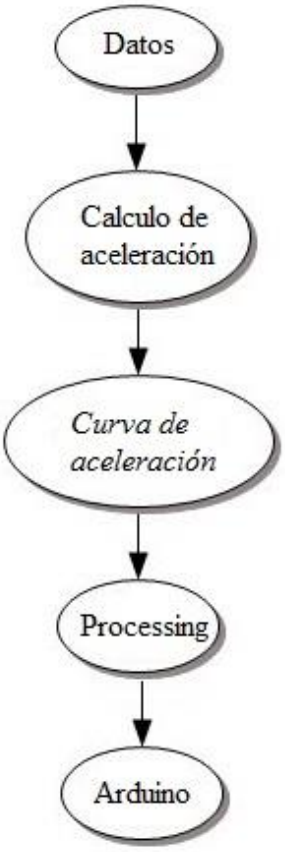

Figura 3.13. Algoritmo del control de velocidad.

La clase *curva de aceleración* se encarga de evaluar las ecuaciones (3.3), (3.6) y (3.8), y el resultado se envía a ser expresado en los indicadores de la pantalla principal, en donde primero se apreciará el movimiento del motor en la animación del centro de la pantalla al mismo tiempo que se grafica la *velocidad instantánea* en la gráfica central en la parte superior de la interfaz.

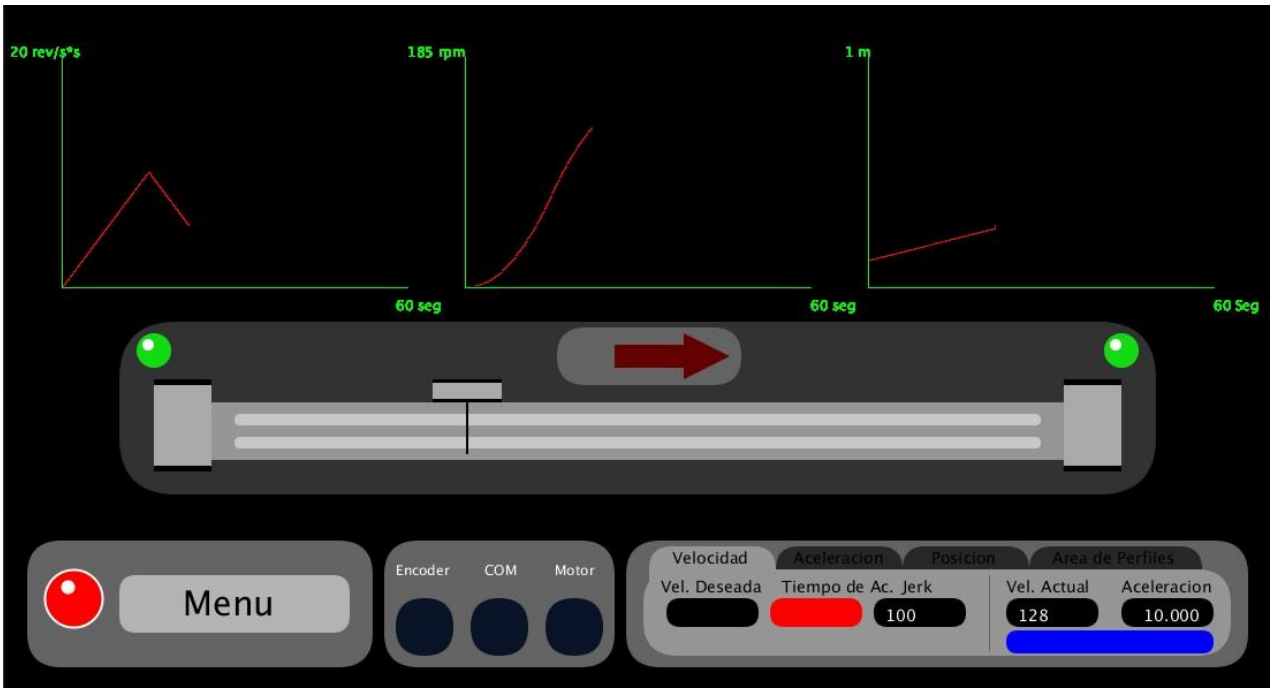

Figura 3.14. Interfaz trabajando con el control de velocidad.

Al observar la figura 3.14 se aprecia cómo se dibujan las gráficas mostrando los datos de velocidad, aceleración y posición. La gráfica del medio muestra la velocidad expresada en revoluciones por minuto (rpm) con un máximo de 185 rpm. Se debe señalar que las demás graficas también llevaran a cabo sus procesos mostrando la información para la que se han diseñadas, aceleración a la izquierda y posición a la derecha.

3.3.3 Control de aceleración.

El control de aceleración en gran parte es similar al control de velocidad, exceptuando que no es necesario introducir la velocidad, pero es necesario introducir como dato la aceleración que se desea que tome el motor. Después de introducir los datos, *tiempo, aceleración máxima* y el *empuje*, se procede a realizar las operaciones, empezando por calcular los tiempos de las secciones del código, y al tener todos los valores necesarios listos se manda a llamar a la clase de *curva de aceleración*.

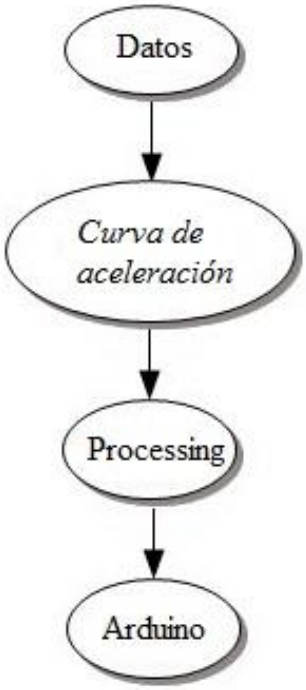

Figura 3.15. Algoritmo del control de aceleración.

3.3.4 Control de posición.

El control de posición para este sistema se basa en la conversión de la distancia lineal que se desea en la guía a las revoluciones que deberá de avanzar el motor para alcanzar dicha distancia lineal. Para esto se tomó en cuenta el diámetro del eje del motor y la formula de la circunferencia de un círculo.

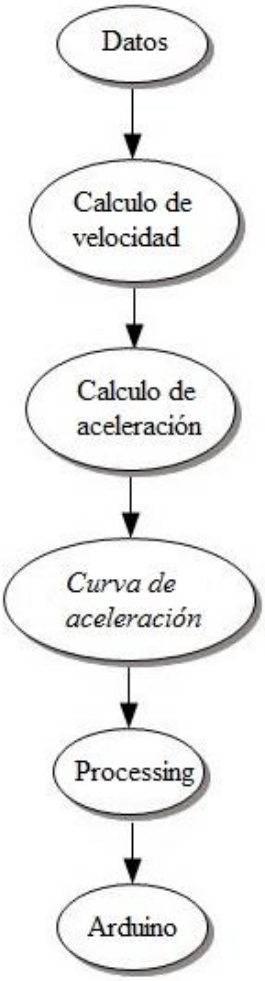

Figura 3.16. Algoritmo del control de posición.

Será necesario realizar dos *curvas de aceleración* para un correcto desplazamiento, el objetivo de ésta última curva de aceleración es que durante todo el avance del motor se tenga un movimiento suave y controlado, la primera parte de la curva corresponde a la primera

mitad del tiempo que el motor se deberá mover, la segunda parte es la mitad de movimiento donde el motor comenzara a disminuir su velocidad lentamente hasta llegar al reposo del motor.

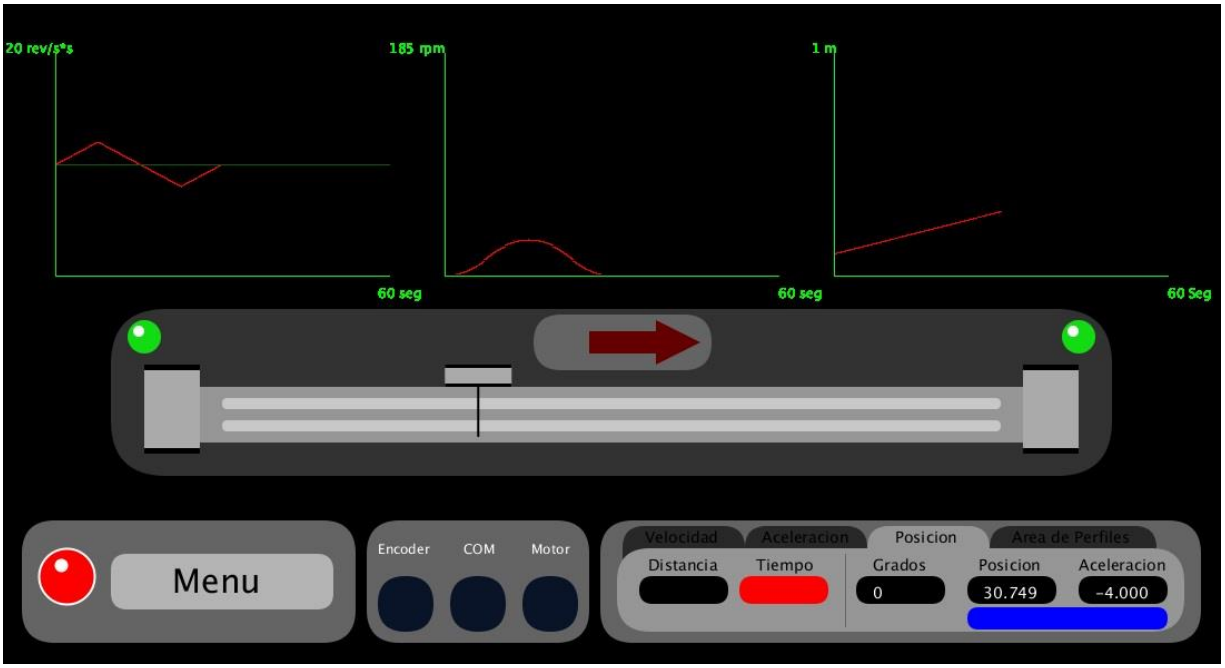

Figura 3.17. Interfaz trabajando con el control de posición.

En este punto es necesario realizar un desarrollo matemático para encontrar las ecuaciones que brindarán los valores necesarios para esta clase. El *control de posición* pide en su *interfaz* como datos el *tiempo* y *posición* que se desea se vea reflejado en el motor, el problema es llegar a la *velocidad* a partir de estos valores. Para solucionar esto se debe calcular la velocidad a partir del tiempo y posición como variables. Es posible integrar la ecuación (2.3) para poder calcular una velocidad en vez de una aceleración dando como resultado la ecuación (3.12) la cual calcula el valor de la velocidad instantánea en base a una velocidad final y el tiempo de la curva de aceleración.

$$V = \frac{Vf(t^2)}{t_1+2t_2+t_3} \tag{3.12}$$

Igualmente (3.12) puede integrarse nuevamente y de esta manera poder calcular la posición a partir de la velocidad introducida por el usuario.

La segunda integración da como resultado a (3.13), cuyo resultado es una posición pero para los cálculos es necesario obtener la velocidad introduciendo una posición y el tiempo, lo siguiente será despejar la ecuación.

$$Pos = \frac{Vf(t^3)}{t_1+2t_2+t_3} \tag{3.13}$$

El resultado final es

$$Vf = \frac{Pos\,(3t_1+6t_2+3t_3)}{t^3} \tag{3.14}$$

3.3.5 Diseño de perfil.

Por mucho, la programación de este control fue el más elaborado y probablemente es el que mejor representa los objetivos del presente trabajo, ya que con esta función es posible diseñar un perfil completo utilizando un método sumamente grafico e intuitivo.

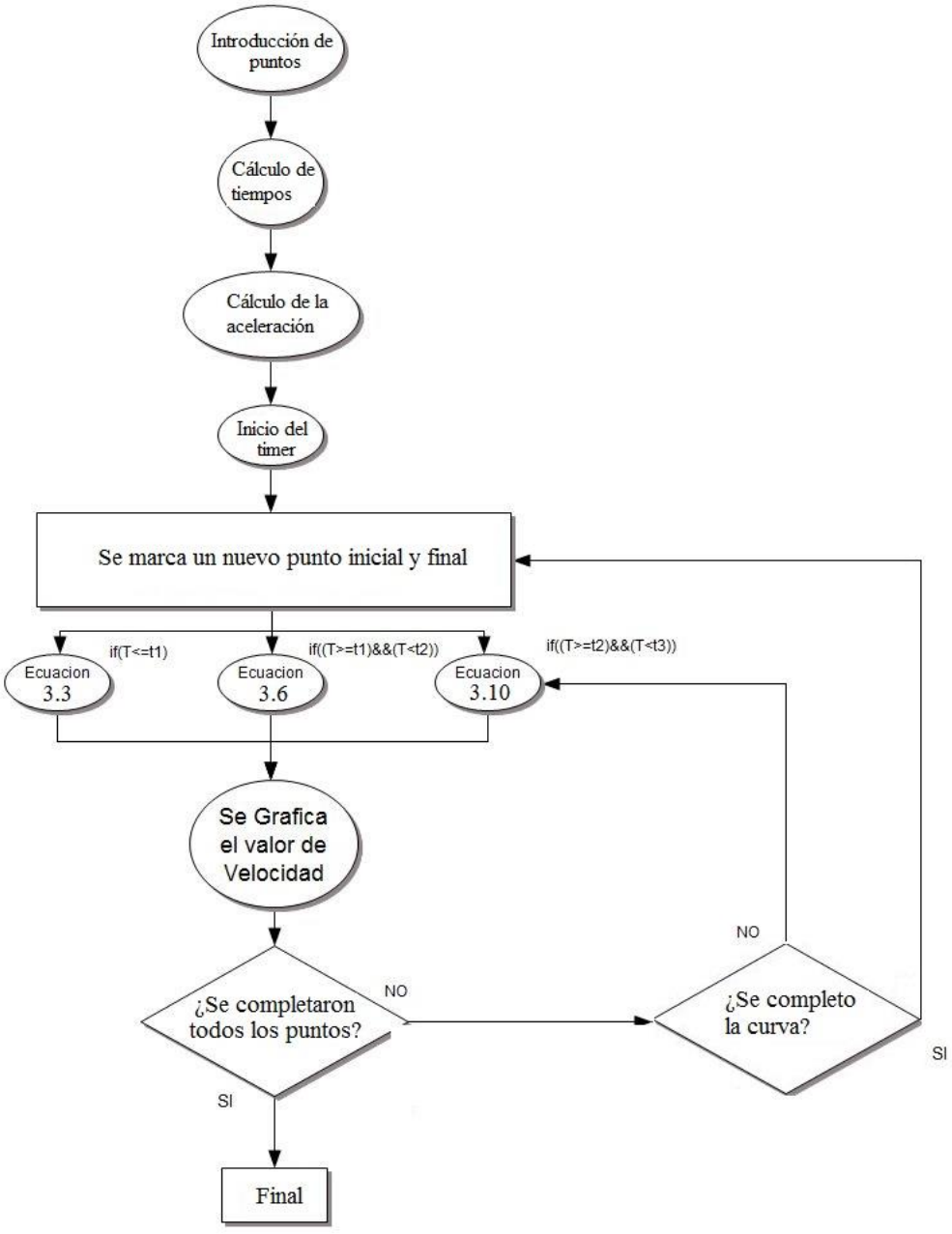

Figura 3.18. Algoritmo del diseño de perfil.

Este control consiste en diseñar un perfil de velocidad marcando puntos en una área instrumentada, para así asignarles un valor a las coordenadas *X, Y*, respecto al tiempo en *X* y a las revoluciones por minuto en *Y*. El programa enlista todos los puntos y sus coordenadas, para asignarles un valor numérico dentro de un arreglo. El programa procede a guardar el primer valor y el punto inicial en 4 variables denominadas *Xpin* y *Ypin* y las variables *Xpfin* y *Ypfin* guardarán las coordenadas del punto siguiente, así siempre se tendrán un juego con 2 puntos, uno inicial y uno final que siempre estar seguidos uno del otro. Estos 2 puntos conforme el tiempo avance cambiarán de valores, cada uno tomando los valores del punto siguiente, así el programa podrá volver a evaluarlos pero ahora el que antes era el punto final pasará a ser el inicial, así la curva nueva podrá continuar dibujándose donde termino la curva anterior, pero con valores diferentes. Cuando el programa alcance el valor en *X* de un punto final sucede el cambio entre puntos.

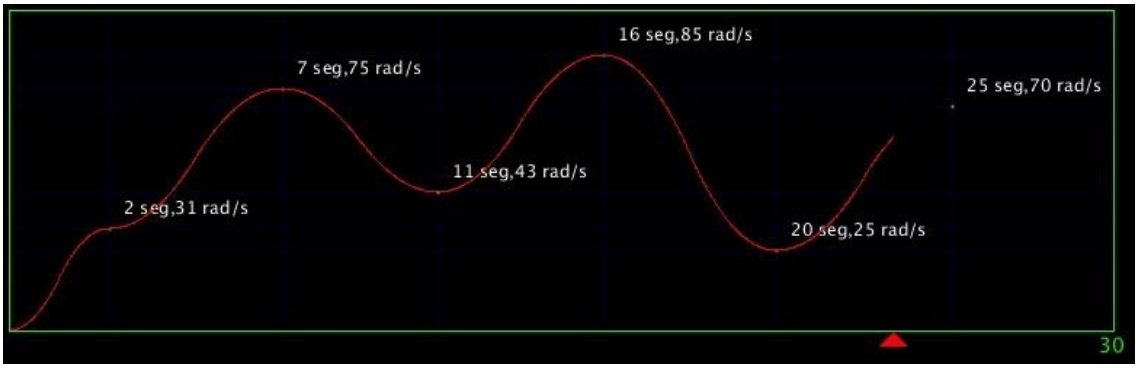

Figura 3.19 a. Coordenadas dibujando un perfil de velocidad.

Figura 3.19 b. Coordenadas capturadas.

3.3.6 Menú.

Entre las funciones que se pueden encontrar en la interfaz, cabe resaltar el menú para seleccionar las opciones con las que cuenta el programa.

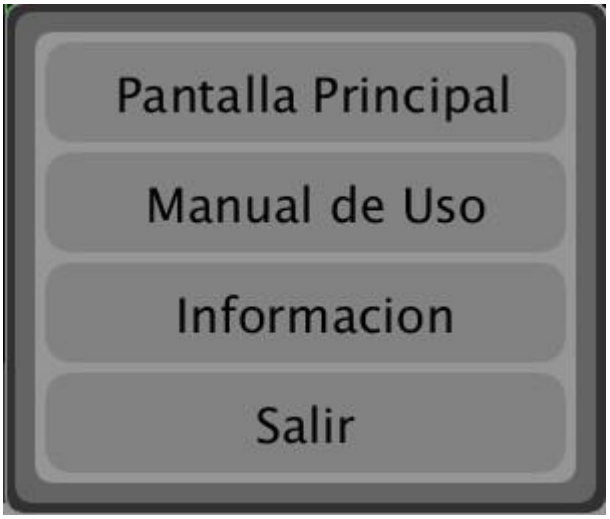

Figura 3.20 Imagen del menú.

En el menú se encuentran 4 opciones para escoger, la primera es la pantalla principal, aquí es donde se puede seleccionar el control que se desee o si se necesita diseñar un perfil de velocidad. Un manual de uso donde se encuentra información acerca de las funciones con las que se cuenta y su correcta utilización, una pantalla con información y al final la opción "*salir*" que permitirá cerrar el programa.

En estas opciones, la función de *Salir* al ser activada desencadena un proceso, con el cual el programa se cerrara adecuadamente. Si el programa es cerrado de otra manera, una manera incorrecta, esto puede propiciar a que el motor continúe girando después de cerrado el programa en *processing*, o que el *puente H* permanezca activado en reversa. Con la función diseñada para esto primero se cancela el envío de datos si el motor se encuentra en movimiento. Después se procede a regresar al motor a su posición de origen, para finalmente cancelar definitivamente todo envió de datos a *arduino*. Finalmente completado este proceso, el programa se cerrara definitivamente de una manera adecuada.

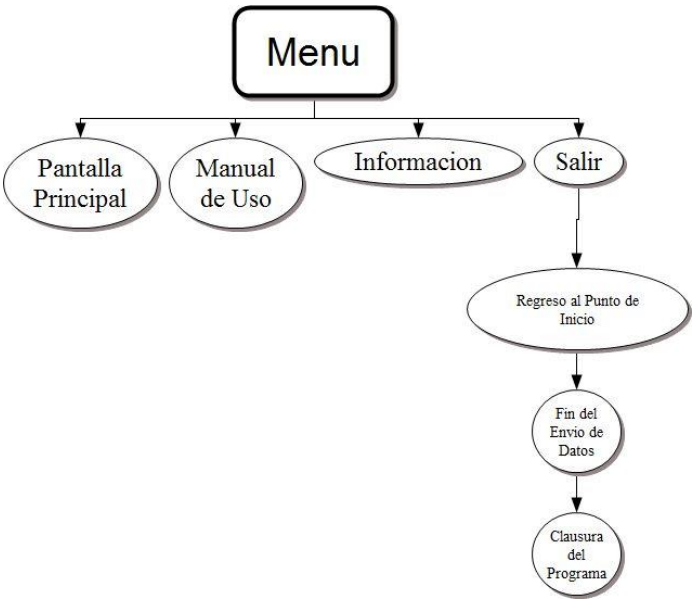

Figura 3.21. Estructura del menú.

3.4 Implementación en *arduino*.

Arduino es una plataforma con una gran flexibilidad, una de las ventajas es que puede comunicarse mediante *comunicación serial* con una gran cantidad de *entornos de desarrollo*, en este caso *processing*.

Para desarrollar la comunicación entre los 2 programas, se necesita adaptar los 2 códigos del sistema para que puedan comunicarse entre sí. Para esto se ha utilizado una técnica de comunicación llamada c*omunicación bilateral*, que consisten en que ambos programas (*arduino* y *processing*), actúen como emisor y receptor en determinados momentos.

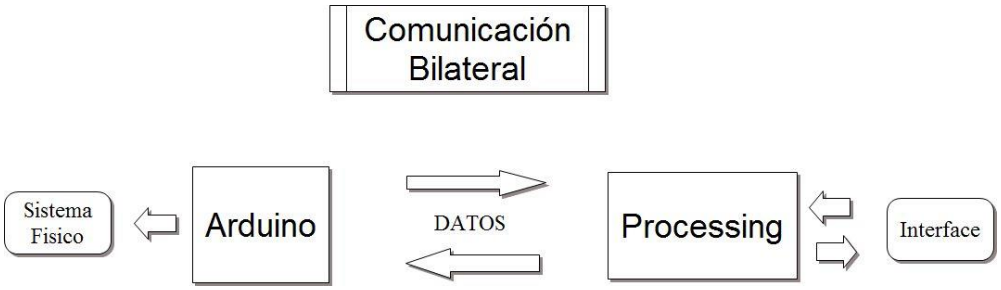

Figura 3.22. Comunicación bilateral.

Esto quiere decir que *arduino* y *processing* estarán intercambiando datos continuamente, en un ciclo *arduino* enviara datos por el puerto serie y *processing* los recibirá e interpretara, pasado esto *processing* será quien envíe los datos para que *arduino* los lea e interprete, en el siguiente ciclo se volverá a repetir el proceso.

3.4.1 Comunicación bilateral en processing.

El código básico en *processing* para esto es el siguiente:

```
//COMUNICACIÓN BILATERAL CON PROCESSING
import processing.serial.*;      //Se declara la librería con las funciones seriales
Serial my Port;                  //Se declara el puerto que utilizaremos
String datos1;                   //Se declara la variable donde guardaremos nuestros
                                   datos
boolean firstContact=false;      //Contacto con processing

void setup() {
  size(200,200);                                    //Se establece el tamaño de la pantalla
  myPort=new Serial(this, Serial.list()[8],9600);   //Se le asigna un nombre al puerto y los
                                                      baudios a los que se hará la
                                                      Comunicación.
}
void draw() {
    // Proceso del programa
}
void serialEvent(Serial myPort){
  datos1 = myPort.readStringUntil('\n');      //Se lee los datos en el puerto serie y se
                                                guardan en la variable datos1.
println(datos1);                              //Se escriben los datos en el panel de
                                                processing.
  }
//Busca la H para establecer el contacto e indicar el handshake
  firstContact==false){
```

```
        if (datos1.equals("H")){
                myPort.clear();
                 firstContact=true;


                  myPort.write("H");                  //Se escribe una “H” en el puerto
                  println("contacto establecido");   //Se escribe en el panel “contacto
                                                      Establecido”
        }} else {              //Si se tiene el contacto, continuar recibiendo datos
  myPort.write('1');          //Se envía un 1 a el puerto serie.
}
//Cuando el contacto está establecido y los datos organizados, Se piden más datos.
myPort.write("H");
        }}
```

El programa en p*rocessing* es muy sencillo, se puede decir que el programa pregunta a la parte en *arduino* si hay comunicación, esto lo hace mandando un carácter, en este caso una “*H*”, cuando el programa en *processing* comprueba que la comunicación fue un éxito, entonces se procede a escribir los datos en el *puerto serie* que posteriormente leerá *arduino*. Si la comunicación fallo y no hay respuesta se limpian los datos enviados al puerto serie y se procede a reenviar el carácter “*H*” al puerto serie, esto para esperar de nuevo si hay respuesta de parte del otro programa. Como se observa, este programa continuamente está leyendo los datos del puerto serial y los almacena en la variable *datos1*.

Al principio es necesario declarar la velocidad de comunicación de los 2 programas, esta velocidad, expresada en baudios, deberá ser la misma para las 2 partes.

3.4.2 Comunicación bilateral en *arduino*.

El código básico en *arduino* para esto es el siguiente:

```
//COMUNICACIÓN BILATERAL CON ARDUINO.
int inByte = 0;                    // Variable serial byte
```

```
void setup(){
  Serial.begin(9600);                 //Empieza la comunicación serial
  while (!Serial) {
  }
  establishContact();                  // Envía un byte para establecer la conexión
}

void loop ()
{
  if (Serial.available() > 0) {
    inByte = Serial.read();             //Lee el puerto serial
    Serial.write(1);                    // Imprime 1 en el puerto serial.
  }}

void establishContact() {
  while (Serial.available() <= 0) {
    Serial.print('H');                  // Envia H al puerto serie
    delay(100);
  }}
```

El funcionamiento de este programa es similar a su versión en *processsing*, aquí no hay necesidad de declarar librerías para empezar a trabajar, debido a que siempre están incluidas en el programa.

Primero se declara la variable donde se guardan los datos leídos en el *puerto serial*, en el ejemplo es la variable "*inByte*", es necesario declarar la velocidad de comunicación que se maneja, y esta tendrá que ser igual que en la versión en *processing*, lo más confiable es utilizar *9600 baudios*. Al igual que en *processing* se envía un carácter al *puerto serial*, esto para poder inicial la comunicación entre los 2 programas, cuando se lea el carácter entonces los 2 programas podrán empezar con la comunicación. Este código guardará los valores que lee en una variable declara al principio de la programación.

3.4.3 Acoplamiento de la señal *PWM*.

Al examinar el *encoder* del motor se encontró que era de tipo *GP1A16R* y a su vez este *encoder* contaba con 2 modelos el tipo A y el tipo B, los cuales se caracterizan por tener una resolución de 120 pulsos por vuelta el tipo A y 200 pulsos por vuelta el tipo B, [23].

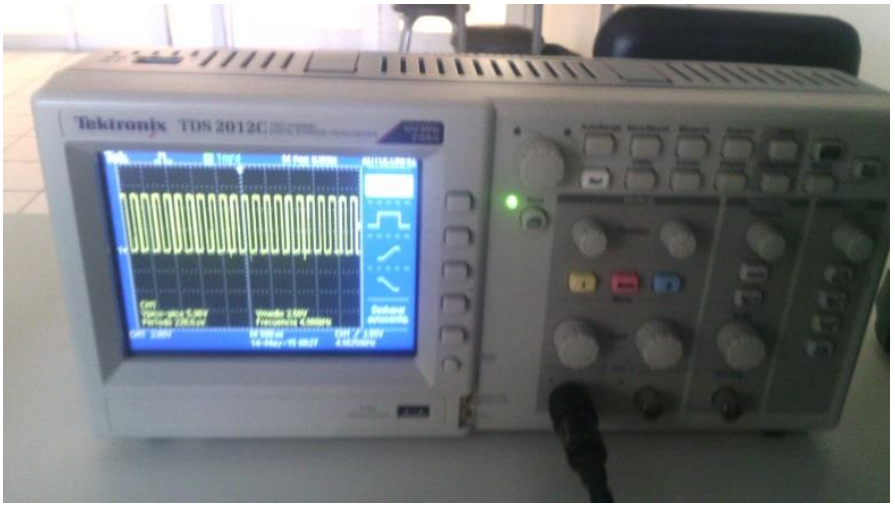

Figura 3.23. Osciloscopio midiendo una señal enviada por el *encoder*.

Utilizando la ecuación (2.4) y algunos valores capturados por medio de un osciloscopio es posible hacer un cálculo para estimar la velocidad en *rpm* que alcanza el motor de corriente directa utilizado en el presente trabajo.

Voltaje	Frecuencia (KHz)	RPM
5v	1.724KHz	862 rpm
8v	2.809KHz	1404 rpm
12v	4.331KHz	2165 rpm
24v	8.911KHz	4455 rpm

Tabla 3.2. Cálculo de revoluciones por minuto de diferentes voltajes (sin tomar en cuenta la relación de transmisión, los valores han sido redondeados).

El siguiente paso era conocer la relación de transmisión del motor, esto fue un problema debido a que no se encontró información confiable que facilitara este dato, en la investigación se encontró una posible relación de transmisión, la cual era 24:1,[24].

RPM	RPM (24:1)
862 rpm	36 rpm
1404 rpm	56 rpm
2165 rpm	90 rpm
4455 rpm	186 rpm

Tabla 3.3. Calculo de revoluciones por minuto de diferentes voltajes (Tomando en cuenta la relación de transmisión de 24:1, los valores han sido redondeados).

Ahora ya se cuenta con una velocidad nominal en el motor, aunque estaba presente la duda si estos valores eran confiables, por tanto se procedió a calcular estos datos mediante un proceso experimental, el cual era un proceso en donde se medía el número de vueltas que realizaba el motor en un minuto, esto con ayuda de un cronometro. Para esta tarea también se diseñó una fórmula que sirviera para calcular las revoluciones por minuto en base a la frecuencia del *encoder*. Los resultados están expresados en la tabla 3.4.

$$RPM = (KHz)(C) \tag{3.12}$$

en donde:

RPM se refiere a las revoluciones por minuto.

KHz es la frecuencia del *encoder* expresada en Kilos.

C es una constante con valor de 20.8395 producto de dividir la frecuencia y las revoluciones por minuto medidas en el método experimental.

Voltaje	Frecuencia (KHz)	RPM
5v	1.724KHz	36 rpm
8v	2.809KHz	59 rpm
12v	4.331KHz	91 rpm
24v	8.911KHz	186 rpm

Tabla 3.4. Datos capturados mediante un método experimental (los valores han sido redondeados).

Al observar los resultados de las tablas 3.4 y 3.3 se puede apreciar un gran parecido con los resultados de la *RPM*, a pesar de que el método experimental descrito anteriormente fue útil para corroborar que la relación de transmisión utilizada es correcta, no es posible tomarlo como 100% confiable debido a que depende mucho de un operador que realice las lecturas visualmente, sin embargo fue útil para comprobar la relación de transmisión de 24.1.

Lo anterior permite conocer la velocidad que tomará el motor dependiendo del voltaje que se le aplique y tomando en cuenta que la máxima señal *PWM* que estará disponible para usar es 255. Se asume que mandando el 100% de la señal es posible manejar los 24 voltios que puede soportar el motor de corriente continua dándole así al motor una velocidad de 186 *RPM*, esto último después de pasar la *señal PWM* por la etapa de potencia que se explicará más adelante en el apartado de circuitos electrónicos. Igualmente si se desea que el motor tome un valor de 90 *RPM* se deberá aplicar el 50% de la señal, en otras palabras enviar una *señal de 127 PWM* mediante *arduino.* El programa en *processing* esta instrumentado tomando como bases los datos presentados anteriormente y solo es necesario introducir la velocidad deseada y no es necesario introducir directamente una *señal PWM* que se busque.

3.4.4 Avance por pulso.

Es necesario conocer la distancia que recorre el motor por cada pulso enviado desde el *encoder* hacia *arduino*. Sabiendo que el *encoder* utilizado es de tipo A y cuenta con 120 marcas en su disco; lo que quiere decir que por una vuelta que realice el motor (sin contar la relación de transmisión de 24:1) el *encoder* cambiara de estado 240 veces. Si se dividen los

360 grados de un círculo entre 240 se tendrá como resultado 1.5 grados, que es la distancia en grados que avanza el motor por un pulso del *encoder.* No hay que olvidar que el motor cuenta con una relación de transmisión; así que los cálculos deben de reflejar esto, Si se toman los 240 cambios de estados del *encoder* y se multiplica por 24 (debido a la relación de transmisión) es resultado será 5760 pulsos por una revolución del motor. Al dividir 360 grados entre 5760 cambios de estado del *encoder* se obtiene 0.0625 grados, que es la distancia que avanza el motor por un cambio de estado del *encoder*.

Es posible multiplicar 0.0625 por 16 para así convertirlo en un grado y no tener una fracción, al final se llega a la igualdad

1 grado = 16 cambios de estado del *encoder*.

Tomando en cuenta la circunferencia del eje del motor que es 6.0475 centímetros podemos calcular la longitud de avance del motor por cada grado que éste gire; dividiendo 6.0475 entre 360 grados se obtiene que el avance en centímetros por grado del motor es de 0.016798 centímetros.

1 grado = 0.016798 centímetros

Es posible igualar las 2 relaciones obteniendo

0.016798 centímetros = 16 cambios de estado el *encoder*

Finalmente es posible dividir 0.016798 centímetros entre 16 obteniendo 0.001049875 centímetros por cada cambio de estado registrado por el *encoder*.

3.5 Circuitos eléctricos/ electrónicos.

Al tener el programa en *arduino* y *processing* funcionado solo faltaría contar con una circuitería que garantice el funcionamiento del motor en un modo óptimo, para esto se ha diseñado el circuito de la figura 3.24.

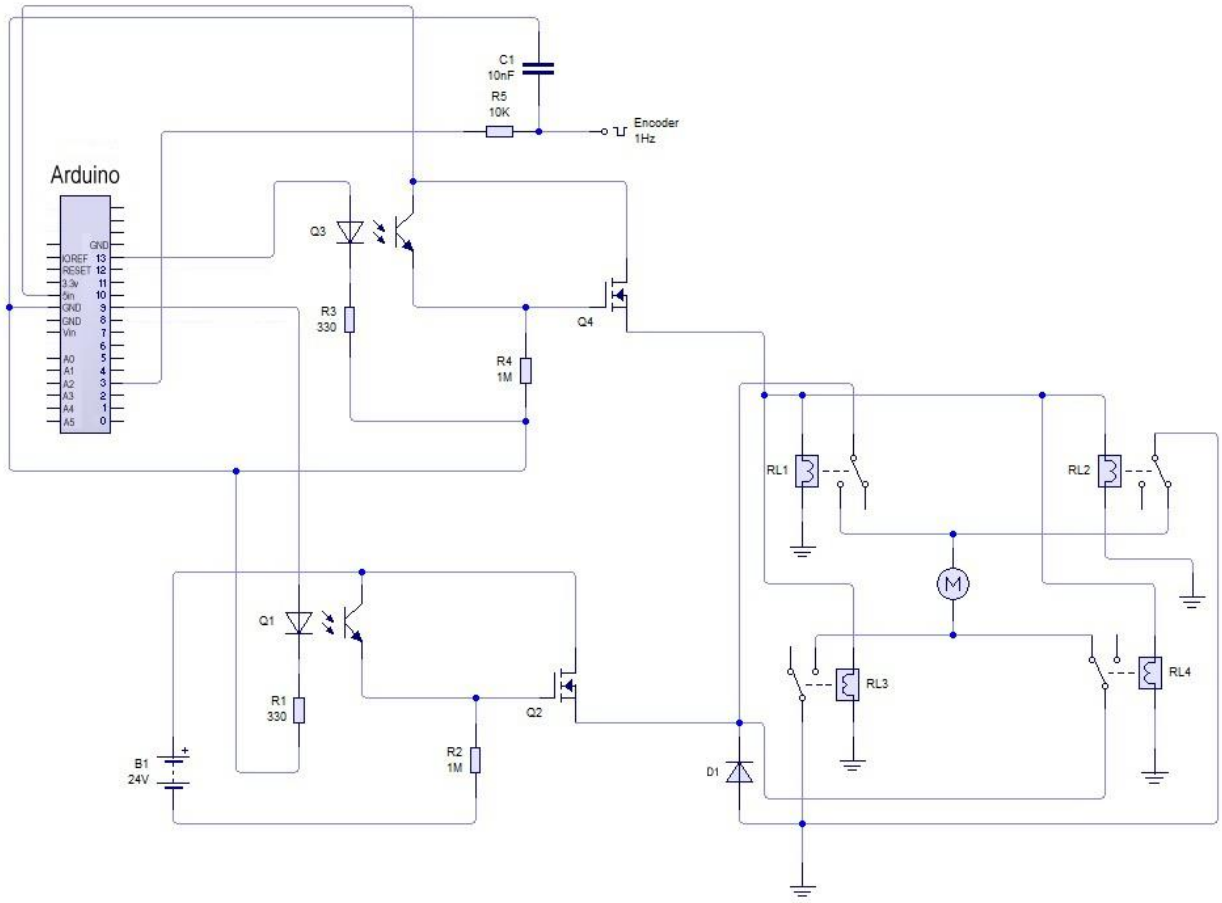

Figura 3.24. Circuito utilizado en el sistema de control.

El funcionamiento del circuito de la figura 3.24 utiliza un *puente H* para el control de giro del motor, además de una etapa de potencia en conjunto con la *señal PWM* generada por *arduino* para controlar la velocidad del motor y una etapa de potencia adicional para aumentar el pulso enviado por *arduino* y que se encargará de accionar las bobinas del *puente H*. En la figura 3.24, se puede observar las entradas y salidas utilizadas en *arduino*, al ver las salidas utilizadas, se puede apreciar cómo es utilizado el *pin digital numero 13* como salida para accionar el *puente H*, esto último pasando por una etapa de potencia que tratará la señal del *pin 13* para que tenga suficiente potencia para conmutar los relevadores que trabajan a *5 volts*.

Tomando en cuenta que el *pin 9* se puede usar como salida analógica de tipo *PWM*, será utilizado para alimentar la etapa de potencia que controla la velocidad del motor. La salida de esta etapa de potencia pasara por el *Puente H*. Para que esta sección de circuito

funcione, es necesario que trabaje a *24 volts*, por lo que será necesaria una fuente externa. Al final, el *pin digital número 3* se utilizó como entrada para medir la señal de pulso enviada por el *encoder* del motor, dicha señal será tratada por un circuito que constará de un capacitor y una resistencia, la función de este pequeño circuito será limpiar la señal y hacerla más legible para *arduino*. Combinando los programas de *arduino* y *processing* junto con el circuito mencionado en la figura 3.24 se logra un sistema que además de funcionar correcta y eficientemente garantiza la seguridad del controlador.

3.5.1 Etapa de control

En este trabajo la etapa de control está conformada por un programa en *processing* en conjunto con un programa con *arduino*. El programa en *processing* realiza los cálculos, las condiciones y sirve como interface gráfica, mostrando todos los valores y sus representaciones. El programa en *arduino* solo mandará una variable para controlar la velocidad del motor, esta variable se encontrará en una de las salidas *PWM* de la placa *arduino*.

Arduino también mandará una señal booleana para controlar un *puente H*, este circuito tendrá como función poder controlar el sentido de giro del motor, así evitando que el motor siga girando al llegar a uno de los extremos del motor, o controlar de manera correcta la posición del motor.

3.5.2 Etapa de aislamiento.

En la etapa de aislamiento se encuentran los *optoacopladores* que protegerán al control del circuito en caso de algún problema. Para el correcto funcionamiento del sistema se hace pasar por la entrada *anodo* la señal de *arduino*, en otras palabras la *señal PWM*.

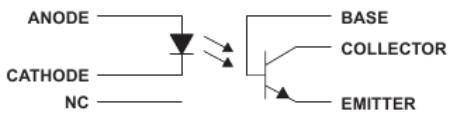

Figura 3.25. Circuito interno de un *optoacoplador*.

Siendo el *emisor* la salida de la etapa de aislamiento, es aquí donde se enviará la señal a la etapa de potencia para incrementar su voltaje.

3.5.3 Etapa de Potencia.

En la etapa de potencia se ha colocado un transistor (*tip 122)* que alimentarán a la carga, debido a que la señal de *arduino* no es suficiente para activar al motor de corriente directa. Una vez que la señal pasa por la etapa de aislamiento, sabiendo que se ha protegido el control, la señal llega a la etapa de potencia, más correctamente a la base del *tip 122*.

3.5.4 Puente H.

Uno de los aspectos de la tesis es poder controlar el sentido de giro del motor, para evitar que choque contra los extremos de la guía.

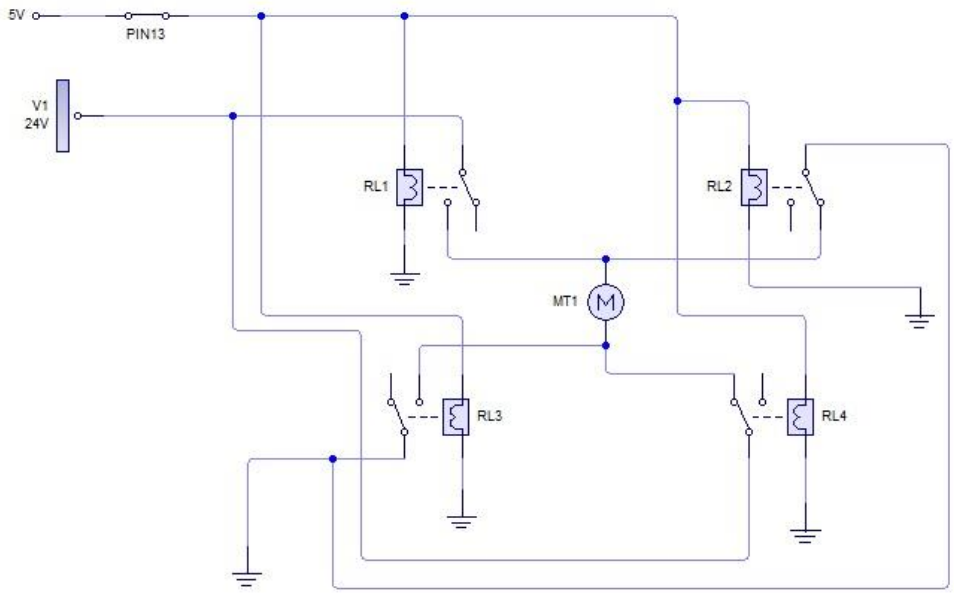

Figura 3.26. *Puente H* en *livewire.*

El circuito del *puente H* de la figura 3.26, está conformado por 4 relevadores que conmutan en pares. La primera pareja de relevadores son *RL1 y RL3*, que controlan el sentido de giro anti-horario y se encuentran en estado normalmente abierto, estos relevadores conmutan al mismo tiempo al recibir un pulso de origen del *pin 13* de *arduino*. Esté pulso es el

mismo que provoca el cambio de estado de los relevadores *RL2* y *RL4*, los cuales se encuentran normalmente cerrados.

3.5.5 Encoder

Para poder realizar una lectura correcta del *encoder* del motor, es necesario aplicar un circuito eliminador de rebote que tiene como tarea acondicionar la señal proveniente del *encoder*. El circuito estará construido a partir de una resistencia en serie a *ardiuno* y el *encoder*, además de un capacitor de 10nF a tierra. *Arduino* tomara lectura por medio de la entrada digital número 3, debido a que esta entrada cuenta con la función para leer interrupciones [25].

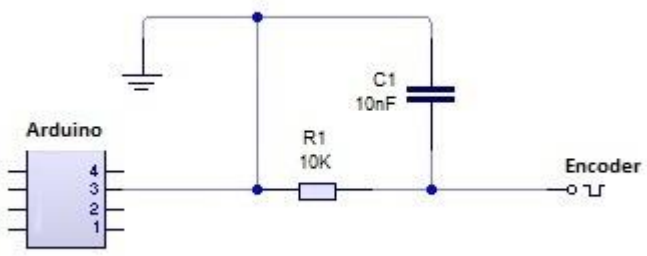

Figura 3.27. Circuito utilizado en la lectura del *encoder*.

3.5.6 Placas electrónicas

Por cuestiones de espacio, ya que se utilizará un gabinete para contener los circuitos, es necesario dividir el circuito en 3 PCBs, que conformaran todo el sistema electrónico antes mencionado. Se fabricaron 2 etapas de potencia y aislamiento y un *puente H* para controlar la dirección del motor. El circuito de la figura 3.28 es donde se encontrará la etapa de potencia y aislamiento para la *señal PWM*.

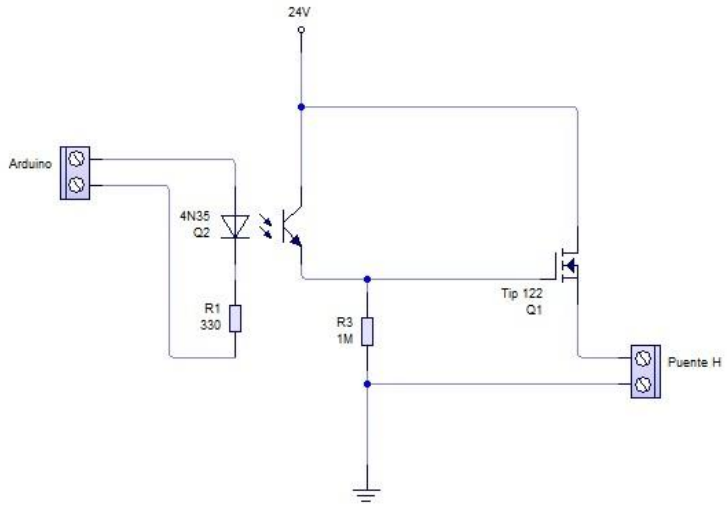

Figura 3.28. Circuito utilizado en una de las placas del sistema electrónico, que contiene una etapa de potencia, utilizada en la *señal PWM*.

Tal y como se muestra en la figura 3.28, el circuito empieza con la señal enviada por *arduino*, pasando por el componente *4N35* y el *tip 122*, para finalmente llegar al *puente H*. El siguiente circuito está conformado por una etapa de potencia trabajando a *5 volts*, que se encarga de conmutar las bobinas de los relevadores como se observa en la figura 3.29. A demás de contener el circuito usado para la lectura de la señal del *encoder*.

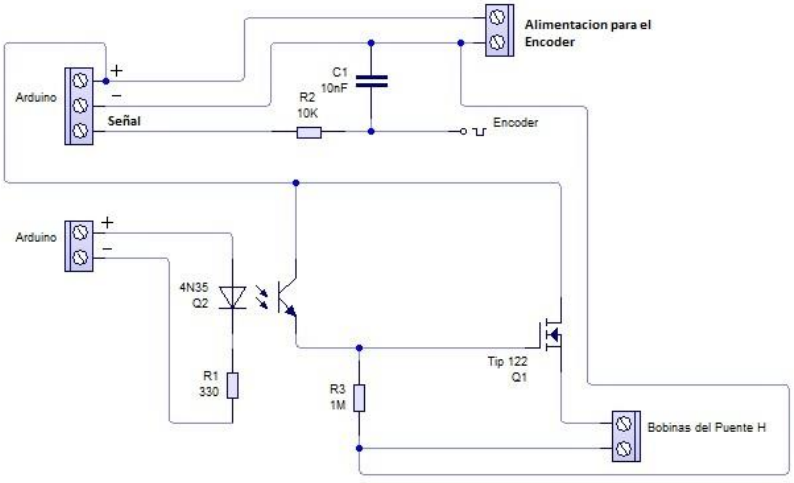

Figura 3.29. Circuito que contiene la etapa de potencia para conmutar las bobinas de los relevadores y el filtro utilizado en la lectura de la señal del *encoder*.

Finalmente la última etapa cuenta solamente con el *puente H*, como se observa en la figura 3.30, y los 2 circuitos antes vistos en las figuras 3.28 y 3.29 mandan las señales a este circuito, para después llegar al motor de corriente directa.

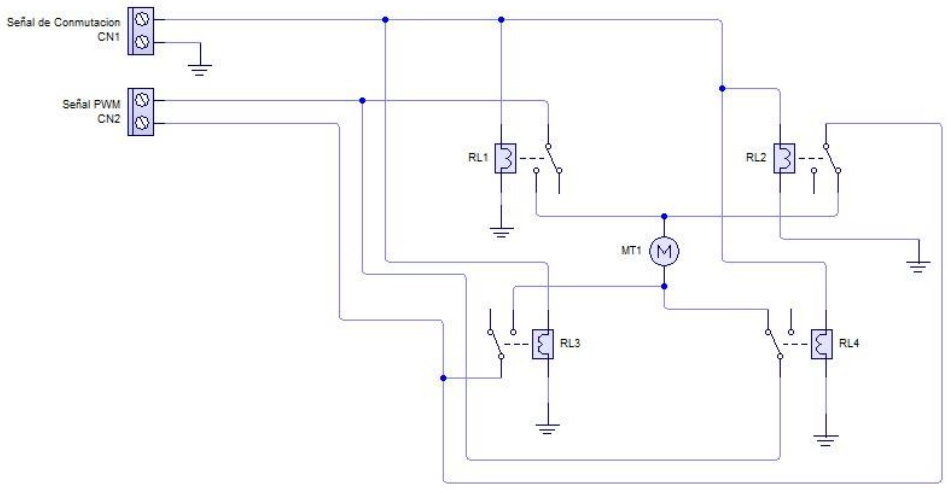

Figura 3.30. Circuito utilizado en el sistema para controlar la dirección de giro.

Capítulo 4

Resultados

4.1 Diseño de curva de aceleración

4.1.1 Velocidad.

Al igual que muchas secciones del código, la función de *velocidad* se encuentra dividida en una clase, esto para marcar una separación entre funciones. Para fines del diseño de la *curva de aceleración* es necesario saber cómo trabaja esta función en conjunto con la clase de *curva de aceleración*.

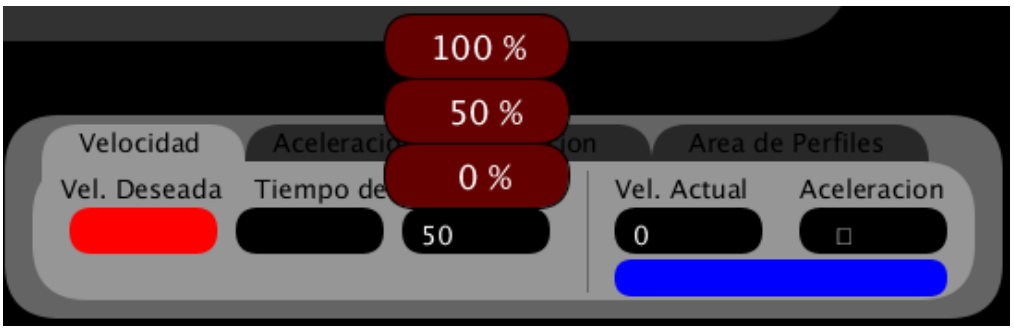

Figura 4.1. Diseño del control de la velocidad.

El valor con el que se trabaja se mostrará en un recuadro rojo, como en el caso de la figura 4.1 donde se opera con la casilla de *velocidad deseada.* Debido a que la clase de *curva de aceleración* necesita realizar los cálculos con una aceleración determinada, es necesario calcular dicha aceleración utilizando el tiempo y la velocidad que se introducen como variables. Estos últimos datos se envían a la clase de *curva de*

aceleración para empezar a trabajar. La ecuación que se utiliza es (2.3), que con base a una velocidad y el tiempo se puede estimar la aceleración [7].

En la parte inferior derecha se puede tener acceso al panel donde se introducen los datos deseados; este panel cuenta con 3 variables las cuales se modifican a voluntad, 2 de ellas se introducen por medio del teclado y la tercera que corresponde al *jerk*, el cual se seleccionará por medio de un selector del programa que ofrece las 3 opciones posibles, 100 %, 50 % y 0 %. Del lado derecho se encuentran 2 indicadores, el primero correspondiente a la *velocidad instantánea* de los cálculos y el segundo mostrando la aceleración con la que se moverá el motor.

Para seleccionar qué datos introducir se debe hacer *click* con el *mouse* en el área que corresponda al valor que se deseé manipular. Para esto se utilizó la función *mouseClicked()*, con la que se habilita utilizar el mouse en la interfaz. En el código se puede observar una variable llamada *DV*, cuya función es marcar la variable que en ese momento pude capturar datos del teclado. Si *DV* tiene un valor igual a 1 entonces el programa espera para capturar la *velocidad deseada* que el motor deberá tomar, en la figura 4.1 corresponde a la casilla en rojo. Si *DV* tiene un valor de 2 el programa puede capturar el *tiempo de aceleración* que se desee. Finalmente se encuentra la condición que activa el selector del *jerk*, dicha condición es activada por la variable *SelecJerk*, como se observa en la figura 4.1, al estar activa ésta variable se despliegan 3 botones correspondientes a las opciones habilitadas de porcentaje de *jerk* para escoger. Al seleccionar cualquiera de estas 3 opciones se cambia el valor de la variable *Jerk*, que como se ha dicho antes solo se ha habilitado para tener 50%, 100% y 0% de valor.

```
void mouseClicked(){
    if((mouseX>=575)&&(mouseX<=675)&&(mouseY>=520)&&(mouseY<=545)){
      DV = 1;}
    if((mouseX>=665)&&(mouseX<=765)&&(mouseY>=520)&&(mouseY<=545)){
      DV = 2;}
    if((mouseX>=755)&&(mouseX<=855)&&(mouseY>=520)&&(mouseY<=545)){
      SelecJerk = 1;}
      if(SelecJerk == 1){
```

```
            if((mouseX >= 745)&&(mouseX <= 845)&&(mouseY >= 485)&&(mouseY <=
510)){
            Jerk = 0;
            SelecJerk = 0;}
            if((mouseX >= 745)&&(mouseX <= 845)&&(mouseY >= 450)&&(mouseY <=
485)){
            Jerk = 50;
            SelecJerk = 0;}
           if((mouseX >= 745)&&(mouseX <= 845)&&(mouseY >= 415)&&(mouseY <= 450)){
            Jerk = 100;
            SelecJerk = 0;}
        } }}
```

Al observar la sección de código perteneciente a la función *mouseClicked()* se aprecian varias condiciones que son activadas mediante la posición en X, Y del *mouse*, todos los botones en la interfaz están delimitados por coordenadas X y Y dentro de la pantalla, tomando como ejemplo el botón de *velocidad deseada* que está colocado en la interfaz en un cuadro con la función *rect()* en las coordenadas 575 en X y 520 en Y, con un largo de 100 pixeles y un alto de 25 pixeles, es decir *rect(575,520,100,25).*

En el caso de los valores que deben ser introducidos mediante el teclado, se utilizó la función *keyPressed()* la cual habilita a la interfaz el uso del teclado. En el código presentado a continuación se encuentran la variable mencionada anteriormente, *DV*, que selecciona que variable es la que captura los datos, si la variable *ValorVel* que corresponde a la *velocidad deseada* o la variable *ValorTiempo* que corresponde al tiempo de aceleración. Ambas variables mencionadas anteriormente en este caso son tratadas como variables *String* debido a que han sido introducidas por el teclado. Al presionar la tecla de *enter* (\n, diagonal invertida y "*n*" son el equivalente a presionar la tecla *enter*) los valores de *ValorTiempo y ValorVel* son convertidos a valores *flotantes* y almacenados en variables del tipo *float* para que puedan ser usados en cálculos. Al mismo tiempo al presionar *enter* los valores *String* de éstas variables se eliminan dejándolas en blanco y la variable *Count* cambia su valor a 1, ésta variable sirve para iniciar el proceso que genera una *curva de aceleración*.

```
    void keyPressed(){
```

```
    if(key=='\n'){
      ValorVelInt = float(ValorVel);
      ValorVel = "";
      ValorTiempoInt = float(ValorTiempo);
      ValorTiempo = "";
      Count = 1;
    }
          //Introducción de la velocidad
          if(DV == 1){
            ValorVel = ValorVel+key;}
          //Introducción del tiempo
          if(DV == 2){
            ValorTiempo = ValorTiempo+key;}
}}
```

Durante el transcurso que la curva está siendo creada, los valores de velocidad y aceleración son el resultado de las ecuaciones se envían de nuevo a la clase de *velocidad* para ser graficados y mostrados en pantalla. Esto es:

```
  float GraficaV = map(VelocidadInt,0,185,250,50);        //Mapeo para ajustar la velocidad a la grafica

  float GraficaVA = map(AcIns,0,20,250,50);               //Mapeo para ajustar la aceleración a la grafica

  float GraficaVP = map(PosicionReal,0,100,250,50);       //Mapeo para ajustar la posición a la gráfica (cm)

  strokeWeight(1);
  stroke(255,0,0);                                  //Color de la grafica
  if(S >= 60){}else{
  point(map(S,0,60,401,700),GraficaV);                //Grafica de Velocidad
  point(map(S,0,60,51,350),GraficaVA);                 //Grafica de aceleración
  point(map(S,0,60,751,1050),GraficaVP); }             //Grafica de posición

 PortVel = map(VelocidadInt,0,185,0,255);
```

Por medio de la función *map* se ajusta el valor de velocidad para que la gráfica de velocidad se mantenga dentro de área que se le ha designado, debido a que la

velocidad corresponde al eje *Y* de la gráfica, se utilizan los valores de 250 a 50 pixeles, para que el valor de la velocidad empiece en 250 pixeles y conforme la velocidad aumente, la gráfica suba hasta alcanzar los 50 pixeles.

Para dibujar la gráfica se utilizó el comando *Point*, tomando como eje *X* al tiempo, éste tiempo es el que se utiliza para evaluar a las ecuaciones con las que se obtiene la velocidad y la aceleración. El tiempo será igualmente *mapeado* a 401 y 700 pixeles, para ajustarlo al área de la gráfica. Al finalizar dicho proceso y se alcance el valor máximo para el tiempo (700 pixeles o 60 segundos), se dibuja un cuadro negro en el área de la gráfica, lo cual significa que se podrá comenzar de nuevo el proceso para graficar.

4.1.2 Aceleración.

Para diseñar este panel se reusaron algunas funciones del control de velocidad, como la forma de seleccionar el valor que se desea introducir, dicho valor estará mostrado con un color rojo para poder distinguirlo. Al igual que en el párrafo anterior, aquí es posible modificar 2 variables con el teclado, la *aceleración* y el *tiempo*. En éste control, no es necesario introducir el *jerk*, debido a que se maneja de 0% en todos los caso, esto es para poder apreciar y controlar la aceleración en su totalidad. Si existiera una variable de *jerk* para modificar, la aceleración variaría durante toda la curva y solo se tendría el valor que se desea en unos pocos instantes antes de llegar al punto donde la velocidad es constante y no existe aceleración, de ésta forma la aceleración es constante en todo momento de la curva, como se muestra en la siguiente imagen:

Figura 4.2. Diseño del control de la aceleración.

En el lado derecho de panel se encuentran 2 indicadores, los cuales marcarán al igual que en el control de velocidad, la aceleración que se deseé y la velocidad a la que

se encontrará el motor del sistema. De la misma manera que en el control de velocidad, para seleccionar la variable que se deseé controlar se usa la función *mouseClicked()* y así alternar entre las variables habilitadas en éste control mediante otra variable llamada *DA*, dichas variables son la *aceleración deseada* y el *tiempo de aceleración.*

```
void mouseClicked(){
  if((mouseX>=610)&&(mouseX<=690)&&(mouseY>=520)&&(mouseY<=540)){
    DA = 1;
   }
  if((mouseX>=740)&&(mouseX<=820)&&(mouseY>=520)&&(mouseY<=540)){
    DA = 2;}}
```

Las variables que se pueden modificar son las de *aceleración deseada* y el *tiempo*. Al igual que en el *control de velocidad* los datos se introducen al programa una vez que las condiciones se han cumplido, es decir se hace *click* con el mouse en el área delimitada para cada variable según sus coordenadas, donde si *DA* toma el valor de 1 es posible modificar la aceleración, cuando toma 2 como valor es posible modificar el tiempo de la *curva de aceleración*. Por medio de la función *KeyPressed()* en conjunto con la variable *DA* es posible controlar en cual variable se introducirán los datos. Esta parte es idéntica a su versión del *control de velocidad*, al presionar *enter* (\n) se cumplirá la condición que permitirá mandar los datos a la clase: *CurvaAceleración* e iniciar el proceso de construcción de una *curva de aceleración*.

```
void keyPressed(){
  if(key=='\n'){
    ValorAcInt = int(ValorAc);
    ValorAc = "";
    ValorTiempoInt = int(ValorTiempo);
    ValorTiempo = "";
    Count = 1;
    PortVel = 0;}
      //Introducción de velocidad inicial
      if(DA == 1){
        ValorAc = ValorAc+key;
      }
      //Introducción de velocidad final
      if(DA == 2){
```

```
ValorTiempo = ValorTiempo+key;}}
```

Después que se ha construido la *curva de aceleración* y los valores han sido enviados para graficarse, el siguiente paso será *mapear* los valores de *aceleración*, *velocidad* y *posición* obtenidos de la clase de *curva de aceleración* para ajustarlos al contenedor adecuado, y los valores dados serán igual que para el *control de velocidad* y al igual que antes el tiempo se ajusta para llegar a 60 segundos antes de reiniciarse como se muestra en el siguiente programa:

```
float GraficaA = map(AcIns,0,20,250,50);                    //Mapeo para ajustar la aceleración a la grafica
float GraficaV = map(VelocidadInt,0,185,250,50);    //Mapeo para ajustar la velocidad a la grafica
float GraficaVP = map(PosicionReal,0,100,250,50);      //Mapeo para ajustar la posición a la grafica

strokeWeight(1);
stroke(255,0,0);                              //Color de la grafica
if(S >= 60){}else{
point(map(S,0,60,51,350),GraficaA);              //Grafica de aceleración
point(map(S,0,60,401,700),GraficaV);              //Grafica de velocidad
point(map(S,0,60,751,1050),GraficaVP);}          //Grafica de posición

PortVel = map(VelocidadInt,0,25,0,15);
```

4.1.3 Posición.

Al igual que en las funciones anteriores, este panel de control retoma parte de la estructura de las funciones anteriores, ahora enfocadas en el control de posición del

motor. En este caso se introducirá la distancia que se desea recorrer y el tiempo en el que se desea que se alcance dicha distancia. Como se muestra en la siguiente imagen.

Figura 4.3. Diseño del control de la posición.

Esta función es diferente de las 2 anteriores debido a que se construirán 2 *curvas de aceleración* en vez de una. La primera representa el arranque del motor y su función es acelerar al motor. La segunda parte es la desaceleración del motor, su función es detener al motor con un movimiento suave y controlado. Al igual que antes, el valor con el que se trabaja se mostrará en un recuadro rojo, como es en el caso de la figura 4.3 donde se trabaja con la casilla de *distancia.* Pero antes de generar la *curva de aceleración* se necesitan realizar varios cálculos con los que utilizando la distancia y el tiempo el programa calcula la velocidad necesaria para cumplir con las especificaciones dadas. Al contar con la velocidad se calculará la aceleración máxima que tendrá la curva, al igual que en la *función de velocidad*, para esto utilizará la siguiente fórmula.

$$Vf = \frac{Pos\,(3t_1+6t_2+3t_3)}{t^3} \tag{3.14}$$

Teniendo la *velocidad final* como dato el siguiente paso es calcular la aceleración de la curva mediante (2.3). En el lado derecho del control se muestran los indicadores de éste, en el lado izquierdo se encuentran los controladores cuyo trabajo es capturar por medio del teclado la posición y el tiempo. La función *mouseClicked()* será de ayuda para seleccionar la variable que se deseé modificar, donde la variable *DP* decide qué valor puede ser modificado entre *distancia* y *tiempo*.

```
void mouseClicked(){
    if((mouseX>=575)&&(mouseX<=675)&&(mouseY>=520)&&(mouseY<=545)){
      DP = 1;
    }
    if((mouseX>=665)&&(mouseX<=765)&&(mouseY>=520)&&(mouseY<=545)){
      DP = 2;
    }}
```

Al igual que en el *control de aceleracion* los datos se introducen al programa una vez que las condiciones se han cumplido, es decir se hace *click* con el mouse en el área establecida para cada variable según sus coordenadas, donde si *DP* toma el valor de 1 es posible modificar la distancia que avanzará el motor, cuando toma 2 como valor es posible modificar el tiempo de la *curva de aceleración*. Por medio de la función *KeyPressed()* es posible introducir datos por el teclado, esto en conjunto con la variable *DA* hace posible controlar en cual variable se introducirán los datos. Ésta parte es idéntica a su versión del *control de aceleracion*, al presionar *enter* (\n) se cumplirá la condición que permitirá mandar los datos a la clase de *CurvaAceleración*.

```
void keyPressed(){
  if(CPVA == 1){//Condición
      if(key=='\n'){
        ValorDisInt = int(ValorDis);
        ValorDis = "";
        ValorTiempoPInt = int(ValorTiempoP);
        ValorTiempoP = "";
        Count = 1;
      }
    //Intrroduccion de "Distancia"
    if(DP == 1){
        ValorDis = ValorDis+key;
      }
    //Intrroduccion de "Tiempo"
    if(DP == 2){
        ValorTiempoP = ValorTiempoP+key;
      }}}
```

4.1.4 Curva de aceleración.

Una vez que las ecuaciones han sido probadas y funcionan correctamente bajo todos los casos posibles, es momento de estructurar la sección de código que generará la

curva de aceleración. Para lo anterior se desarrolló un algoritmo que realiza los pasos necesarios para diseñar dicha curva. Esta función es la más importante del programa debido a que aquí se realizan la mayor parte de los cálculos que se necesitarán para trabajar. Éste algoritmo deberá realizar una serie de tareas antes de empezar a evaluar a las ecuaciones (3.3), (3.6) y (3.11); El primer paso será definir los tiempo en que se evaluará cada ecuación, el segundo paso es iniciar el cronometro, que actuará como la variable del tiempo en las ecuaciones, el tercer paso será marcar las condiciones que controlan que ecuación se reproducirá en determinado momento para finalmente empezar con la evaluación de éstas.

4.1.4.1 Tiempos de aceleración

El primer paso del algoritmo es definir los tiempos que durará cada sección de la curva, incremento, decremento y aceleración constante. Para esto se utilizan condiciones *if* que actúan dependiendo del valor de *jerk* o *empuje* que se deseé utilizar. Para uso de éste programa se limitó a solo 3 opciones de *jerk*, 100, 50 y 0, estos valores expresados en porcentaje como se muestra:

```
If (Jerk == 100){//---Si Jerk es 100----------//
t1 = Tiempo/2;                    //Calculo del tiempo 1
t2 = 0;                           //Calculo del tiempo 2
t3 = t1;}                         //Calculo del tiempo 3

if (Jerk == 0){//---Si Jerk es 0----------//
t2 = Tiempo;                      //Calculo del tiempo 2
t1=0;                             //Calculo del tiempo 1
t3=0;}                            //Calculo del tiempo 3

if (Jerk == 50){  //-Si Jerk es 50 % ----------//
t2 = Tiempo/2;                    //Calculo del tiempo 2
t1 = t2/2;                        //Calculo del tiempo 1
t3 = t1;}                         //Calculo del tiempo 3
```

Como se observa en el segmento de código anterior, el procedimiento a realizar varía dependiendo del valor de *jerk* introducido; si el *jerk* tiene un valor de 100% el tiempo 1 y 3 (*t1 y t3*) serán expresados como la mitad del tiempo total de la curva y el tiempo 2 (*t2*) que corresponde a aceleración constante será igual a 0, si el *jerk* toma un valor de 0% el tiempo 2 abarcará el total del tiempo del que se dispone, finalmente si se escoge 50% de *jerk* la mitad del valor total de tiempo será tomada por el tiempo 2 (*t2*) y

la otra mitad del tiempo se divide entre 2 para darle una mitad al tiempo 1 (*t1*) y otra al tiempo 3 (*t3*).

4.1.4.2 Inicio del cronometro

El siguiente paso es corto, se inicia el cronometro. En base a éste tiempo se registrará en una variable en este caso nombrada como *"m",* se obtendrán los demás valores de la curva de aceleración. Para crear el cronometro se hará uso del comando *millis (),* éste comando registra una cuenta de los milisegundos desde el momento en que ha empezado funcionar el programa. Por tanto si se guardara en una variable el tiempo actual hasta el momento que se necesite llevar una cuenta de tiempo, se podrá solamente restar dicha variable, *"m"* en este caso, al mismo comando *millis()* para obtener la cuenta de tiempo que se necesita. Por ejemplo, la variable *"m"* guarda el tiempo transcurrido hasta ese momento.

```
m = millis();
```

Al tener el tiempo guardado es necesario restar la variable *"m"* al comando *millis().*

```
rm = millis() - m;
S = rm/1000;
```

El resultado de esta resta no es más que la cuenta del tiempo a partir de cuándo la variable *"m"* registro su valor. En otras palabras el resultado de esta resta es un cronometro en milisegundos que empieza al realizar la igualdad de *m = millis()*. El segundo comando *millis()* deberá de estar colocado en un ciclo que se repita constantemente, para que así el tiempo se actualice y el cronometro pueda funcionar. Por el lado contrario la variable *"m"* debe solamente reproducirse una vez, debido a que su valor debe ser el mismo, o sea el tiempo en el que sea necesario que empiece la cuenta del tiempo. Al contar con el cronometro solo queda transformar los milisegundos en segundos por medio de una división entre 1000.

4.1.4.3 Control del tiempo

Cada segmento de la curva debe ser evaluado con su respectivo tiempo. Los tiempos calculados anteriormente *t1, t2 y t3,* son la duración de cada parte de la curva, pero no funcionan para evaluarla ya que son valores constantes, para realizar esto último se deben crear nuevas variables que adopten el valor del tiempo y que aumenten su valor conforme al cronómetro.

```
//Tiempo 1
  if((S >= 0)&&(S <= t1)){
     St1 = (millis() - m)/1000;
     m2 = millis();}

//Tiempo 2
   if((S >= t1)&&(S <= (t2+t1))){
      St2 = (millis() - m2)/1000;
      m3 = millis();}

//Tiempo 3
    if((S >= (t2+t1))&&(S <= (t3+t2+t1))){
       St3 =  ((millis() - m3)/1000);}
```

Para calcular el tiempo de la primera sección de la curva se utiliza una variable llamada *St1* que está dada por la resta del tiempo inicial del programa menos la variable "*m*" que es el tiempo en que se empezó con este proceso, al final dividido entre mil para transformar los milisegundos a segundos, por tanto, esta variable alcanza valores de 0 hasta el valor del tiempo del primer segmento, es decir *t1,* debido a la condición que limita que pase de este periodo de tiempo.

Con el tiempo siguiente el procedimiento es similar pero al finalizar el cálculo de la variable *St1* y tras alcanzar la condición de *S <=t1* se obtiene una variable con el valor del tiempo en que termina la cuenta del primer segmento, ésta variable será llamada *m2* y servirá para calcular *St2,* ahora con esta nueva variable el procedimiento se repite, pero las condiciones han cambiado, por lo que el tiempo variable se evalúa entre el tiempo *t1* y la suma de *t1* mas *t2*, este tiempo *St2* servirá para evaluar el segmento de aceleración contante.

Para terminar con este paso se procede a evaluar el segmento de decremento de aceleración, al igual que en el periodo pasado se crea una variable llamada *m3* que registra el tiempo en que termina el segundo segmento. Ahora este periodo se iniciará al

finalizar el segmento pasado pero concluirá al llegar al tiempo dado por la suma de *t1*, *t2* y *t3*. Cabe señalar que al mismo tiempo que los valores de *St1*, *St2* y *St3* se están generando también se está evaluando en las ecuaciones que formarán la curva de aceleración.

4.1.4.4 Evaluación de las ecuaciones

El último pasó será evaluar las ecuaciones con sus valores correspondientes. Lo que se necesita para evaluar las ecuaciones son los tiempos *t1*, *t2* y *t3* que marcaran el inicio y el fin de cada segmento y controlan en que momentos se aplica cada ecuación. Además serán necesarios los tiempos variables *St1*, *St2* y *St3* para que adopten los valores que se sustituirán en (3,3), (3.6) y (3.11).

```
//Incremento de aceleración
        if((S >= 0)&&(S <= t1)){
            V = (Amax*pow(St1,2)) / (2*t1);
            AcIns = ((Amax - Ao)*St1)/(t1 - to);}

//Aceleración constante
        if((S >= t1)&&(S <= (t2+t1))){
            V = (Amax * ( t1/2 + St2));
            AcIns = Amax;}

//Decremento de aceleración
         if(S >= (t2+t1)){
            V = Amax * ( t1/2 + t2 + St3 + pow(St3,2)/(2*(t3-t2)) );
            AcIns = (Amax *St3)/(t3 - t2)+ Amax;}
```

Para una correcta sincronización las condiciones de los tiempos deben ser las mismas que en el paso del *control del tiempo*, esto para que las partes equivalentes se reproduzcan al mismo tiempo y los valores concuerden con sus respectivas ecuaciones. De no ser así las ecuaciones no podrán evaluarse con sus valores de tiempo y no habrá resultado alguno.

Para poder reproducir los procesos necesarios se deben colocar en una estructura *switch* y agregar una variable que controle la secuencia de *casos*. Dicha variable será llamada *Count* y al final de cada proceso aumentará su valor en una unidad, esto para alternar entro los diferentes casos de la estructura. El caso número 3 es el más extenso de la estructura al contener las ecuaciones que se evaluarán, las variables del tiempo y el cronómetro del proceso.

```
switch(count) {

  case 1:
        If (Jerk == 100){//---Si Jerk es 100----------//
        t1 = Tiempo/2;                    //Calculo del tiempo 1
        t2 = 0;                           //Calculo del tiempo 2
        t3 = t1;}                         //Calculo del tiempo 3

        if (Jerk == 0){//---Si Jerk es 0----------//
        t2 = Tiempo;                      //Calculo del tiempo 2
        t1=0;                             //Calculo del tiempo 1
        t3=0;}                            //Calculo del tiempo 3

        if (Jerk == 50){  //-Si Jerk es 50 % ----------//
        t2 = Tiempo/2;                    //Calculo del tiempo 2
        t1 = t2/2;                        //Calculo del tiempo 1
        t3 = t1;}                         //Calculo del tiempo 3
        Count = Count + 1;                //Se suma 1 unidad a la
                                          variable "Count" y así
                                          avanzar al siguiente
                                          proceso
   break;

  case 2:
        m = millis();                     //Se guarda la variable "m"
        Count = Count + 1;
   break;

  case 3:
        rm = millis() - m;
        S = rm/1000;

        //Tiempo 1
          if((S >= 0)&&(S <= t1)){
             St1 = (millis() - m)/1000;
             m2 = millis();}


        //Tiempo 2
           if((S >= t1)&&(S <= (t2+t1))){
              St2 = (millis() - m2)/1000;
              m3 = millis();}

        //Tiempo 3
            if((S >= (t2+t1))&&(S <= (t3+t2+t1))){
               St3 =  ((millis() - m3)/1000);}
        Count = Count + 1;


//Incremento de aceleración
        if((S >= 0)&&(S <= t1)){
            V = (Amax*pow(St1,2)) / (2*t1);
            AcIns = ((Amax - Ao)*St1)/(t1 - to);}
//Aceleración constante
```

```
            if((S >= t1)&&(S <= (t2+t1))){
                V = (Amax * ( t1/2 + St2));
                AcIns = Amax;}
//Decremento de aceleración
             if(S >= (t2+t1)){
                V = Amax * ( t1/2 + t2 + St3 + pow(St3,2)/(2*(t3-t2)) );
                AcIns = (Amax *St3)/(t3 - t2)+ Amax;}
   break;
}
```

Durante éste proceso las ecuaciones guardan su valor en una variable llamada *V*, que guardará el valor de la *velocidad instantánea* en ese momento y posteriormente será graficado y mostrado en la interfaz. Éste valor es enviado a *arduino* por medio del *puerto serial*, donde *arduino* lo convertirá en una *señal PWM*. En el programa se puede observar debajo de la ecuación para la velocidad una ecuación llamada *AcIns* o *aceleración instantánea,* ésta ecuación, usando los valores antes mencionados calcula la aceleración del sistema en cualquier momento para posteriormente mandarlo a graficar y mostrarlo en la interfaz al igual que la velocidad calculada anteriormente. Con estos 4 pasos es posible diseñar un algoritmo que fabrique las curvas que se necesiten. Éste algoritmo se encuentra separado en forma de una clase de *procesing*, y solo es llamado cuando se introducen los valores en los diferentes tipos de controles con los que cuenta el programa.

4.2 Estructura en processing

Processing cuenta con una clase principal denominada *Interface_v10*, es donde se manda a llamar a las demás clases, donde se puede acceder al menú del programa. Aquí se encuentra las opciones para alternar entre los diferentes tipos de controles con los que se cuenta, se envían los valores al *puerto serie* y se reciben los datos de *arduino* se encuentra el método que inicia la comunicación con *arduino,* entre otras funciones un poco menos significativas.

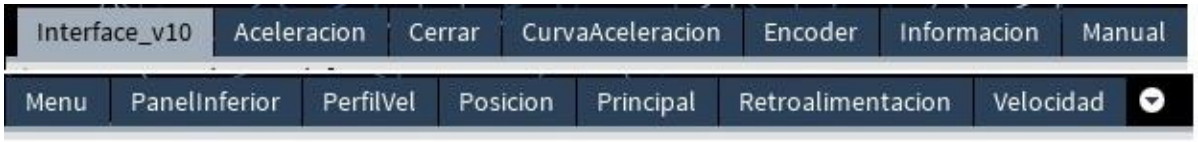

Figura 4.4. Clases utilizadas en la *interfaz*.

Ahora se hablará acerca de cómo *processing* maneja internamente sus datos. Como se observa en la figura 4.5 el programa inicia en la clase principal, *Interface_v10* de *processing*. Ahí se encuentra el botón del menú donde se podrá seleccionar la opción que se deseé, además de seleccionar el tipo de control que se necesite utilizar y modificar los valores deseados. También se puede observar los cambios que sufre el motor en el tiempo además de diferentes datos como revoluciones del motor y los datos que son enviados, recibidos desde *arudino* y algunos indicadores que muestran datos útiles como por ejemplo, si se logró con éxito el primero contacto de comunicación entre ambos programas. Todo lo anterior es posible verlo en la figura 4.6, donde se muestra la imagen de la interfaz al abrir el programa.

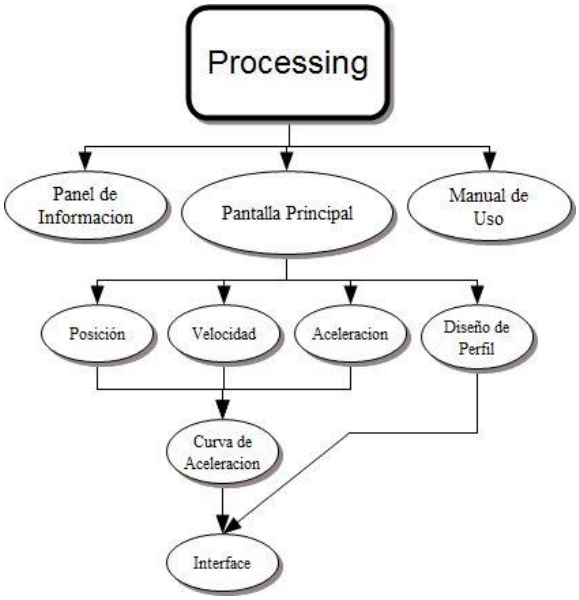

Figura 4.5. Diagrama de flujo de la estructura de las funciones de *processing*.

El objetivo principal de crear diferentes *clases* para cada función del programa es mantener una separación y orden entre las secciones del código, esto tomando en cuenta la función que desempeñan dentro del programa.

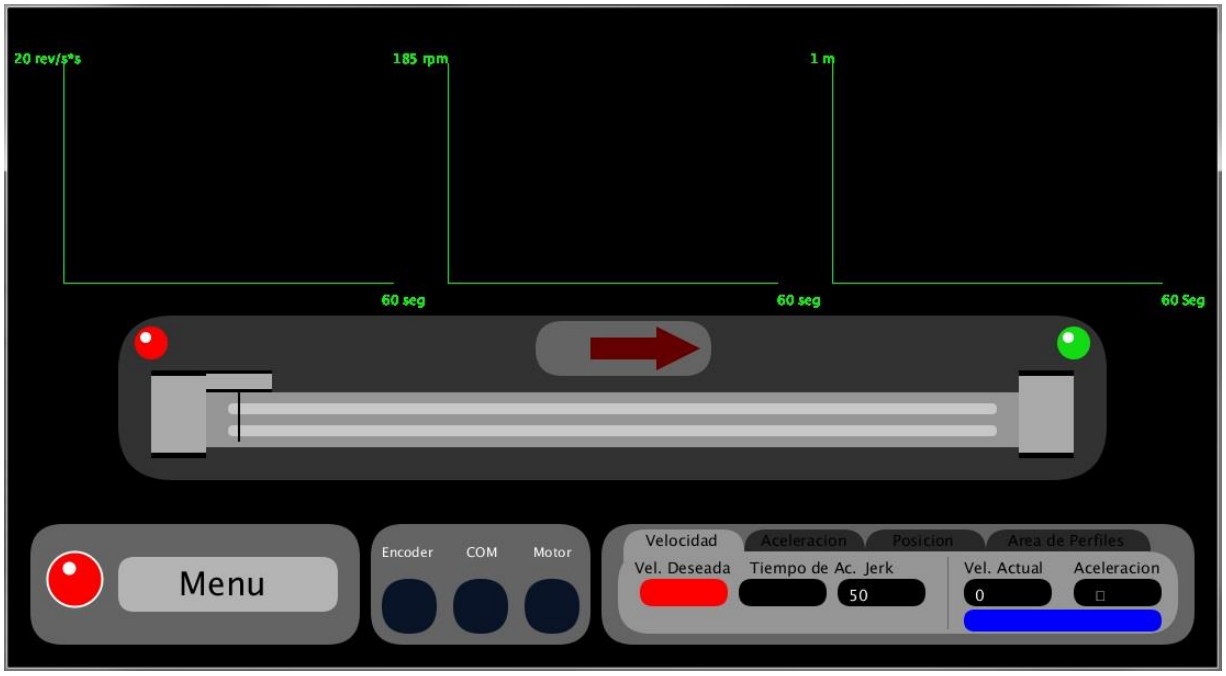

Figura 4.6. Vista principal del programa en *processing.*

4.2.1 Menú de opciones

Para crear los botones del menú se utilizó un método sencillo. Al tener el cuadro del menú abierto, dependiendo de las coordenadas en las que se encuentre el *mouse*; si está dentro de las coordenadas designadas para el botón de la pantalla principal y se presiona con el botón izquierdo de *mouse*, entonces la variable: *Opción* que es la que controla la pantalla que se muestra cambiara su valor al valor correspondiente de la pantalla principal, esto usando la función *mouseClicked().* Lo mismo para las demás pantallas con las que se cuenta.

```
void mouseClicked(){

    if((mouseX >= 420)&&(mouseX <= 680)&&(mouseY >= 120)&&(mouseY <= 170)){
//Pantalla principal
    Opción = 1;
    Menu = 0;
    background(0);
    }

    if((mouseX >= 420)&&(mouseX <= 680)&&(mouseY >= 175)&&(mouseY <= 225)){
//Manual de uso
    Opcion = 2;
    Menu = 0;
    }

    if((mouseX >= 420)&&(mouseX <= 680)&&(mouseY >= 230)&&(mouseY <= 280)){
```

```
//Selecciona el panel de información
    Opcion = 3;
    Menu = 0;
    }

    if((mouseX >= 420)&&(mouseX <= 680)&&(mouseY >= 285)&&(mouseY <= 335)){
 //Salir
    exit();
    }}
```

Dependiendo de las coordenadas donde se presione el *mouse* es el valor que se obtendrá; cada valor está ligado a alguna clase del programa. Cabe señalar que el menú debe estar abierto para que esta sección del código funcione.

4.2.1.1 Pantalla principal

La pantalla principal de la interfaz muestra varios de los aspectos principales del trabajo. La animación se mueve de manera que simule el movimiento del motor. El programa utiliza una variable denominada *PosicionReal*, localizada en la clase de *Encoder*, dicha variable es el resultado de la suma de otras 2 variables llamadas *Avance* y *Reversa*, las cuales convierten los pulsos enviados desde *Arduino* por el *encoder* a distancia en centimetros, cada una lleva una cuenta de los pulsos que se registrán en cada dirección del motor, al final se convierten a posición, lo anterior se analizará más adelante en el apartado del *encoder*. La variable de *PosicionReal* se encuentra *mapeada* para ajustarla a la guía de metal, la cual mide 94.92 cm, descontando los topes de goma de los extremos.

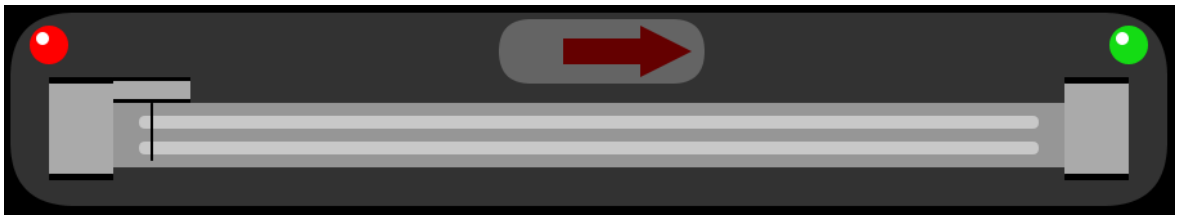

Figura 4.7. Imagen de la guía de metal.

Ésta sección cuenta con 2 indicadores que simulan la forma de *leds* a los extremos, dichos indicadores se muestran en rojo cuando el motor se encuentra topando con uno de los extremos de la guía. En la figura 4.7 el *led* izquierdo brilla en un color rojo, lo cual indica que el motor no puede seguir retrocediendo hacia la izquierda. Al

igual que todos los procesos y funciones del programa, dichos *leds* son controlador por 2 variables que se activan cuando la variable *PosicionReal* es igual o menor a 0 e igual o mayor que 94.92, en otras palabras, cuando se ha detectado que el motor alcanza los extremos de la guía metálica.

En la parte central del cuadro se apreciar una flecha que cumple con 2 funciones: ser un botón para controlar la dirección del motor y ser un indicador que muestre en qué dirección avanza el motor.

```
if((mouseX >= 480)&&(mouseX <= 640)&&(mouseY >= 285)&&(mouseY <= 335)){
   if(Sentido == 1) {
     Sentido = 0;
      }else{
      Sentido = 1;}}
```

Para ahorrar espacio y por cuestiones de estética se diseñó un botón que cumpla con la función de alternar entre los dos estados del motor (avance y reversa), así se evita insertar 2 botones para cada dirección del motor. Al observar en su código la variable *Sentido* es la encargada de dicha tarea, ésta sección de código se encuentra dentro de la función de *mouseClicked()* para poder reconocer al *mouse* de la computadora, esta valor es enviado a *arduino* y posteriormente enviado a la etapa de potencia para controlar un circuito puente H. Hay que señalar que el valor 0 establece el avance del motor y el valor 1 la reversa.

Figura 4.8. Indicador y botón del control de dirección.

En la parte superior de la pantalla principal se encuentran las gráficas donde se puede observar la velocidad, la posición y la aceleración del motor en cualquier momento en que se encuentre en movimiento, éstas graficas están ajustadas para que trabajen en un tiempo de 60 segundos, al terminar el tiempo la gráfica se detiene y es posible reiniciar para empezar a trabajar de nuevo. Para mostrar valores en tiempo real se opta por usar las variables *VelocidadInt, AcIns, PosicionReal* para insertarlas en las gráficas. La variable *VelocidadInt* será la variable que represente la velocidad

instantánea del motor, ésta variable no es más que la igualdad de la variable *V* en el diseño de la curva de aceleración, esto se aprecia al observar dicha sección de código, igualmente *AcIns* también puede ser encontrada en la sección de código que diseña la *curva de aceleración*, ésta variable se calcula mediante ecuaciones y representa la aceleración instantánea del sistema, finalmente la variable *PosicionReal* es calculada de la lectura tomada por el *encoder*.

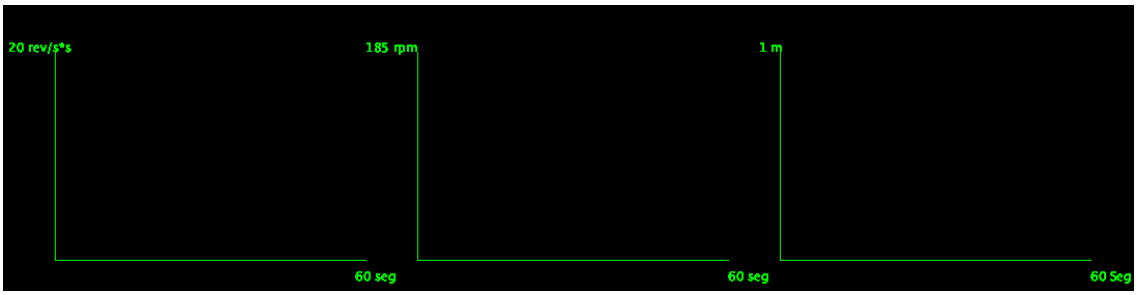

Figura 4.9. Graficas de las propiedades del motor, *aceleración, velocidad y posicion*.

4.2.1.2 Información

En la opción nombrada como *información* se podrá encontrar un poco de los objetivos básicos del presente trabajo, título de la tesis, el propósito de este trabajo y el asesor de tesis.

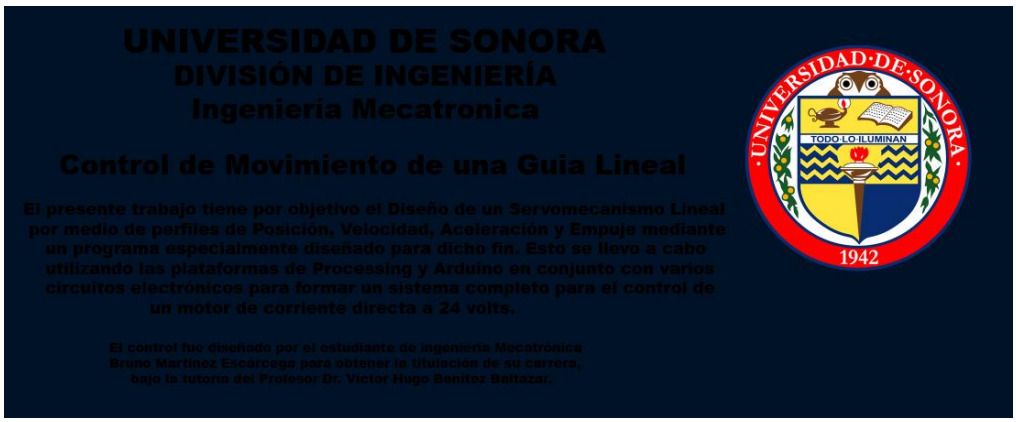

Figura 4.10. Imagen que muestra una breve introducción a la interfaz.

4.2.1.3 Manual de uso

El objetivo de este pequeño manual es el de proporcionar información acerca de las partes y el uso de las funciones con las que cuenta el programa; Éstos manuales

hablan acerca de las funciones presentadas por el programa, al igual que menciona cuales son las variables importantes de cada tipo de control.

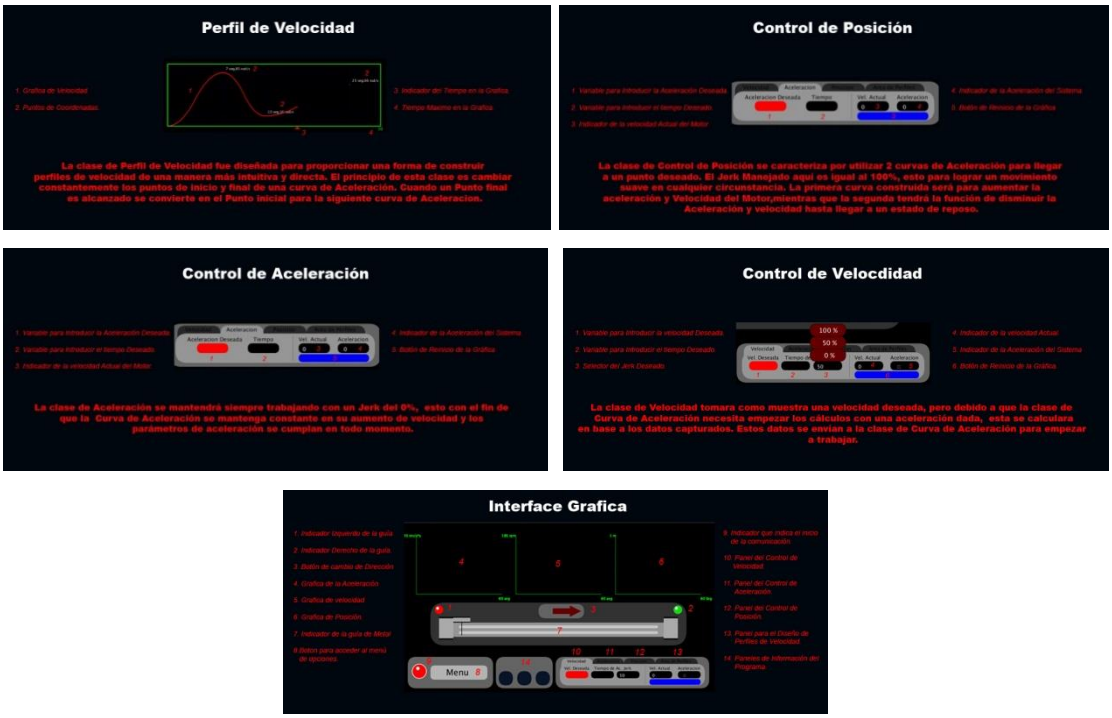

Figura 4.11. Imágenes de los manuales presentados en el programa.

4.2.1.4 Salir

Si el programa es cerrado de una manera repentina es probable que el motor mantenga su velocidad con el último valor registrado y con la última dirección señalada, esto aún con el programa cerrado, para compensar esto se creó un proceso ligado a la función de *Salir* cuya función es terminar con el movimiento del motor en caso de un cierre repentino.

Figura 4.12. Aviso de cierre del programa.

El primer paso para este proceso será detener momentáneamente el flujo de datos de *processing* a *arduino,* ésto para detener el motor y poder pasar al segundo paso, el cual consiste en llamar a la clase de control de posición para devolver al motor a su punto de inicio, una vez el motor se encuentre en el inicio, el programa comenzaré con el paso final de este proceso, con lo cual el programa cancelará todo valor enviado o recibido entre *processing* y *arduino,* al hacer esto el motor detendrá su marcha y el programa podrá cerrarse finalmente.

```
switch(Salir){
 case 0:
            Salir = 1;
 break;
 case 1:
            fill(255,0,0);
            textSize(18);
            text("Cancelando envió de datos....",400,160);
            delay(2000);
            Port.write(0);
            Sentido = 255;
            Salir = 2;
 break;
 case 2:
            fill(255,0,0);
            textSize(18);
            text("Cancelando envio de datos....",400,160);
            text("Regreso al Punto de Origen....",400,190);
            CPVA = 1;                          //Se selecciona el Control de Posicion
            DistanciaRegreso = PosicionReal;          //Calculo de la distancia de regreso
            text("Regresar "+int(DistanciaRegreso)+" cm",430,220);
            delay(2000);
            ValorDisInt = int(DistanciaRegreso);
            ValorTiempoPInt = 20;
           Count = 1;
            if(TopeIzq == 1){
              Salir = 3;}
 break;
 case 3:
            fill(255,0,0);
            textSize(18);
            text("Cancelando envio de datos....",400,160);
            text("Regreso al punto de origen....",400,190);
            text("Cerrando puerto serie...",400,220);
            delay(2000);
            Port.write(0);
          Salir = 4;
 break;
 case 4:
            fill(255,0,0);
            textSize(18);
            text("Cancelando envió de datos....",400,160);
            text("Regreso al punto de origen....",400,190);
            text("Cerrando puerto serie...",400,220);
            delay(2000);
            exit();
```

```
break;}
```

4.2.2 Diseño de perfiles de velocidad

El objetivo del presente trabajo es mostrar de manera gráfica los aspectos que rigen a un motor de corriente directa, el problema de las funciones previamente mostradas es que al introducir los datos deja de ser un entorno 100% gráfico, esto debido al uso del teclado para introducir los datos. Para compensar esto, se creó una función llamada *PerfilVel* (perfil de velocidad) la cual permite crear perfiles de velocidad de una manera totalmente grafica e intuitiva, mostrando los valores usados para el diseño.

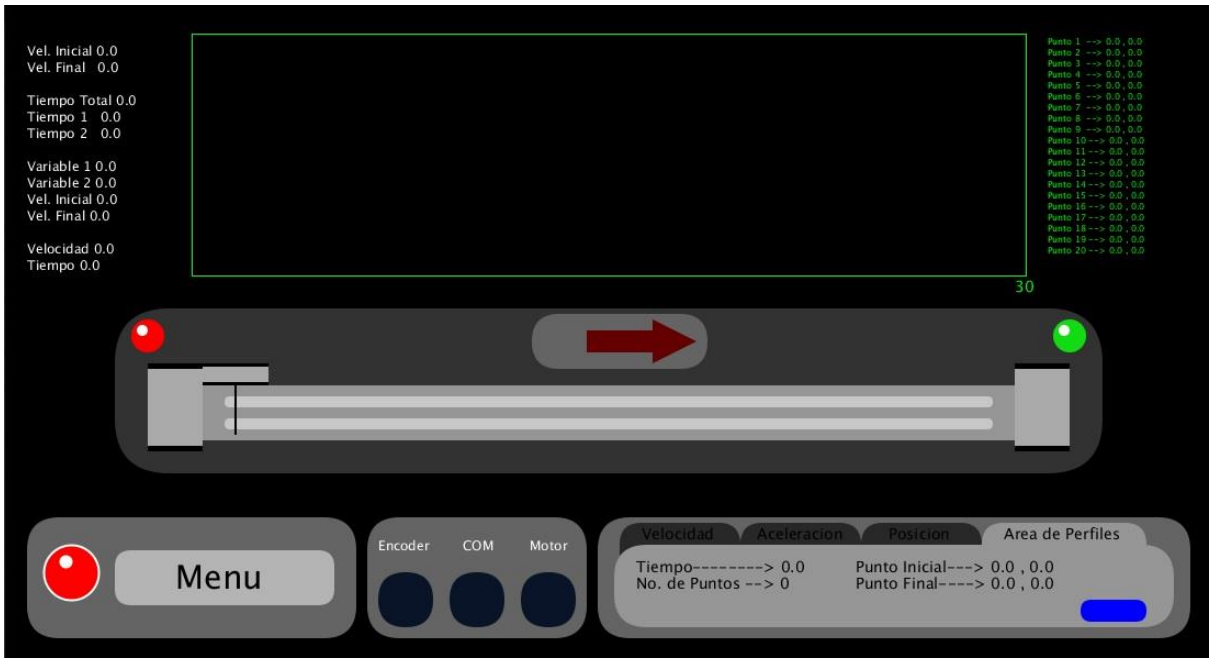

Figura 4.12. Pantalla del área para el diseño de perfiles.

El diseño de ésta función es complicado debido a que los valores que se utilizan a lo largo del perfil son capturados por la máquina antes de que empiece a graficar, a diferencia de los controles anteriores donde se introducen valores solamente de una *curva de aceleración* de un perfil. En principio la función de *PerfilVel* consiste en marcar puntos con el *mouse* dentro del recuadro designado para esta tarea.

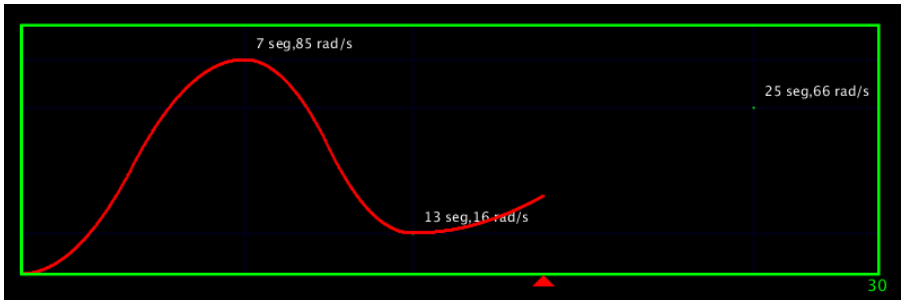

Figura 4.13. Área para el diseño de perfiles.

Una vez que se han marcado los puntos que se deseen, empezará a correr el tiempo y la gráfica empezará a formarse poco a poco. Un indicador en forma triangular en la parte inferior mostrará el avance del tiempo. A diferencia de las otras gráficas ésta se encuentra ajustada para durar 30 segundos trabajando antes de terminar. Otro aspecto importante es que esta función cuenta con un seguro que evita que se introduzcan valores de tiempo menores al último valor introducido, gracias a esto los puntos siempre serán mayores al anterior, lo que evitará errores al momento de graficar.

El código para esta función es uno de los más elaborados del programa, esto se debe a que constantemente cambia su punto inicial y punto final para adaptarse a las *curvas de aceleración* que forman al *perfil de velocidad*. Cuando un nuevo punto es marcado sus valores *X* y *Y* son guardados en arreglos, *Xp* para *X* y *Yp* para *Y*.

```
if(Seg >= Xpfin){
      Xpin = Xp[CPp];    Ypin = Yp[CPp];
      Xpfin = Xp[CPp+1]; Ypfin = Yp[CPp+1];
      //Aceleraciones
        Apin = Apfin;

      CPp = CPp+1;
  }
```

Como se observa en la sección de código se puede encontrar una gran cantidad de variables que constituyen el sistema de control de puntos que es utilizado para decidir qué valores son los que se introducirán en las ecuaciones, aquí es donde se determina qué punto será evaluado como inicial y que punto será evaluado como final. La variable *Seg* es el tiempo en segundos. Las primeras variables que se puede observar además de la variable del tiempo es *Xpfin* y *Xpin* estas 2 variables son las que serán introducidas en las ecuaciones, y marcan el tiempo final e inicial de cada curva, estas 2

variables cambian constantemente sus valores. Luego se observa la variable *CPp*, dicha variable es la cuenta de puntos con los que el perfil debe trabajar. Las coordenadas introducidas se guardan en 2 arreglos de 20 variables cada uno, cada arreglo esta creado para guardar el tiempo en caso del arreglo *Xp*, y la velocidad en el caso del arreglo *Yp*.

Figura 4.14. Puntos guardados en arreglos *Xp* y *Yp*.

Todo el proceso de actualización de los puntos sucede cuando el tiempo, *Seg* alcanza al valor de *Xpfin*, que es el final de cada curva de aceleración. Una vez se cumpla esa condición empieza el proceso. El primer paso es igualar los valores iniciales de *Xpin* y *Ypin* a los valores de los arreglos *Xp* y *Yp*. El segundo paso será igualar los valores finales de la curva al siguiente valor del arreglo, para eso se suma 1 a la variable *CPp*, ésta variable controla los valores utilizados, siendo siempre *CPp* para los valores iniciales y *CPp+1* para los valores finales, de esta manera siempre se tendrá 2 valores continuos evaluando a las ecuaciones. Es así como siempre se actualizarán los valores y nunca se utilizaran 2 valores iguales para la curva, al final de este proceso la aceleración final de la curva pasara a ser la aceleración inicial y se calculará una nueva aceleración final. El paso final es agregar una unidad a la variable *CPp (control de puntos del perfil)* de esta manera cuando el proceso se repita se trabajará con 2 nuevos valores, los valores siguientes.

Para hacer los cálculos que fabricarán la *curva de aceleración* para cada sección del *perfil de velocidad* no se utiliza la clase de *CurvaAceleracion* como ha sido el caso anteriormente, para esta función se utiliza su propio algoritmo que realiza la misma

función, solo adaptado para éstas necesidades. Debido al uso de variables que se actualizan constantemente no era posible realizar el mismo procedimiento que fue utilizado en las funciones de *velocidad*, *control* y *aceleración*, el código de la clase *CurvaAceleracion* necesitaba demasiados cambios para adaptarlos a la función que se está presentando, tomando esto en cuenta se optó por diseñar un algoritmo para esta tarea y que está incluido en la clase de *Perfilvel*, dicho algoritmo iniciará su proceso al presionar la tecla *enter*.

El primer paso para este nuevo algoritmo consiste en iniciar la cuenta del *timer*, esto usando la variable *Seg*, además de esto se calcula la diferencia entre el punto inicial y el punto final de la *curva de aceleración* que se va a generar y se guarda en la variable *Tdp* (Tiempo diferencial entre puntos), ésta variable es el tiempo que dura la curva de aceleración actual.

```
Seg = (T - mp)/1000;
Tdp = Xpfin - Xpin;
```

Conociendo el tiempo es posible realizar la distribución de las secciones de la *curva de aceleración*, es importante resaltar que el programa esta acondicionado para generar los *perfiles de velocidad* utilizando el 100% de *jerk*, esto para generar los movimientos más suaves posibles. Por lo que solo se utilizarán 2 secciones para crear la *curva de aceleración*, *incremento de aceleración* y el *decremento de aceleración*. El tiempo será divido a la mitad, donde la primera mitad se asigna al tiempo *tp1* y la segunda mitad al tiempo *tp3*, dejando a *tp2* (tiempo asignado para la aceleración constante) en cero. Con el tiempo se realiza el cálculo de aceleración utilizado en las formulas, al iniciar la aceleración siempre es cero. Para este cálculo se utiliza la ecuación (2.3).

```
//Tiempo
  tp1 = Tdp/2;
  tp2 = 0;
  tp3 = tp1;
//Aceleraciones
   Apfin = (Ypfin-Ypin)/((tp1/2)+(tp3/2));
```

Es necesario llevar una cuenta del tiempo, que se utilizará para evaluar las ecuaciones, la sección *del control del tiempo* se encarga de esto último. La primera

ecuación se evalúa al cumplirse la condición de que el tiempo (*Seg*) sea mayor a *Xpin*, en otras palabras al punto inicial que se está evaluando en ese momento, y menor a *Xpin + tp1*, que no es otra cosa más que el punto inicial que se está evaluando más el tiempo de la primera sección de la *curva de aceleración* actual; esta cuenta es guardada en la variable *Stp1*. De igual manera se repite el proceso, esta vez para la variable *Stp3*, cuyas condiciones piden que *Seg* sea mayor a *Xpin + tp1* pero menor a *Xpfin*.

```
//Control del tiempo
  //Tiempo 1
  if((Seg >= Xpin)&&(Seg <= Xpin+tp1)){
     Stp1 = ((millis() - mp)/1000)-Xpin;
     m3 = millis();}
  //Tiempo 3
  if((Seg >= Xpin+tp1)&&(Seg <= Xpfin)){
     Stp3 = ((millis() - m3)/1000);}
```

Al mismo tiempo que el tiempo está avanzando es necesario evaluar las ecuaciones, las condiciones que selecciona la ecuación que se evaluará son las mismas que determinan la variable de tiempo que será activada (*Stp1* y *Stp2),* esto asegura una correcta sincronización entre las 2 partes. Las ecuaciones usadas en este paso son las anteriormente estudiadas y utilizadas en la clase de *CurvaAceleracion,* con la diferencia que en este algoritmo las variables han sido cambiadas para ajustarse a las necesidades presentes. Los resultados de las operaciones son guardados en una variable llamada *Vp,* (*Velocidad del Perfil*).

```
//Ecuaciones
//Incremento de aceleración
        if((Seg >= Xpin)&&(Seg <= Xpin+tp1)){
                Vp = ( (Apfin*(pow(Stp1,2)))/(2*tp1) ) + Ypin;
                V1 = ( (Apfin*(pow(tp1,2)))/(2*tp1) ) + Ypin;}
//Decremento de aceleración
        if((Seg >= Xpin+tp1)&&(Seg <= Xpfin)){
                Vp =  ( (-Apfin * pow(Stp3,2) / (2 * tp3))+(Apfin * Stp3) + V1) ;}
```

Finalmente los resultados son enviados al puerto serie mediante la variable *PortVel.* Cuando el tiempo ha alcanzado 30 segundos es necesario detener éste proceso; Al cumplirse la condición de *Seg >= 30* las variables que controlan el tiempo son igualadas a cero.

```
PortVel = Vp;
//Reseteo de valores al llegar al tiempo máximo
    if(Seg >= 30){
      Seg = 0;
      mp = 0;
    }
```

4.2.3 Encoder

En este punto hay que mencionar que la señal del *encoder* es leída e interpretada por *arduino*, antes de ser enviada a *processing*, pero una vez enviada hay que señalar algo interesante en la comunicación de esta variable, el valor máximo que *arduino* envía a través del puerto serial es de 255, lo que dificulta transmitir la cuenta exacta de pulsos que *arduino* ha registrado en tiempo real. Para solucionar esto se crearon algunas condiciones en *processing*,

```
if(Pulsos >= 230)
  cond1 = 1;
  if(cond1 == 1){
     if(Pulsos <= 50)
       cond2 = 1;}
      if((cond1 == 1)&&(cond2 == 1)){
        Ciclos ++;
        cond1 = 0; cond2 = 0;}
```

Los pulsos recibidos se almacenan en una variable llamada *Pulsos*, además de esto se crearon 2 variables llamadas *cond1* y *cond2* que ayudarán en la cuenta de pulsos. La primera condición se cumplirá si los pulsos enviados pasan el valor de 230 pulsos, esto permitirá acceder a la condición 2, que se cumplirá cuando la variable *Pulsos* pase a tener un valor menor de 50, la razón de esto es que como se mencionó anteriormente los valores enviados de *arduino* solo pueden alcanzar un máximo de 255 pulsos, luego de esto se reinician a 1 nuevamente y la cuenta empieza de nuevo. La idea de esta sección de código es poder diferenciar cuando la cuenta ha pasado de ser 255 y ha empezado de nuevo, esto para llevar una cuenta de las veces que ha reiniciado y poder tomarlo en cuenta para los cálculos futuros. Debido a la velocidad a la que gira el motor, se dejaron tolerancias grandes en el código, 230 pulsos antes de llegar a 255 y 50 pulsos después de pasar 0 para evitar que en un avance rápido de velocidad *processing* no pueda diferenciar el reinicio de los pulsos. Cuando las 2 condiciones se cumplen se activa un contador para la variable *Ciclos*, dicha variable registrará el número de veces

que se ha reiniciado la cuenta de pulsos. Al final las condiciones volverán a valer cero, lo que indica que se han cumplido; empezando así el proceso de nuevo.

La clase de *encoder* está diseñada para diferenciar cuando el motor gira en una dirección o en otra, tomando en cuenta esto, los pulsos y ciclos capturados por el programa se guardan en diferentes variables, *PulsosAvance* y *CiclosAvance* si el motor se encuentra en dirección de avance (Sentido es igual a 0) y las variables de *PulsosReversa* y *CiclosReversa* para la reversa (Sentido es igual a 1).

```
if(Sentido == 0){
        PulsosAvance = Pulsos;
        CiclosAvance = Ciclos;}
if(Sentido == 1){
        PulsosReversa = Pulsos;
        CiclosReversa = Ciclos;}
```

Teniendo la cuenta de ciclos y pulsos que se han enviados de *arduino* se procede a calcular el total de pulsos que *encoder* ha enviado, para esto se utilizan 2 formulas, aplicadas 2 veces, para reversa y para avance. El primer par de fórmulas trabajan cuando el motor está avanzando, donde la variable *PulsosTotalAvance* realiza una suma de los pulsos totales que el motor ha avanzado (*PulsosAvance*) y el número de ciclos que se ha avanzado multiplicado por 255 (*CiclosAvance*), debido a que cada ciclo vale 255 pulsos. La segunda fórmula toma el número total de pulsos (*PulsosTotalAvance*) y los multiplica por 0. 0010498, que es el avance en centímetros por cambio de estado.

```
PulsosTotalAvance = ((255*CiclosAvance) + PulsosAvance);
Avance = PulsosTotalAvance * 0.0010498;
```

Para la reversa las formulas son iguales exceptuando que las variables son alternadas por su equivalente en el sentido contrario.

```
PulsosTotalReversa = ((255*CiclosReversa) + PulsosReversa);
Reversa = PulsosTotalReversa * 0.0010498;
```

Finalmente se efectúa la suma entre ambos resultados (avance y reversa), donde el resultado de la suma de pulsos se guarda en la variable *PulsosTotal* pero el resultado

de la posición real del motor representado por la variable *PosicionReal* es la resta del avance menos la reversa que se ha recorrido.

```
PulsosTotal =  PulsosTotalAvance + PulsosTotalReversa;
PosicionReal = Avance - Reversa;
```

La variable de *PosicionReal* es la variable utilizada en la pantalla principal para mostrar en la gráfica la posición del motor. Es en esta clase donde se pueden encontrar restricciones de movimiento para la guía, lo que es muy necesario para evitar sobre esfuerzo del motor por si ocurriera alguna falla o tratará de avanzar cuando esté posicionado en uno de los extremos de la guía de metal.

```
if(PosicionReal>= 94.92){
  TopeDer = 1;
}else{ TopeDer = 0;}

if(PosicionReal<= 0 ){
  TopeIzq = 1;
}else{ TopeIzq = 0;}
```

PosicionReal no es más que la posición actual del motor en cualquier momento del movimiento. Utilizando ésta variable se puede detectar cuando es posible retroceder o avanzar y cuando lo anterior no es una opción viable. Si la posición del motor se encuentra en 94.92 centímetros significa que ha llegado al extremo derecho de la guía y no es posible avanzar, esto será indicado con la animación del avance del motor en la pantalla principal y con los *leds* colocados en la parte superior de los extremos del riel de metal. Por el contrario si *PosicionReal* tiene un valor de 0, el valor inicial, *processing* no permitirá retroceder con el motor, indicándolo en la pantalla principal por medio de un *led*.

4.2.4 Paneles inferiores

La función principal de estos paneles es mostrar información acerca de los procesos que está efectuando el programa para dar una noción más clara y acertada acerca de que está pasando. Es importante mencionar que estos no son procesos indispensables, solamente ayudan a comprender mejor el funcionamiento interno de la interfaz.

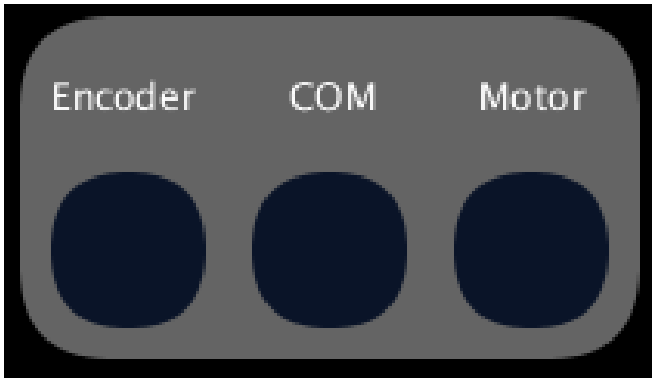

Figura 4.15. Imagen exterior de los paneles de información.

4.2.4.1 Panel del encoder

El primer panel pertenece al *encoder* y mostrará la posición real del motor dada por la variable *PosicionReal*, el número total de pulsos registrados dados por la variable *PulsosTotal*, y en la parte inferior los pulsos recibidos de parte de *arduino*, variable que como se ha mencionado antes alcanza su máximo de 255 unidades antes de reiniciar y volver a valer 1.

Figura 4.16. Panel de *encoder*.

4.2.4.2 Panel del puerto serie

Éste panel proporciona información del *puerto serie* con el que se trabaja, el primer valor mostrado es la *velocidad virtual*, que es la velocidad calculada por medio de ecuaciones, ésta velocidad es la que se genera mediante los procesos de éste programa, el segundo valor es un mapeo de la *velocidad virtual*, valor que es enviado a *arduino* para ser expresado en forma de *señal PWM.*

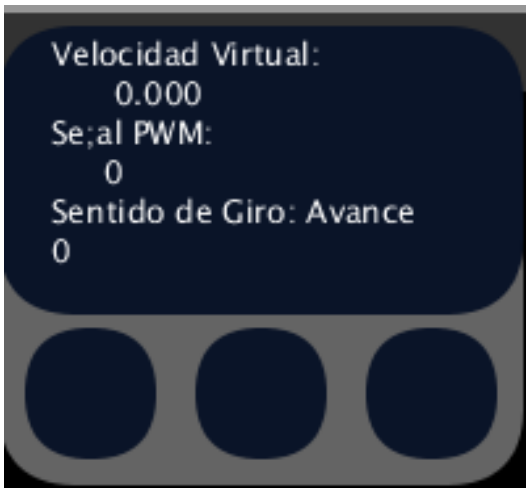

Figura 4.17. Panel del puerto serie.

El tercer valor es el sentido de giro que actuará sobre el motor, éste valor es enviado al puerto serie en forma de una señal *booleana*, 0 para avanzar y 1 para retroceder. Al final se encuentra la variable, *SerialVar* que no es más que la variable donde se almacenara los valores recibidos del *puerto serial* por parte de *arduino*.

4.2.4.3 Panel del motor

El último panel muestra el tiempo en que el programa ha estado funcionando, tiempo que es expresado en segundos mediante el comando *millis()* que se ha estudiado anteriormente; además de mostrar las revoluciones por segundo y minutos del motor, dichos valores son calculados en base al *encoder*.

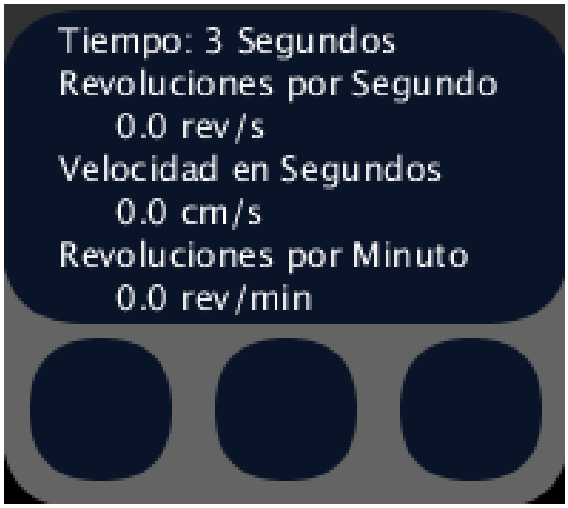

Figura 4.18. Panel del motor.

Como se ha explicado en el capítulo número 2, en el apartado de "*estimación de velocidad*", lo mejor es medir el diferencial de distancia en un minuto y estimar la velocidad en base a esos datos. Para la estimación de la velocidad se creó un algoritmo el cual está incluido en la clase del *encoder* en el programa.

```
switch(VelCase){
        case 1:
                tv = millis();
                pv = PulsosTotal;
                VelCase = 2;
        break;
        case 2:
                pt = millis() - tv;
                cp = PulsosTotal - pv;
                pcm = cp * 0.001049875;

                if( pt >= 1000){
                        VelRealcm = pcm;
                        VelRealrev = (cp/5760);  //Revoluciones por Segundo
                        VelRealrevmin = VelRealrev*60;
                        VelCase = 1;}
        break;
}
```

Para esta tarea el proceso se encuentra dentro de una estructura *switch* para poder alternar entre operaciones. La variable *VelCase* es la encargada de controlar la estructura *switch* donde el primer paso establece los valores que se utilizarán de referencia para los cálculos del segundo paso, primero establece el tiempo inicial de los cálculos mediante la variable *tv,* posteriormente se establece la cantidad de pulsos registrados en ese momento utilizando la variable *pv,* a continuación *VelCase* cambia su valor a dos dando inicio al segundo paso del proceso. Estas variables serán la referencia para los cálculos de la velocidad. En el segundo paso del proceso la primera operación calcula el periodo tiempo desde el momento en que ha iniciado el proceso y lo guarda en la variable *pt,* la variable *cp* registra el número de pulsos que se han contado en el periodo de tiempo que la variable *pt* ha trabajado, finalmente se realiza el cálculo de los centímetros que el motor ha avanzado multiplicando el número de pulsos contados y multiplicándolo por la constante 0.001049875, que como se ha mencionado es la distancia en centímetros que avanza el motor por cada pulso del *encoder*, el resultado es guardado en la variable *pcm* (pulsos a centímetros.). Los cálculos mencionados para este proceso se llevan a cabo continuamente excepto cuando la variable *pt* alcance los 1000 milisegundo, cuando ésta condición es cumplida el ultimo valor guardado por la variable *pcm* será equivalente a la velocidad en centímetros/segundos y se guardara en la variable *VelRealcm*; El segundo calculo corresponde a las revoluciones por segundo representadas por la variable *VelRealrev*, calculadas por la división del número de pulsos contados entre 5760 que como se mencionó en el capítulo 3, es el número de pulsos necesarios para una revolución del eje del motor. El tercer cálculo se guarda en la variable *VelRealrevmin*, revoluciones por minuto que es la multiplicación de las revoluciones en un segundo multiplicadas por 60 segundos. Finalmente la variable *VelCase* cambia su valor a 1 y el proceso se repite nuevamente desde el principio.

4.3 Implementación en arduino

Arduino tendrá la función de ser el puente entre *processing* y el sistema físico, debido a esto es necesario que tenga una comunicación rápida y precisa con los datos necesarios para los cálculos. El programa en *arduino* mandará una señal hacia *processing*, dicha señal es el número de pulsos registrados por el programa, al mismo tiempo *arduino* deberá recibir 2 variables de *processnig*, una variable que contenga el

valor que será enviado al motor de corriente directa y una segunda variable con la dirección de giro que éste deberá tener.

```
if (Serial.available() > 0) {
    SerialVar = Serial.read();
     //SerialVar = SerialVar - '0';
     if (SerialVar == 66){}else{
  if(Count == 0){
   Pwm = SerialVar;
   Count = 1;}
  if(Count == 1){
   Sentido = SerialVar;
   Count = 0;}
    }
```

Es necesario organizar los valores del *puerto Serie* en variables independientes para poder trabajar con ellas. Antes de esto se debe eliminar el valor 'B', que es el valor utilizado por *processing* para entablar la comunicación entre los 2 programas. Por tanto se agrega una condición que evita que se capturen valores cuando se detecte un 66, que en la tabla *ASCII* 66 corresponde a B, al quedar solamente con los valores importantes, falta por medio de un contador repartir los datos en las variables *PWM* y *Sentido*, que dichos valores guardaran la señal *PWM* calculada en *processing* por medio de ecuaciones y el sentido que el motor deberá presentar para moverse adecuadamente.

La variable de *Sentido* será expresada por medio del comando, *digitalWrite()*, en la salida digital número 6, mandando un *1 lógico* si el sentido tiene un valor de 1, *0 lógico* si sentido tiene un valor de 0.

4.3.1 Comando para la señal *PWM* en arduino

Después de que el valor de la velocidad que se desea ha sido transmitido de *processing* a *arudino*, lo siguiente será mandarlo al exterior para controlar el motor de corriente continua. Para exteriorizar este valor se utilizará la función *analogWrite();* para expresar un valor en forma que se simule una *señal PWM,* para esto se define el *pin digital número 9* que cuenta con la función para trabajar con *señales PWM,* seguido de esto es necesario utilizar la variable donde se guarda el valor *PWM* de *processing,* en otras palabras la variable *PWM* en conjunto con el comando anterior.

Al final se obtiene el comando listo para controlar el motor adecuadamente, *analogWrite (9, PWM);* el comando anterior dice que el *pin digital 9* funcionará como salida para la *señal PWM.*

4.3.2 Lectura del encoder con interrupciones

Uno de los aspectos más importantes del programa en *arduino* es la lectura del *encoder*, lo cual se hace mediante el uso de *interrupciones*. Utilizando el comando *attachInterrupt(1,blink,CHANGE),* se lleva a cabo una lectura precisa del número de veces que el lector del *encoder* cambia su estado, de alto a bajo y de bajo a alto, cada uno de estos cambios representa una pequeña fracción de un centímetro y conociendo ese valor es posible estimar el avance del motor.

Para utilizar el comando *attachInterrupt* se debe configurar una de las dos entradas con ésta función, en el presente trabajo se utilizó la entrada digital 3, lo que corresponde en el software a la interrupción número 1. Para configurar el contador se utiliza una función a la que se denomina *blink*, la cual se activa cuando se detecta alguna interrupción y contiene un contador que aumenta su valor en uno cada vez que esta función *blink*, es ejecutada.

```
void blink(){
  Contador ++;
}
```

Para finalizar se utilizó la función *CHANGE* cuyo objetivo es que una interrupción se tome en cuenta cada que la señal cambia su valor, si en otro caso se hubiese utilizado otra función tal como *HIGH* o *LOW*, las interrupciones se activaran cuando el valor cambie a *1 lógico* o a *0 lógico*, respectivamente.

Con todos estos datos definidos se da forma al comando, el cual se activara cada que se detecte un cambio en la señal del *encoder*, cualquier cambio de estado, con cada registro de una interrupción se llama a la función *blink*, que aumentara el valor de un contador en 1 por cada interrupción.

4.4 Circuitos electrónicos

Una vez que las señales de *processing* han sido enviadas a *arduino*, el sistema está listo para exteriorizar esas señales y hacer funcionar a los actuadores, así como capturar las señales dadas por los sensores del sistema. Para esto fueron diseñados varios circuitos que en conjunto logran realizar las tareas anteriores. Por cuestiones de espacio se diseñaron 3 placas electrónicas que contienen todos los circuitos necesarios, además de trabajar en conjunto con la placa *arduino*.

4.4.1 Placas electrónicas

El presente sistema se basa en 2 salidas que provienen de *processing*, la primera es la *señal PWM* que dictara la velocidad del motor y físicamente estará fijada en el *pin 9* de *arduino* debido a que éste es uno de los pines capaces de manejar las señales *PWM* y una señal *booleana* que servirá para indicar el sentido de giro del motor, dicha señal está dirigida para el *puente H.*

4.4.1.1 Señal PWM

Como se ha mencionado la *señal PWM* saldrá del *pin 9* de *arduino*, ésta señal antes de llegar al motor pasará por un tratamiento que la dejará en un estado óptimo, como se mencionó en el capítulo pasado el tratamiento consta de 2 etapas.

1) Etapa de aislamiento.
2) Etapa de potencia.

Tomando la información del capítulo número 2, metodología y conociendo el funcionamiento de dichas etapas es posible hacer un diseño que se ajuste a las necesidades del trabajo. Teniendo el diseño terminado se procederá a llevarlo al PCB.

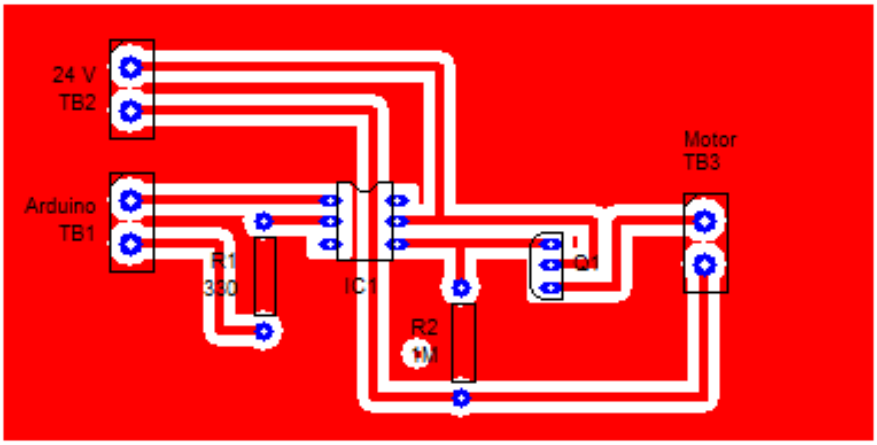

Figura 4.19. PCB de la etapa de potencia y aislamiento utilizados para la señal *PWM* de *arduino*.

Como se puede apreciar el *PCB* contará con 3 *terminal block* de 2 entradas, en el primer *terminal block*, *TB1* está conectado a la *señal PWM* que saldrá del *pin 9* de *arduino,* además de utilizar la tierra del mismo para cerrar el circuito. El *terminal block 2, TB2*, proporciona el voltaje de *24 volts* que el motor necesita para funcionar, éste voltaje será dado por una fuente externa, el tercer *terminal block, TB3*, es por donde pasará la señal que llegará al *puente H* para posteriormente salir al motor. Este circuito es el equivalente al circuito 3.28 del capítulo 3, metodología.

Entre los componentes se utilizó un *optoacoplador* como se observó en el capítulo pasado, aquí se utilizará un *4N35* para la *etapa de asilamiento* y proteger a *arduino* ya que éste componente es ideal para el trabajo debido a que puede ser activado con una señal de *5 volts*, igual a la proporcionada por *arduino* y es capaz de soportar los *24 volts* necesarios para activar el motor. Se utilizó un *tip 122* para la etapa de potencia debido al voltaje y amperaje que soporta.

4.4.1.2 Sentido de giro

Para exteriorizar la señal que controle al *puente H*, es necesario contar con una *etapa de potencia* y la *etapa de aislamiento* trabajando a *5 volts*, que bien, pueden ser suministrados por *arduino*; *TB1* proporciona la señal de *arduino* que controla al *puente H;* Mientras que *TB2* suministrara el voltaje de *5 volts* de *arduino. TB3* será la salida hacia las bobinas de los relevadores del *puente H*.

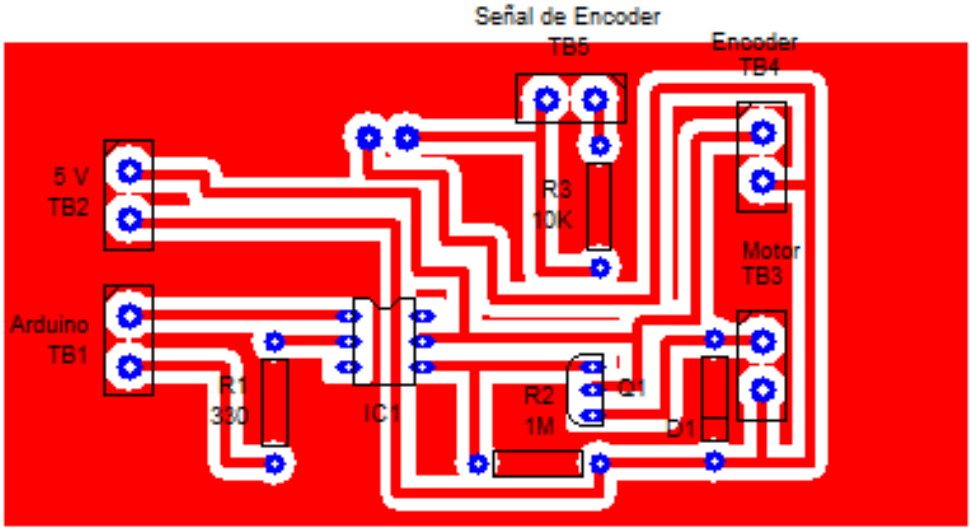

Figura 4.20. Etapa de potencia para el control del *puente H.*

Al igual que en el punto anterior se seleccionó un *optacoplador 4N35* por los mismos motivos y un *tip 122.*

Parte del circuito es la etapa de potencia y aislamiento de la señal de conmutación del *puente H*, otra parte del circuito, en la parte superior de éste, corresponde al tratamiento que se le da a la señal del *encoder* para que pueda ser interpretada por *arduino*. Donde se tomará la alimentación de *5 volts* de *arduino* para energizar el *ancoder*, aquí *TB3* es el *terminal block* por donde será alimentado el *encoder* y *TB5* corresponde a la señal registrada por el *encoder* al variar la posición del motor. Este circuito es el equivalente al 3.29 del capítulo de metodología.

4.4.1.3 Puente H

Los circuitos anteriores podrán controlar al *puente H* correctamente al trabajar en conjunto. El circuito del sentido de giro de la figura 4.20 manda una señal al *puente H.* Con lo que así podrá alternar entre los estados de los relevadores y controlar la dirección del motor.

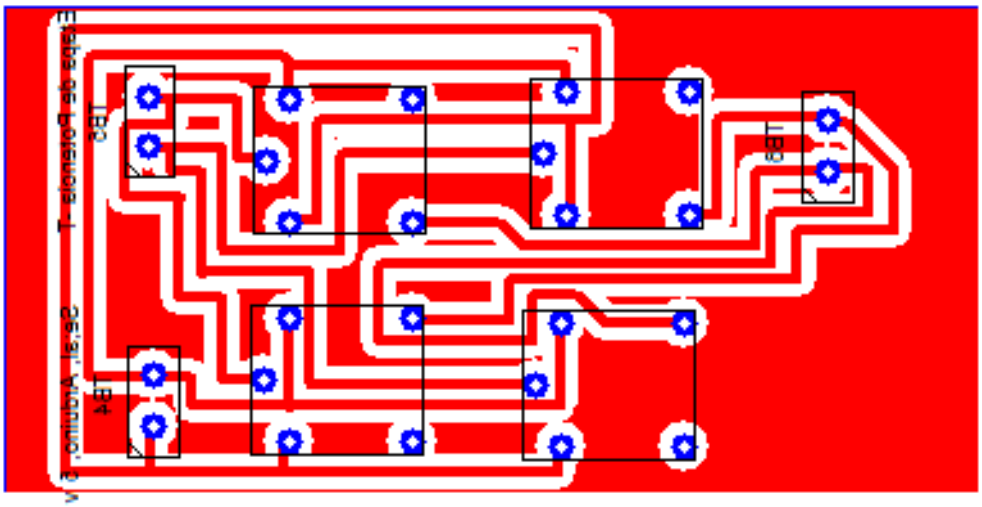

Figura 4.21. *Puente H.*

En el circuito de la figura 4.21 solo se utilizan 4 relevadores de marca *SUN HOLD,* que tengan una conmutación a *5 volts* y puedan soportar *24 volts* a través de ellos, con el último aspecto no habrá problema en lo absoluto. Éste circuito equivale al circuito 3.30 del capítulo de metodología.

4.4.1.4 Encoder

El sistema necesita tener algún sensor para ayudar al programa a conocer mejor los efectos de las salidas en el motor, para ésto se utilizó un *encoder* acoplado al motor. Debido a que el *encoder* tiene solo una señal de salida constituida por un tren de pulsos hay que diseñar un sencillo circuito para poder hacer esta señal apta para que *arudino* pueda leerla e interpretarla, como se ha mencionado este circuito está incorporado en la parte superior del circuito de la figura 4.20 y es el circuito 3.27 del capítulo 3.

4.5 Gabinete

Una vez los circuitos electrónicos estén funcionando correctamente, éstos se deben de colocarse dentro de un gabinete diseñado para contener la *placa de arduino.*

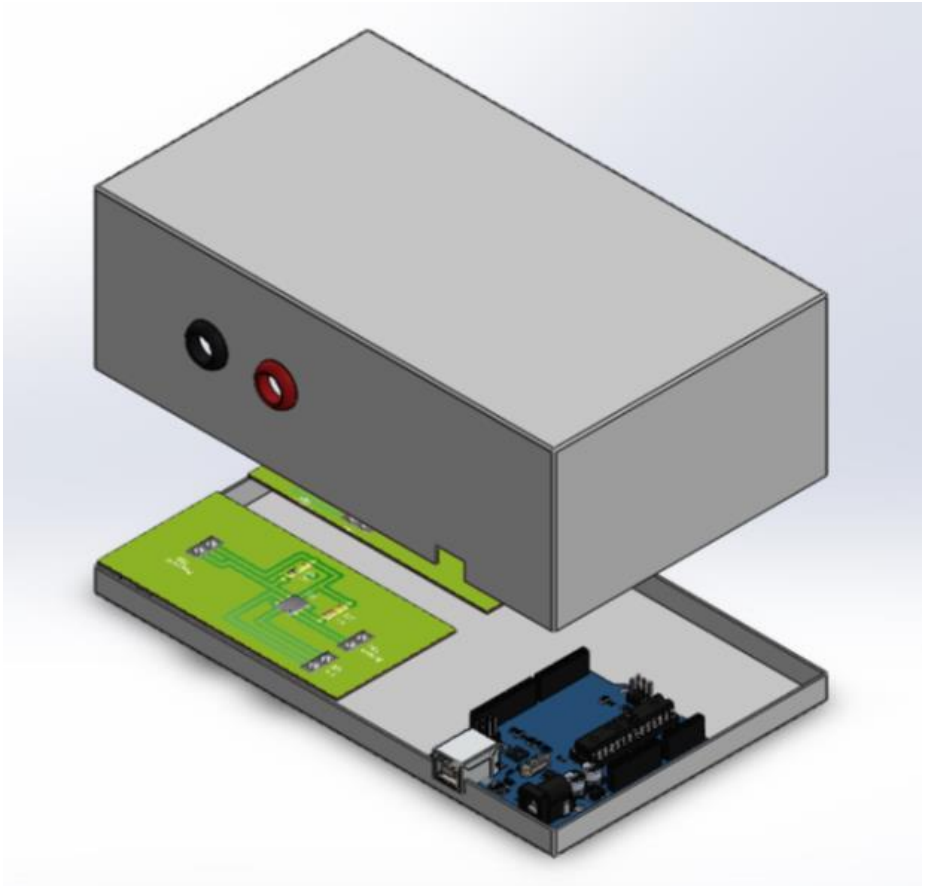

Figura 4.22. Simulación de la vista del gabinete terminado.

Al final se tendrá un contenedor diseñado especialmente para las necesidades del presente trabajo que cuenta con las entradas y salidas necesarias.

Figura 4.23. Resultado final del contenedor con los circuitos electrónicos.

4.5.1 Contenedor

Para el contenedor donde descansarán los circuitos se utilizó una caja genérica de marca *steren* de medidas 19.3 x 11.5 x 7.5 cm.

Figura 4.24. Gabinete marca *steren* de 19,3 x 11,5 x 7,5 cm.

Antes de montar los 3 circuitos se debe que hacer algunas modificaciones a la caja. Lo primero es instalar las entradas por las cuales se alimentarán los circuitos a *24 volts*.

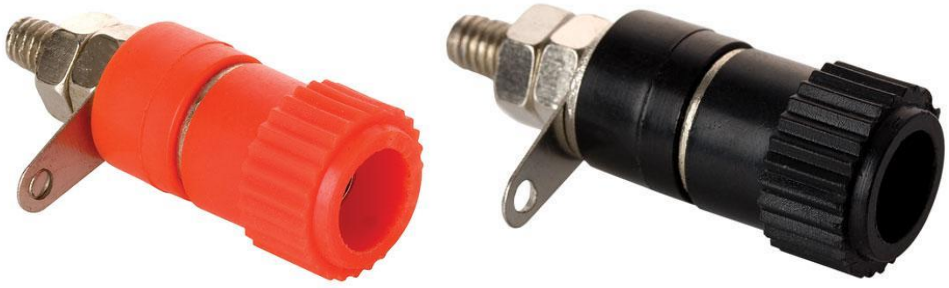

Figura 4.25. Bornes de 10 amperes con entrada tipo *phillips*.

Para esto se requiere entradas tipo *phillips* colocadas en el costado donde se encuentra la entrada para el cable *usb* de *arduino*.

Figura 4.26. Entradas tipo *phillips*.

Mientras que por el otro costado se necesitará un orificio para los cables del motor y el *encoder*,

Figura 4.27. Salida para los cables del motor.

4.5.2 Circuitos

Por cuestiones de espacio la placa *arduino* junto con las *etapas de potencia* se han colocado en la base del gabinete de *steren* por medio de sujetadores para placas electrónicas.

Figura 4.28. Placa de *arduino* y circuitos electrónicos.

El *puente H* estará situado junto a la salida del gabinete, debido a que es este circuito de donde saldrán las señales que controlarán al motor, la *señal PWM* y la dirección de giro.

Figura 4.29. *Puente H* posicionado en una de las paredes del gabinete.

4.6 Servomecanismo lineal

Una vez que todas las partes del sistema han sido desarrolladas y probadas por separado, es momento de acoplar todos los sistemas independientes en el producto final. Los circuitos electrónicos se han diseñado para trabajar con *24 volts*, por lo que el gabinete al contar con entradas *phillips* es posible conectarlo a cualquier *fuente de*

poder. Al mismo tiempo *arduino* solo necesita de un *cable usb* para conectarse a la computadora donde se encontrará el programa.

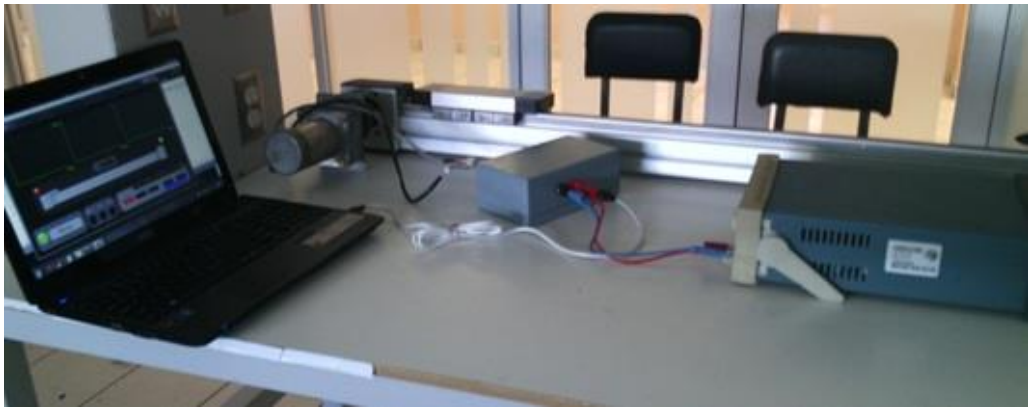

Figura 4.30. Imagen del sistema del servomecanismo lineal.

Como se aprecia en la figura 4.31 el riel metálico y el motor estaban diseñados para trabajar en conjunto.

Figura 4.31. Motor ensamblado en el elemento rotativo de la guía lineal.

Capítulo 5

Conclusión

En general se cumplió con los objetivos generales y específicos del trabajo. Se diseñó una interfaz capaz de construir perfiles de velocidad en base a las necesidades de un operador. Esto de manera intuitiva y sencilla, solo con un básico conocimiento de control; a pesar de lo anterior se encontraron aspectos que entorpecieron el desarrollo del proyecto, como la planeación previa al trabajo, la cual no abarco los puntos necesarios en el programa en *processing* por lo que provoco que algunas partes de la programación tuvieron que ser diseñadas varias veces para adaptarse a los cambios que surgieron durante el avance del proyecto generando una gran pérdida de tiempo.

Al realizar las pruebas físicas del sistema, el motor fue capaz de variar su velocidad de manera correspondiente al programa, sin embargo permanecieron algunas imprecisiones entre la velocidad real del motor y la velocidad virtual del programa. En la realización de las pruebas se concluyó que estas fallas tenían su origen en la comunicación entre *processing* y *arduino*, debido a la velocidad de captura de datos del *encoder* que era demasiado rápida y los datos no podían transmitirse a una velocidad óptima tomando en cuenta que el número más alto que puede transmitirse es 255; dichos problemas provocaban que la estimación de velocidad tuviera inconsistencias y fuera poco estable.

A pesar de los problemas antes mencionados el trabajo fue satisfactorio, presentando diferentes opciones de control funcionales, y logrando representar todos los aspectos necesarios en la interfaz diseñada.

Bibliografía

[1] (Junio, 2015). MD03 - 24Volt 20Amp H bridge motor drive. [Online] Available: http://www.robot-electronics.co.uk/htm/md03tech.htm

[2] (Julio, 2015). MD49 - Dual 24 volt 5 Amp H bridge motor drive. [Online] Available: http://www.robot-electronics.co.uk/htm/md49tech.htm

[3] (Agosto, 2015). Controlador motores DC 10 AMP por RC. [Online] Available: S310108http://www.superrobotica.com/S310108.htm

[4] (Junio, 2015). [Online] Available. http://www.ti.com/lit/ds/symlink/drv8801.pdf

[5] (Diciembre, 2002). Control de motor DC con 555. [Online] Available: http://www.unicrom.com/cir_control-motor-dc-555.asp

[6] (Julio, 2015). How to Estimate Encoder Velocity. [Online] Available: http://www.embeddedrelated.com/showarticle/158.php

[7] W. Voss, A Comprehensible Guide to Servo Motor Sizing, Copperhill Media, United States, 2007.

[8] B. Júnez, A. López, R. Rojas, Matemática III Geometría Analítica. Fondo de Cultura Económica, Distrito Federal, México, 2004.

[9] B. Fry, C. Reas. (2001). Processing Foundation. [Online] Available: https://processing.org

[10] Arduino. [Online] Available https://www.arduino.cc

[11] (Marzo, 2015). Comunicando Arduino con Otros Sistemas. [Online] Available: http://playground.arduino.cc/ArduinoNotebookTraduccion/Appendix4

[12] (2015). Modulación Por Ancho de Pulsos. Available [Online] Available: http://es.wikipedia.org/wiki/Modulación_por_ancho_de_pulsos

[13] (Agosto, 2015) Etapa de potencia. [Online] Available: https://es.wikipedia.org/wiki/Etapa_de_potencia

[14] (Febrero, 2015) Optoacoplador. [Online] Available: https://es.wikipedia.org/wiki/Optoacoplador

[15] (feb, 2015). Puente H (Electornica). [Online] Available: http://es.wikipedia.org/wiki/Puente_H_(electrónica)

[16] (Mayo, 2015) Rotatory Encoder. [Online] Avalible: http://en.wikipedia.org/wiki/Rotary_encoder

[17] J Chiasson, Modeling and High-Performance Control of Electric Machines, IEEE, New Jersey and Canada, 2005.

[18] (Mayo, 2015). [Online] Available: https://mx.answers.yahoo.com/question/index?qid=20130513133747AA8TqCk

[19] C. Collaguazo, J. Litardo, “Diseño Y Construcción De Un Sistema De Control para Motores CC. En Base a un Circuito Integrado 555.”, Facultad de Electrónica y Ciencias Industriales, Escuela de Control y Automatización..

[20] L. Flores, A. García, O. Molina. “Arranque suave para un motor de CD a través de un convertidor reductor CD–CD“, (Octubre, 2010), Instituto de Electrónica y Mecatrónica Universidad Tecnológica de la Mixteca.

[21] A. Del Carmen Jaime, C. Hernández Ramírez, “Control PID de la velocidad de una banda transportadora para la clasificación de objetos”, (Noviembre, 2011), Unidad Académica de Ingeniería Eléctrica, Zacatecas, Zacatecas.

[22] A. Morales, O. Nuñez,”Control de Motores de Corriente Directa con interface a PC basada en un microcontrolador”, Instituto tecnológico de Querétaro.

[23] (Marzo, 2015). Opic Photointerrupter with encoder function. [Online] Available: http://pdf1.alldatasheet.com/datasheet-pdf/view/118742/SHARP/GP1A16R.html

[24] (Marzo, 2015). [Online] Available: http://bogotacity.olx.com.co/motoreductor-motorreductor-motor-reductor-tokushu-24v-dc-150rpm-ref-td3152g24f9k24c-iid-710902105

[25] (Junio, 2015). [Online] Available: http://playground.arduino.cc/Main/RotaryEncoders