Universidad de La Laguna

Escuela Superior de Ingeniería y Tecnología

Sistema de control para una silla de ruedas eléctrica

Grado en Electrónica Industrial y Automática

Trabajo de fin de grado realizado por

Carlos Menéndez Perdomo

bajo la supervisión del tutor

Jonay Tomás Toledo Carrillo

y la cotutora

Bibiana Fariña Jerónimo

Convocatoria de Julio, curso 2022/23

Universidad de La Laguna

Tabla de contenido

Universidad de La Laguna

Lista de figuras

Lista de tablas

Lista de ecuaciones

Resumen

En este Trabajo de Fin de Grado se ha llevado a cabo el desarrollo del sistema de control para una silla de ruedas eléctrica, reemplazando el hardware original por uno desarrollado específicamente. Se habilita así la posibilidad de recibir comandos desde un ordenador, ampliando las posibilidades de control y facilitando una futura implementación algoritmos de navegación autónoma.

En primer lugar, se han instalado encoders incrementales en las ruedas de la silla de ruedas eléctrica. Estos sensores proporcionan información precisa sobre la velocidad de las ruedas, lo cual resulta fundamental para el control adecuado del sistema.

A continuación, se ha llevado a cabo el desarrollo de la electrónica y el software de control. El controlador ha sido sintonizado específicamente para adaptarse a las características y necesidades de la silla de ruedas eléctrica. Esto ha implicado el ajuste de los parámetros del controlador PI y la implementación de algoritmos *anti-windup* para evitar comportamientos indeseados.

Además, se han desarrollado dos interfaces de control diferentes. La primera es a través de un joystick, lo cual proporciona una opción de control manual y directo para el usuario. La segunda es a través de una interfaz de conexión con el ordenador de control, lo cual permite enviar comandos desde el ordenador para controlar la silla de ruedas eléctrica de manera remota.

Palabras clave: Silla de ruedas eléctrica, control de velocidad, sistema de control, encoders incrementales, controlador PI, interfaz de control.

Abstract

In this Final Degree Project, we have developed a control system for an electric powered wheelchair, replacing the original hardware with a specifically developed one. This enables the possibility of receiving commands from a computer, extending the control possibilities, and facilitating future implementation of autonomous navigation algorithms.

Incremental encoder sensors have been installed on the wheels of the electric wheelchair. These sensors provide accurate information about the speed of the wheels, which is essential for the proper control of the system.

This was followed by the development of the electronics and control software. The controller has been tuned specifically to suit the characteristics and needs of the electric wheelchair. This has involved the adjustment of the PI controller parameters and the implementation of anti-windup algorithms to avoid unwanted behaviour.

To avoid unwanted behaviour, two different control interfaces have been developed. The first is via a joystick, which provides a manual and direct control option for the user. The second is via a connection interface to the control computer, which allows commands to be sent from the computer to control the power wheelchair remotely.

Keywords: Electric wheelchair, speed control, control system, incremental encoders, PI controller, control interface.

1. Introducción

Las sillas de ruedas eléctricas proporcionan una alternativa muy atractiva a las convencionales: la autonomía y libertad que ofrecen a las personas de movilidad reducida y a las personas mayores hacen que este tipo de dispositivos sean objeto de mejora continua, especialmente respecto a las funcionalidades que implementan y los algoritmos de control que las gobiernan. En cierto modo, este interés por la incorporación de sistemas automatizados y de control no es más que un reflejo de una sociedad que reclama cada vez más este tipo de comodidad en todos los aspectos de su vida. Las ventajas en este caso específico son evidentes: no todos los usuarios de estos dispositivos tienen las mismas capacidades para poder manipularlos con facilidad, por lo que la ayuda de los sistemas de control resulta crucial para poder desplazarse de manera segura y confortable.

En el presente proyecto se sientan las bases de una silla de ruedas eléctrica que pueda implementar en el futuro esta clase ayudas y funcionalidades tanto semiautomáticas como completamente automáticas, incluyendo las propias de un AGV[1]. Para ello se lleva a cabo el diseño e implementación de un controlador PI (Proporcional Integral) que permita realizar un control apropiado de la velocidad de las ruedas de la silla. Previo al control de la velocidad, es necesaria la puesta a punto de las conexiones pertinentes, entre las que se encuentran la comunicación entre el controlador de los motores y el microcontrolador, los periféricos utilizados como los sensores y el joystick, así como la alimentación de todos los dispositivos.

Este documento detalla la instalación y las conexiones realizadas de los componentes necesarios para realizar este tipo de control, el código desarrollado, la estrategia de control llevada a cabo para sintonizar el controlador PI, y una relación de problemas encontrados y cómo se han abarcado durante la elaboración del proyecto.

[1] Automated Guided Vehicle

2. Objetivos

El objetivo de este proyecto es desarrollar y evaluar un controlador Proporcional-Integral (PI) que permita un control adecuado de la velocidad de los motores de una silla de ruedas eléctrica, de modo que la conducción sea agradable y segura para los usuarios.

A través de la aplicación de los principios de control estudiados a lo largo del grado se busca llevar a cabo una implementación práctica en un caso real. En esta aplicación se incluye el diseño del sistema de control y también el ajuste de los parámetros del controlador.

También se pretende preparar la silla para que pueda recibir comandos por ordenador, abriendo las puertas a la implementación de algoritmos de navegación autónoma en futuros proyectos.

Para lograr estos objetivos se llevarán a cabo las siguientes tareas:

- Instalación de los componentes necesarios para realizar el control de velocidad, incluyendo los sensores utilizados para medir la velocidad de las ruedas y los dispositivos de control.
- Desarrollo de un código que permita interpretar la información proveniente de los sensores y la implementación del controlador PI. También se implementará la comunicación entre los dispositivos de control y el sistema de la silla de ruedas.
- Realización de experimentos y pruebas para ajustar los parámetros del controlador PI y otros algoritmos relacionados, como los utilizados para evitar comportamientos indeseados.
- Implementación de la comunicación con un ordenador, permitiendo recibir comandos de velocidad desde el ordenador y enviar información de odometría de vuelta.

3. Descripción del sistema

3.1 Silla de ruedas eléctrica

La silla de ruedas eléctrica utilizada para este proyecto es de la casa Vermeiren Ibérica S.L., del modelo Forest 3 (*Figura 1*). Es capaz de alcanzar los 14 kilómetros por hora y permite seleccionar distintas velocidades intermedias. Es capaz de superar pendientes con una inclinación de hasta 9º gracias a la tracción proporcionada por dos motores de 350W.

Figura 1. Silla de ruedas Vermeiren Forest 3

La silla cuenta con dos ruedas motrices traseras, las cuales giran alrededor de un eje horizontal. Para girar es necesario, por lo tanto, un control diferencial de estas ruedas motrices. Las ruedas delanteras permiten rotación además en torno a un eje vertical, de modo que se facilitan las diferentes trayectorias posibles al reducir la fricción en curvas.

Estas ruedas guía dificultan en gran medida el control del sistema al existir un número muy amplio de escenarios en los que las ruedas no se alinean con la trayectoria deseada. La fricción añadida por estos escenarios crea problemas relacionados sobre todo con la rotación diferencial, tales problemas se detallarán más adelante en la sección 7.2 Ruedas guía.

Ambas ruedas motrices tienen su propio freno electromagnético individual, mostrado en la *Figura 2*, el cual está siempre activado por seguridad hasta que es alimentado por 24V tras recibir una señal de control. Este freno permite su desarme manual mediante una palanca de control situada en los propios frenos. Idealmente siempre estarán armados y se controlarán electrónicamente.

Figura 2. Freno del motor izquierdo junto a la palanca de desarme manual.

3.2 Controlador Sabertooth 2x32

Las ruedas motrices son propulsadas por dos motores de corriente continua independientes entre sí, los cuales son excitados mediante una señal PWM proveniente del controlador Sabertooth 2x32, mostrado en la *Figura 4*. Esta señal controla un puente H que permite la inversión del sentido de giro de las ruedas, pero que también es capaz de aportar la potencia que un motor de estas características requiere. Por estos motivos los puente H son ampliamente utilizados en el control de este tipo de motores, pero

también en una amplia variedad de aplicaciones robóticas. Un puente en H con transistores actuando de interruptores se muestra en la *Figura 3*.

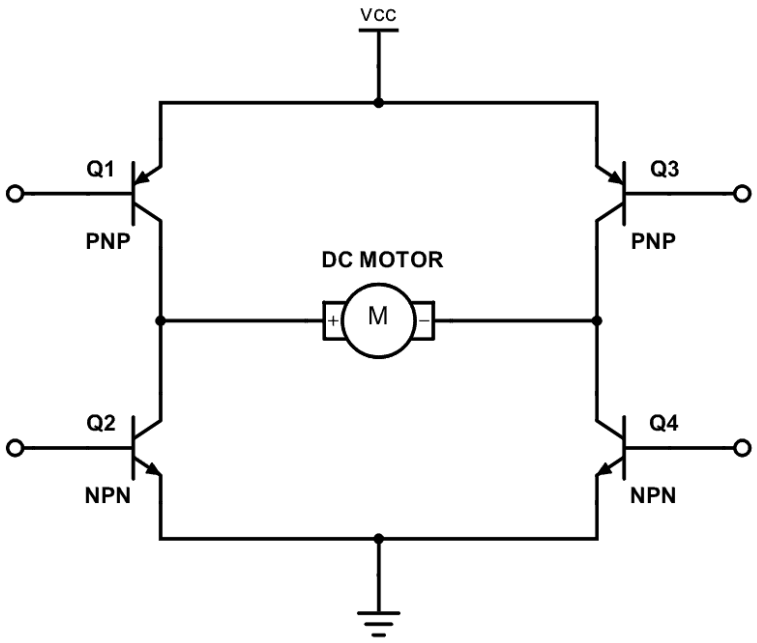


Figura 3. Puente H controlado por transistores BJT

Este controlador permite suministrar 32 amperios a dos motores, con picos de hasta 64 amperios. Permite también una amplia variedad de modos de control (radio control, analógico, etc.) pero a nosotros nos interesa exclusivamente el control serial TTL en la elaboración de este proyecto [1]. Mediante este control serial podemos comunicarnos con la placa de desarrollo Arduino, detallada en más profundidad en el apartado 3.3 Arduino MEGA 2560, permitiéndonos recibir los comandos que enviaremos a los distintos motores a través de uno de los puerto UART[2] que incorpora el Arduino.

[2] Universal Asynchronous Receiver-Transmitter (Transmisor-Receptor Asíncrono Universal).

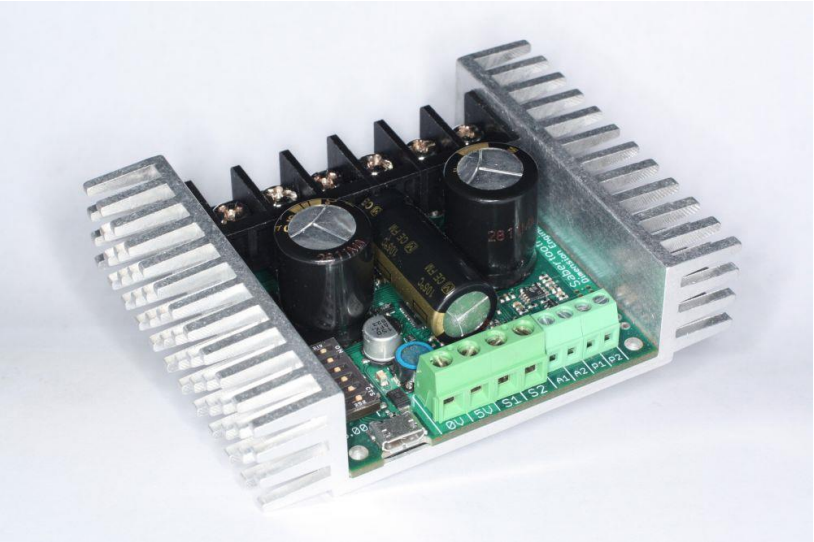


Figura 4. Controlador Sabertooth 2x32 [1]

Un diagrama con las entradas y salidas del controlador Sabertooth 2x32, las cuales se describirán a continuación, se proporciona en la *Figura 5*.

Para realizar la conexión serial se utilizan los puertos S1 y S2 del Sabertooth, actuando el primero como receptor *Rx* y el segundo como transmisor *Tx*. El Sabertooth también permite la alimentación de dispositivos periféricos, como nuestro Arduino o los sensores, para ello se usan las salidas de 0V, 5V situadas al lado de S1 y S2. Los *Power Outputs* P1 y P2 permiten controlar cargas de hasta 8 amperios de corriente máxima. Inicialmente estas salidas eran utilizadas para controlar los frenos electromagnéticos de la silla de ruedas, sin embargo, por motivos que se describen en 7.1 Frenos, finalmente no se han utilizado estos puertos.

B+ y B-, referidos conjuntamente como *Main Power Input* son las entradas de la fuente de alimentación o batería utilizada, permitiendo un rango de entre 6V a 33.6V. Como se ha mencionado en el apartado 3.1 Silla de ruedas eléctrica, nuestra silla de ruedas incorpora unas baterías que aportan 24V cuyos terminales se conectan a estas entradas del controlador.

Las conexiones a los motores se realizan a través de las salidas M1A-M1B y M2A-M2B, referidas simplemente como *Motor 1* y *Motor 2* en el diagrama.

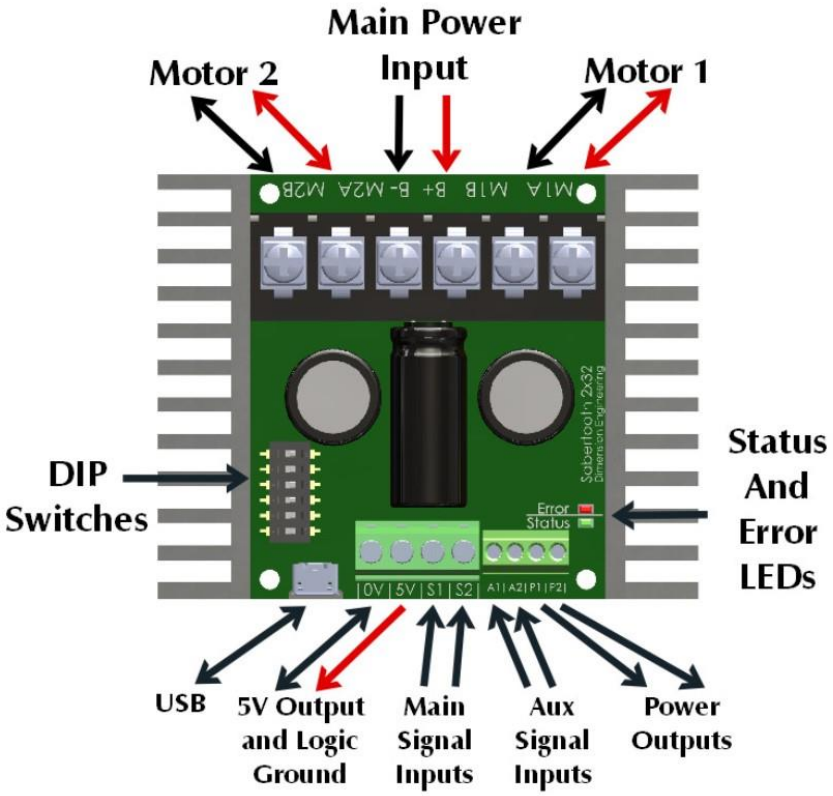


Figura 5. Diagrama de conexiones del controlador Sabertooth 2x32 [1]

El controlador incorpora también un puerto USB para configurar distintas funcionalidades del controlador mediante el software propio *DEScribe*. En la *Figura 6* se muestra la ventana principal de este software, que permite la configuración de distintos parámetros que definen el comportamiento del Sabertooth. Entre ellos se encuentra la limitación de corriente, el freno regenerativo, compensación de la batería y muchos otros definidos en [1]. A través de este software se definió inicialmente el comportamiento de los frenos de la silla, utilizando las salidas P1 y P2.

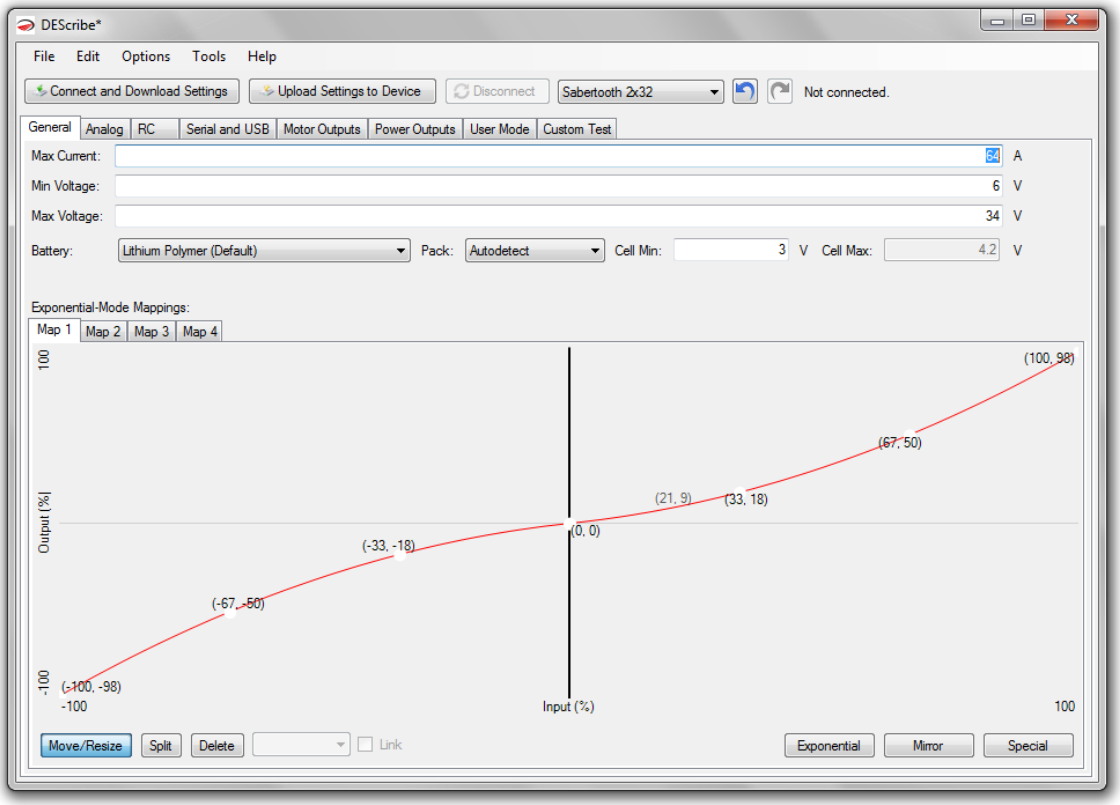


Figura 6. Ventana del software DEScribe de configuración del Sabertooth

3.3 Arduino MEGA 2560

La placa Arduino MEGA 2560 rev 3 (*Figura 7*) incorpora, como su nombre indica, un microcontrolador ATMega2560 de la familia Atmel. Esta placa de desarrollo, y otras similares de la familia Arduino, se utilizan en un amplio abanico de aplicaciones de

electrónica debido a su facilidad de implementación en todo tipo de proyectos. El Arduino MEGA 2560 ofrece una gran cantidad de entradas/salidas tanto analógicas como digitales, lo cual ha facilitado la conexión de los distintos dispositivos. El gran número de pines con capacidad de interrupción es uno de los principales motivos por los que se ha seleccionado esta placa de desarrollo, ya que son necesarios para leer correctamente los datos enviados por los sensores. Se ofrece una explicación más detallada de estos sensores en el apartado 3.4 Encoders ópticos.

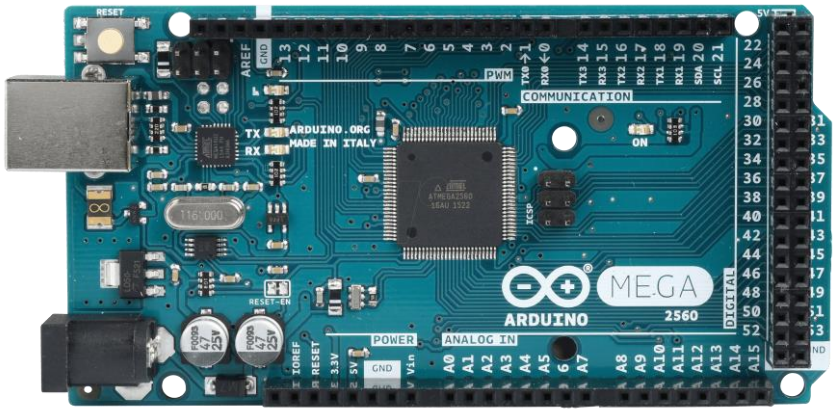


Figura 7. Arduino MEGA 2560 rev 3 [2]

Como se mencionó en el apartado anterior, también incorpora cuatro puertos UART de comunicación serial. Se utiliza uno de ellos para comunicarnos con el Sabertooth, enviando los comandos que se obtendrán mediante el control PI.

3.4 Encoders ópticos

Un par de encoders ópticos incrementales se utilizan para obtener lecturas del cambio en la rotación de las ruedas. Poseen tres canales que envían pulsos en referencia a la lectura que realizan del disco codificado, el cual se sitúa en el eje de la rueda. Dos canales, normalmente denominados “A” y “B”, como los mostrados en la *Figura 8*, leen las mismas marcas del disco codificado, pero se encuentran desfasados 90º eléctricamente entre sí para poder inferir información acerca del sentido de giro de la rueda. Un tercer

canal envía un pulso cada vez que se ha completado una revolución permitiendo, por ejemplo, comprobar si las velocidades calculadas son correctas.

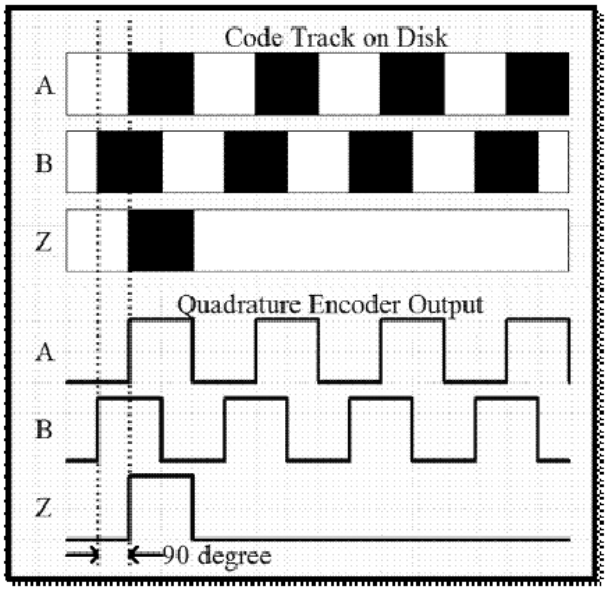


Figura 8. Output de los distintos canales del encoder [3]

En nuestro caso el disco codificado tiene 1024 marcas, pero la resolución final de las lecturas vendrá dada por la decodificación de los pulsos enviados por los canales "A" y "B". Evaluando las posibles combinaciones de los flancos de subida y bajada de ambos canales obtenemos cuatro veces la resolución del disco codificado, esto se conoce como decodificación de encoders de cuadratura o *quadrature decoding*, tal y como se muestra en la *Figura 9*

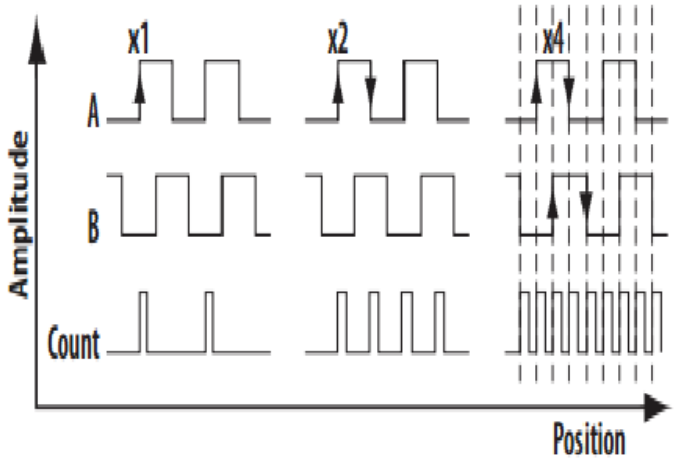


Figura 9. Quadrature decoding [3]

Los discos codificados utilizados, como se ha mencionado, tienen 1024 marcas, es decir: aproximadamente 2.84 lecturas de las marcas por cada grado. Gracias a la decodificación de cuadratura, esta resolución aumenta a 11.38 lecturas de las marcas por cada grado. En otras palabras, la resolución de las cuentas es de aproximadamente 0.09 grados por cuenta. Además, teniendo en cuenta el radio de las ruedas, podemos determinar también la resolución del desplazamiento lineal.

$$Resolución\ en\ mm = \frac{2\pi * Radio\ [mm]}{4096\ [\frac{cuentas}{revolucion}]}$$

Ecuación 1. Determinación de la resolución del desplazamiento lineal de los encoders en función de las cuentas.

Con un radio aproximado de 0.175m o 175mm, obtenemos:

$$Resolución\ en\ mm \approx 0.27\ [\frac{mm}{cuenta}]$$

Para fijar cada uno de los encoders a los ejes de las dos ruedas se ha diseñado una pieza para sujetar mediante tornillos ambos sensores (*Figura 10*). Las dos piezas impresas

son simétricas, de modo que los encoders puedan obtener las lecturas en el mismo sentido de giro.

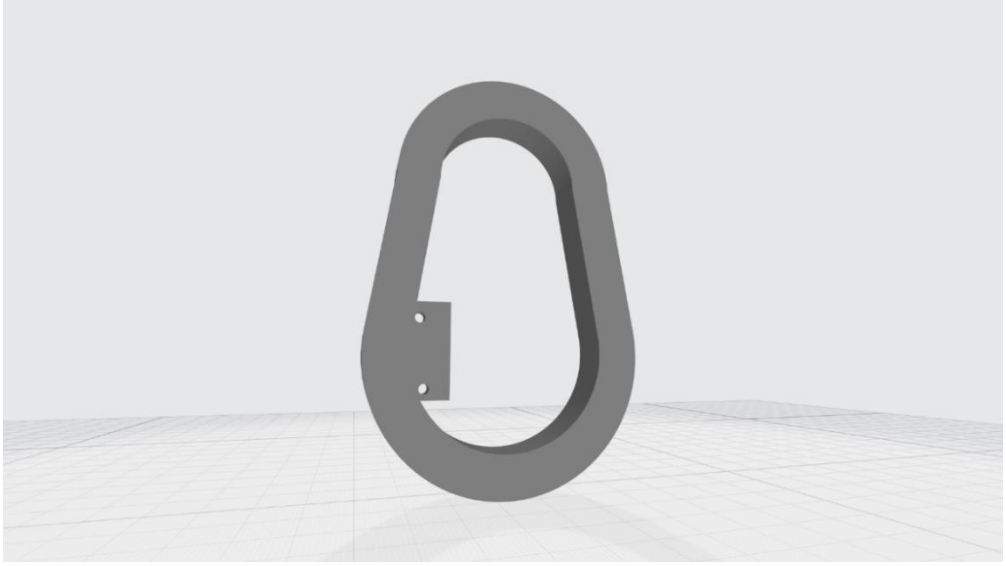


Figura 10. Pieza de sujeción encoders

En la siguiente *Figura 11* se muestra el encoder derecho sujeto a la pieza diseñada, que a su vez se encuentra encajada alrededor del eje del motor. La tolerancia para introducir la apertura del encoder en el disco codificado es bastante estrecha, de ahí la utilidad de los tornillos, que permiten cierta regulación. Los canales "A" y "B" del encoder son conectados directamente al Arduino, de modo que se pueda realizar la adquisición de los datos. Los pines utilizados del Arduino para este propósito tienen capacidad de interrupción, de modo que las lecturas de los encoders sean lo más precisas posibles. Al realizar la lectura del encoder, esta acción toma prioridad gracias a la capacidad de interrupción de los pines utilizados, evitando la pérdida de cuentas durante este proceso.

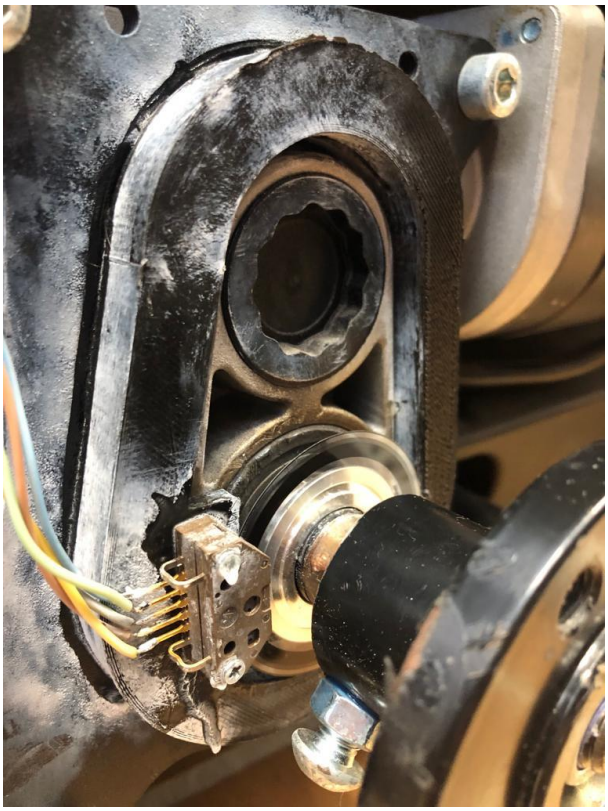


Figura 11. Encoder instalado en la pieza de sujeción diseñada

3.5 Periféricos

Para recibir las consignas de velocidad, se ha instalado un joystick analógico, mostrado en la *Figura 12*. Una vez alimentado, podemos leer valores analógicos generados al manipular el joystick utilizando el Arduino.

Figura 12. Joystick analógico utilizado para recibir inputs del usuario

Se ha implementado también un selector de velocidades, permitiendo elegir entre cuatro velocidades distintas, que actúan de valor máximo de las consignas posibles. Su funcionamiento se detalla en el apartado 4.3 Selección de velocidades y de modo de funcionamiento. El selector utilizado es el mostrado en la *Figura 13*. Este selector se implementa junto a un vúmetro digital, y un interruptor de dos posiciones que permite escoger el modo de funcionamiento de la silla. El funcionamiento de estos componentes se describe en el apartado 4.3 Selección de velocidades y de modo de funcionamiento.

Figura 13. Selector de tres canales y cuatro posiciones

Para encender y apagar el sistema, cortando la alimentación desde la batería, se ha utilizado un interruptor de palanca capaz de aguantar altas tensiones: 15 A – 125VAC, 10 A – 250VAC. El modelo utilizado es KN3(c) – 202. Se muestra en la *Figura 14*.

Figura 14. Interruptor dos posiciones KN3(c) – 202

3.6 Conexiones y funcionamiento

El controlador Sabertooth 2x32 toma como entrada la salida de nuestro controlador PI, el cual está implementado en un Arduino MEGA 2560, y cuyo valor depende del muestreo de la velocidad realizado a través de dos encoders ópticos incrementales. El controlador Sabertooth toma este comando y lo utiliza para inyectar el voltaje necesario para excitar los motores de las ruedas traseras en base a estos comandos.

Para poder enviar una consigna a nuestro controlador PI es necesario algún tipo de entrada física por parte del usuario, para ello se ha hecho uso de un joystick analógico, ya que es un componente simple de implementar y además su uso es muy frecuente en este tipo de sillas de ruedas eléctricas, alrededor del 80% de los usuarios utiliza este tipo de interfaz de control al utilizar una silla de estas características [4].

Un interruptor rotatorio permite seleccionar entre distintas velocidades máximas. La selección realizada se indica al usuario mediante el uso de un vúmetro digital. Para cambiar entre el modo manual, donde las consignas de velocidad son generadas por el

joystick analógico, y el modo automático, donde se obtienen desde un ordenador, un interruptor de dos posiciones está al alcance del usuario.

Inicialmente, el controlador Sabertooth también tenía la tarea de conectar y desconectar los frenos de la silla, pero debido a problemas detallados más adelante en el apartado 7.1 Frenos, se ha optado por utilizar el Arduino para comandar los frenos mediante un relé.

Para alimentar todos los dispositivos y los motores, dos baterías de ácido-plomo conectadas en serie suministran un voltaje de 24V. El controlador Sabertooth proporciona salidas de 5V una vez alimentado, de modo que se utilizan para poder suministrar voltaje a los dispositivos TTL y al microcontrolador Arduino.

El diagrama mostrado en la *Figura 15* pretende ilustrar las conexiones realizadas entre los componentes introducidos en los apartados anteriores, y también las interacciones definidas en este apartado.

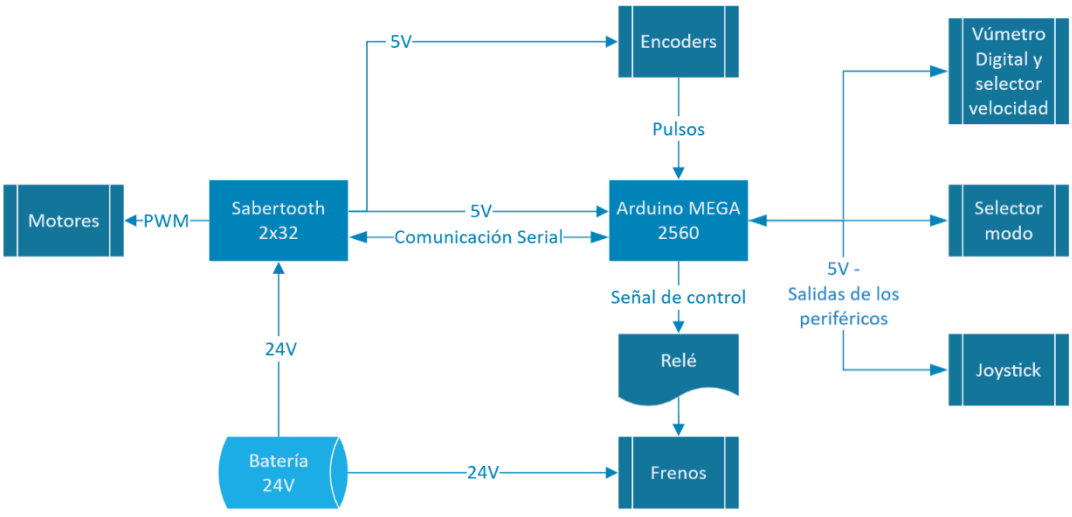


Figura 15. Diagrama de bloques de las conexiones

4. Diseño e implementación

4.1 Adquisición de datos y estimación de la velocidad

El primer paso para obtener la velocidad de las ruedas es realizar la lectura de los encoders. Como se ha mencionado anteriormente en este documento, los canales del encoder emiten pulsos relacionados con la lectura que han realizado del disco codificado. Con la ayuda de la librería Encoder [5], la lectura de los encoders a través de Arduino devuelve una cuenta de los pulsos, teniendo también en cuenta el sentido de giro.

Estas cuentas son almacenadas en un vector en el tiempo de muestreo que se ha determinado para realizar estas lecturas, que es de 20 milisegundos. Para realizar la lectura, la función de muestreo ha sido condicionada a una interrupción, la cual asegura que este tiempo de muestreo se cumple, tomando prioridad sobre el resto de las funciones del Arduino. Dicha interrupción es distinta a las interrupciones utilizadas para obtener la lectura de los encoders. Este tiempo de muestreo aporta la resolución suficiente sin "bloquear" la ejecución del resto de funciones.

El vector mencionado en el párrafo anterior almacena diez muestras de estas cuentas, las cuales son utilizadas para obtener un valor fiable del cambio entre cuentas a lo largo del tiempo. La diferencia entre cuentas se realiza entre el valor más nuevo y el más antiguo, con la ventaja de que este vector permite almacenar los datos que existen entre ambos, por lo que no se pierde resolución en el posterior cálculo de la velocidad. Teniendo en cuenta esto, aunque el tiempo de muestreo sea de 20 milisegundos, el tiempo entre diferencias es diez veces mayor, de 200 milisegundos. Por lo tanto, la resolución de la velocidad obtenida es de 200 milisegundos, aunque estemos realizando una lectura cada 20. En la *Figura 16* se ilustra el vector donde se almacenan las cuentas.

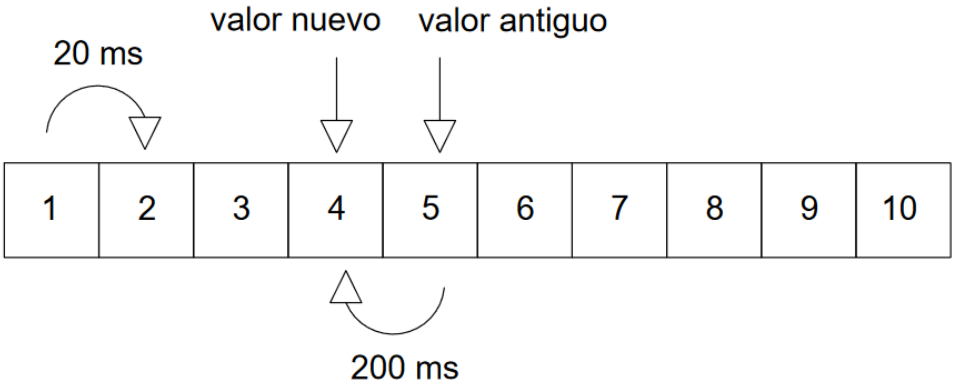


Figura 16. Ventana deslizante

Una vez obtenido el incremento entre cuentas podemos calcular la velocidad angular. Para ello utilizamos la siguiente expresión:

$$w\left[\frac{rad}{s}\right] = \frac{2\pi * \Delta cuentas}{resoluciónEncoder * t_{muestreo} * tamañoVentana}$$

Ecuación 2. Estimación de la velocidad a partir del incremento de cuentas obtenido

Donde la resolución del encoder es de 4096 cuentas por revolución, debido a que al utilizar la decodificación de cuadratura obtenemos cuatro veces la resolución, que en este encoder es de 1024 cuentas por revolución en la decodificación más simple. El tiempo de muestro en nuestro caso es de 20 milisegundos y el tamaño de la ventana es de 10 elementos. Una vez obtenida la velocidad angular, la velocidad lineal se obtiene instantáneamente multiplicando por el radio de la rueda.

La adquisición de los distintos valores en el tiempo durante las pruebas se ha realizado a través de *Processing Grapher*, software que permite almacenar en formato .CSV[3] los mensajes transmitidos por el Arduino al ordenador mediante la

[3] Comma-Separated Values. Es un tipo de documento para representar datos en forma de tabla, donde las columnas se delimitan por comas y las filas por saltos de línea.

comunicación serial [6]. Entre sus funciones destacan también la visualización en tiempo real de la evolución de los valores, así como la inspección de los archivos .CSV grabados con anterioridad. Estas funcionalidades resultaron extremadamente útiles a la hora de obtener una retroalimentación visual e instantáneamente interpretable de lo que estaba sucediendo durante la realización de los experimentos y las pruebas.

Figura 17. Ejemplo de visualización en Processing Grapher [6]

4.2 Input del usuario

Para generar las consignas de velocidad se hace uso de un joystick analógico. Este tipo de joysticks genera un voltaje regulado por dos potenciómetros: uno que define el eje *x*, y otro que define el eje *y*. Dichos voltajes se leen en el Arduino como un valor entero comprendido entre 0 y 1023. Para mapear estas lecturas a valores de consigna apropiados se ha optado por utilizar un algoritmo que [7] concluye como uno de los preferidos por los usuarios de este tipo de sillas de ruedas tras realizar un estudio con varios modelos de mapeo implementados por los autores. En el estudio mencionado, se ha pedido a distintos usuarios que navegasen a través de un recorrido en un entorno simulado con distintos modelos de mapeo propuestos. Se han recogido sus opiniones en un estudio estadístico y se ha determinado que el modelo de mapeo que se pretende utilizar en este proyecto es el preferido por la mayoría de los usuarios.

El algoritmo de mapeo en cuestión establece las velocidades de las ruedas mediante las siguientes ecuaciones:

$$\begin{cases} R = y - nx \\ L = y + nx \end{cases}$$

Ecuación 3. Asignación de consignas de velocidad a las ruedas

Siendo *R* y *L* las velocidades de las ruedas derecha e izquierda respectivamente desde el punto de vista del usuario, y *nx* un valor obtenido de la siguiente expresión:

$$nx = \begin{cases} u_1 c_{point} + (x - c_{point}) * u_2 & if\ x > c_{point} \\ -u_1 c_{point} + (x + c_{point}) * u_2 & if\ x < -c_{point} \\ u_1 * x \quad if\ -c_{point} < x < c_{point} \end{cases}$$

Ecuación 4. Obtención del valor 'nx' para el mapeo de la velocidad

Donde *cpoint*, *u1* y *u2* son valores ajustados empíricamente por los autores y tienen los siguientes valores:

$$c_{point} = 0.2, \qquad u_1 = 0.5, \qquad u_2 = 0.25$$

Para utilizar estos valores, las posiciones *x* e *y* han sido normalizadas en el intervalo [-1, 1]. Posteriormente estos valores son mapeados linealmente hasta abarcar un intervalo comprendido entre las consignas máxima y mínima aceptadas.

Representando los valores mapeados de la velocidad en las dos ruedas con este algoritmo obtenemos el siguiente resultado, ilustrado por la *Figura 18*.

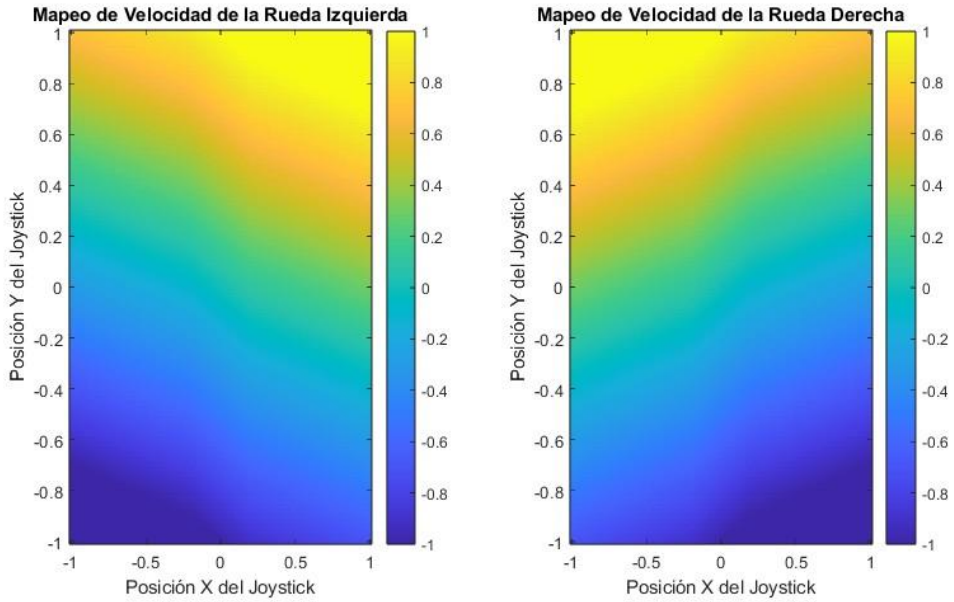


Figura 18. Consignas de velocidad de las ruedas en función de la posición del joystick.

Mediante este mapeo podemos controlar la silla como si se tratara de un robot diferencial, de modo que los giros dependen de las diferencias en velocidades entre ambas ruedas. Tal y como se puede observar en la *Figura 18*: al variar la posición x del joystick para indicar un giro, observamos que las velocidades son distintas en las ruedas para una misma posición y del joystick. Con este mapeo la consigna para la rotación pura es bastante inferior en módulo a la velocidad máxima posible, por lo que la silla, dependiendo de en qué posición estén orientadas las ruedas guía, puede llegar a tener bastantes problemas para comenzar la rotación.

Aunque en la figura no es inmediatamente apreciable, en este tipo de joysticks analógicos es importante incluir una zona muerta en el mapeo, de modo que al estar el joystick en la posición de reposo no se envíe consigna alguna al controlador. Debido al uso continuado de joysticks analógicos, el desgaste y la suciedad pueden crear una situación en la cual la posición de reposo se vea desplazada ligeramente. Al mapear los valores analógicos leídos a intervalos más pequeños (aunque intuitivos para nosotros), los valores truncados y redondeados pueden acentuar el problema inicial.

4.3 Selección de velocidades y de modo de funcionamiento

Para indicar la velocidad seleccionada dentro de las posibles, se ha incluido un vúmetro digital compuesto de cuatro LED y un selector de doce posiciones que ha sido codificado. Se ha instalado de modo que el usuario tenga acceso a ellos sentado desde la silla, como se muestra en la *Figura 19*, en la que también se incluye el interruptor que permite seleccionar el modo de funcionamiento.

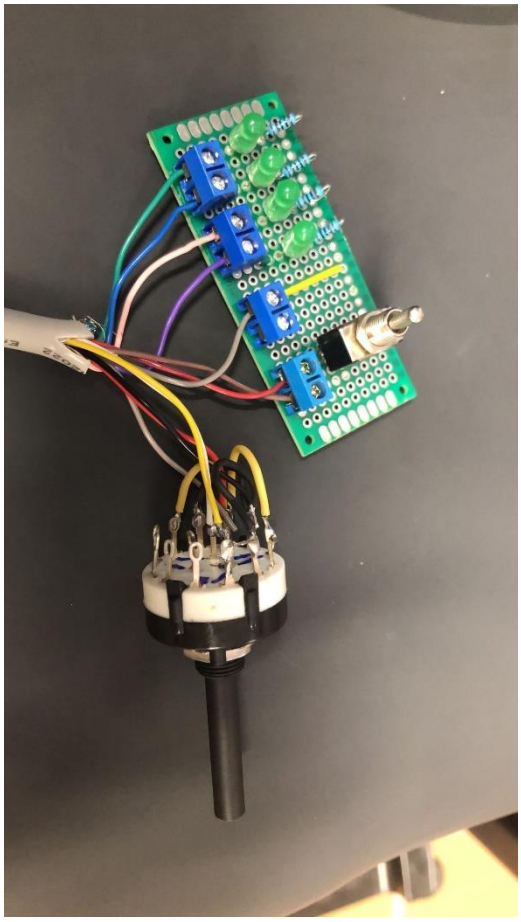


Figura 19. Vúmetro digital, selector de velocidad e interruptor modo automático-manual

El selector ha sido interconectado de tal manera que a través de dos canales es capaz de enviar en codificación binaria un número relacionado con cada una de las cuatro posiciones posibles. Una vez recibido este número binario se interpreta la selección y se

encienden los LED correspondientes. Finalmente, se actualiza la consigna máxima del lazo de control, quedando fijada la velocidad máxima alcanzable para esa selección.

Un interruptor establece el modo automático o manual de la silla. En el modo manual las consignas del lazo de control vienen dadas por un joystick analógico, mientras que para el modo automático estas consignas provienen de un ordenador, de modo que el sistema queda preparado para la implementación de algoritmos de navegación automáticos, o para controlar la silla directamente a través de las teclas de dirección, por ejemplo.

Un esquema del conjunto se proporciona en la *Figura 20*

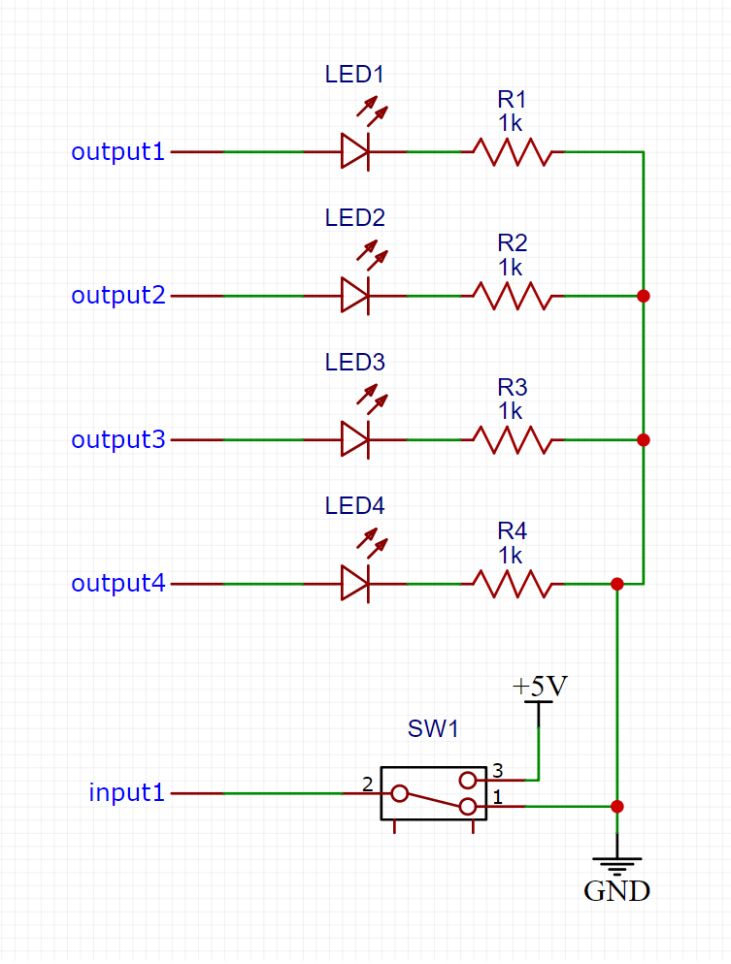


Figura 20. Esquema eléctrico del vúmetro digital e interruptor modo automático-manual

5. Control

5.1 Control PID

El controlador PID (Proportional, Integral, Derivative) tiene un largo recorrido histórico dentro del contexto del control por realimentación. Este controlador nace de la experiencia y de un uso extensivo de la prueba y el error. La acción proporcional resulta la más intuitiva de las tres: acerca nuestro sistema linealmente a la referencia que le hemos proporcionado, pero nunca termina de eliminar el error en el estado estacionario. De esta limitación nace la acción integral, que es capaz de eliminar este error, pero a costa de una respuesta transitoria algo más impredecible, poniendo incluso en riesgo la estabilidad del sistema en algunos casos. Por último, para combatir las limitaciones de la acción integral, la acción derivativa intenta anticiparse a la evolución dinámica del sistema, estabilizando el sistema y evitando sobrepasamientos en la respuesta [8].

Una posible expresión del controlador PID en el dominio temporal es la siguiente:

$$PID(t) = Kp * e(t) + Ki \int_{t0}^{t} e(\tau) * d\tau + Kd * \dot{e}(t)$$

Ecuación 5. Controlador PID(t) en configuración paralela

Donde *Kp*, *Ki* y *Kd* son respectivamente la ganancia proporcional, integral y derivativa. Observamos que el papel que juega cada uno de los tres términos depende del tipo de error. Mientras que en el término proporcional el error depende sólo del instante anterior, en el término integral el error se acumula en el tiempo, lo que le permite aumentar su acción cuando la referencia no esté siendo alcanzada. El término derivativo "observa" la tendencia del error, de modo que pueda anticiparse a cambios bruscos.

Se entiende por sintonizar un controlador PID la búsqueda de una combinación de las tres ganancias que logre la respuesta deseada del sistema que queremos controlar. A menudo el método de prueba y error conforma una gran parte de la búsqueda de estos valores. Para lograr resultados más sofisticados y que tengan en cuenta otros criterios de diseño, como pueden ser el rechazo a perturbaciones, márgenes de fase y ganancia u otros parámetros críticos, existen una infinidad de métodos de análisis y sintonización.

La función de transferencia en el dominio de Laplace para un controlador PID puede ser descrita de la siguiente manera en la conocida como forma paralela:

$$PID(s) = Kp * \frac{Ki}{s} + Kd * s$$

Ecuación 6. Controlador PID(s) en configuración paralela.

La representación en diagrama de bloques del controlador PID en el dominio de Laplace se ilustra en la *Figura 21* .

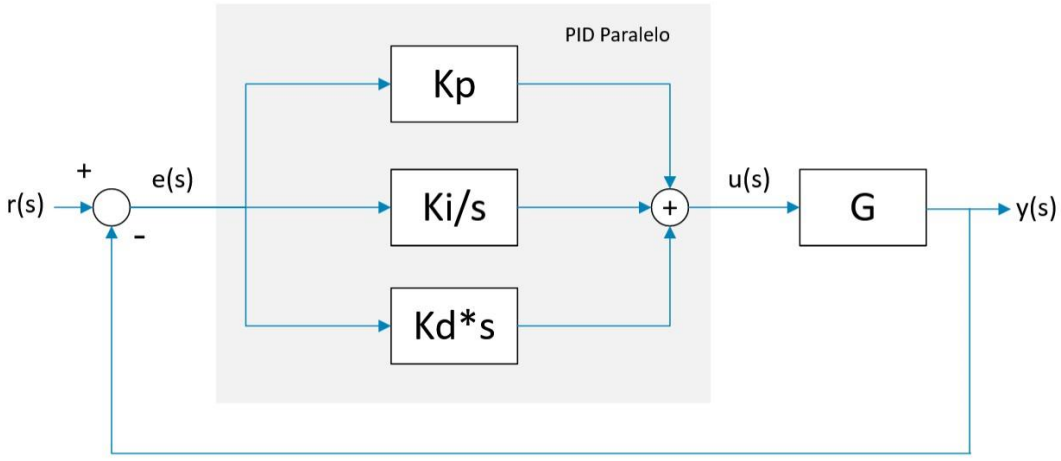


Figura 21. Controlador PID paralelo en el dominio de Laplace

5.2 Anti windup

Nuestro controlador se satura al alcanzar valores que superan los límites que le hemos impuesto (delimitados por el controlador Sabertooth). Cuando esta saturación ocurre, el lazo de control queda abierto en la práctica. Si el error permanece constante o aumenta, la acción integral lo seguirá haciendo indefinidamente también. En el momento que nuestra consigna sea distinta nuestro controlador seguirá saturado y este problema puede generar situaciones peligrosas debido al gran sobrepasamiento que generará la respuesta del sistema. Para evitar estas situaciones se hace uso de algoritmos *anti-windup*,

los cuales son implementados de maneras muy diversas, pero cuyo objetivo es limitar o eliminar la acción integral ante posibles situaciones de saturación del controlador [8].

En nuestro proyecto es importante eliminar cualquier potencial pérdida de control de la silla de ruedas, y para ello se ha hecho uso de estos algoritmos. El primero de ellos consiste en la eliminación del error integral acumulado en el tiempo después de que la silla de ruedas comience a moverse desde una posición estacionaria. Para superar la fricción estática y la inercia de los motores, la acción integral aporta una potencia extra para comenzar a mover las ruedas, pero una vez la silla comienza a moverse el error integral acumulado puede dar la impresión de una aceleración muy brusca en los primeros instantes. Para limitar la acción integral borramos el error integral acumulado cada vez que se detecta el cambio de estado de reposo a cuando nos movemos. Aquí hacemos uso del incremento de cuentas utilizado para medir la velocidad; establecemos un umbral para el cual consideramos que nos estamos moviendo, y eliminamos el error integral si es así. Este umbral ha sido determinado mediante prueba y error hasta que se han logrado unos valores que permiten una transición cómoda desde el reposo.

Sin embargo, si nos limitásemos a borrar el error integral cada vez que el incremento en cuentas superase cierto umbral, la silla sería controlada enteramente por la acción proporcional durante la conducción. La manera que se ha ideado para evitar esta situación es la implementación de un nuevo umbral, mayor que el anterior, de modo que existe una “zona muerta” de la acción integral entre ambos. De esta manera la aceleración de la silla es gradual hasta que se alcanza la consigna que le hemos enviado, pero sin acelerones bruscos al inicio.

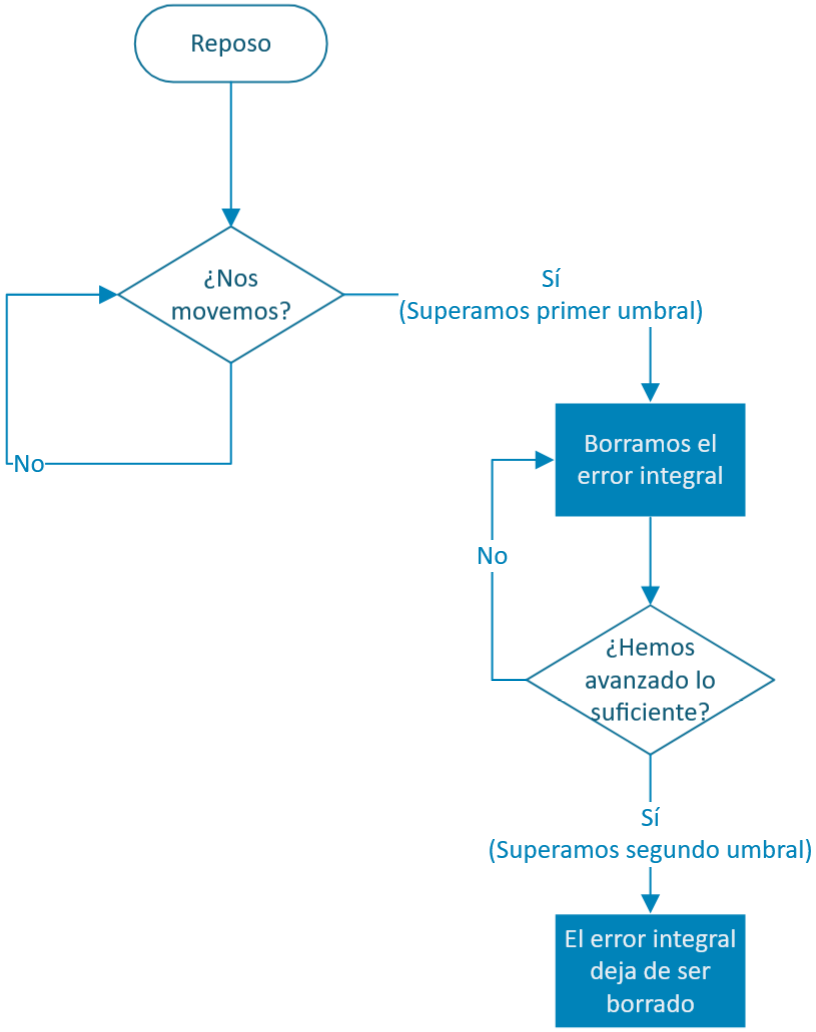


Figura 22. Primer método anti-windup implementado.

Otro método muy utilizado es la limitación del error integral o de la propia acción integral. Establecer límites a alguno de los valores mencionados anteriormente, del mismo modo que imponemos un límite en el controlador, puede ayudar a mantener estable el sistema. Por mucho tiempo que el actuador se encuentre saturado la acción integral permanecerá limitada en el valor que causó la saturación. Este es el segundo método implementado en nuestro código: comprueba si el controlador ha llegado a los límites preestablecidos y deja de acumular error integral, si este es el caso, manteniendo la acción integral constante en el tiempo. De esta manera evitamos que el error integral continue incrementándose indefinidamente durante el tiempo que el controlador esté saturado.

5.3 Primera identificación del sistema

Para realizar una estimación inicial de la planta del sistema se ha llevado a cabo un experimento en lazo abierto con las ruedas motrices elevadas, de modo que la fricción, el peso del usuario y de la silla no son factores considerados en esta primera estimación. Aun así, este experimento tiene la función de actuar como una primera toma de contacto con el sistema y con las herramientas utilizadas para su identificación.

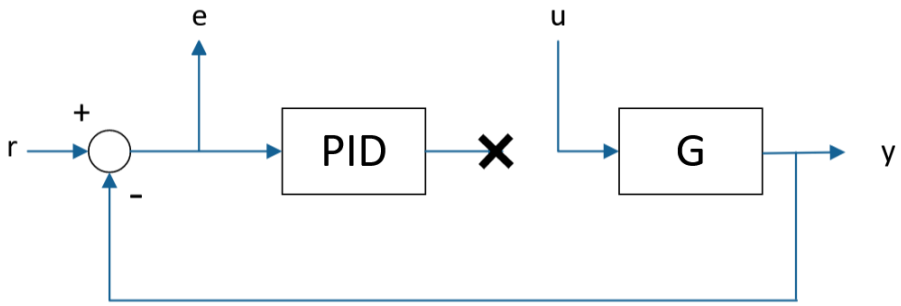


Figura 23. Lazo abierto

Como se pretende ilustrar mediante la *Figura 23*, este experimento consiste en excitar la planta directamente mediante entradas específicas y observar cómo se comporta ante ellas sin ninguna realimentación y, por lo tanto, sin acción del controlador. Un tipo de entrada muy utilizada en este tipo de experimentos es la entrada de tipo escalón, la cual consiste en una excitación a un valor constante, sin transición gradual desde el valor inicial hacia el valor de excitación. También se utilizará una entrada de tipo rampa, la cual consiste en una excitación con una pendiente constante.

Las entradas del sistema para este experimento serán entonces las mostradas en la *Figura 24*.

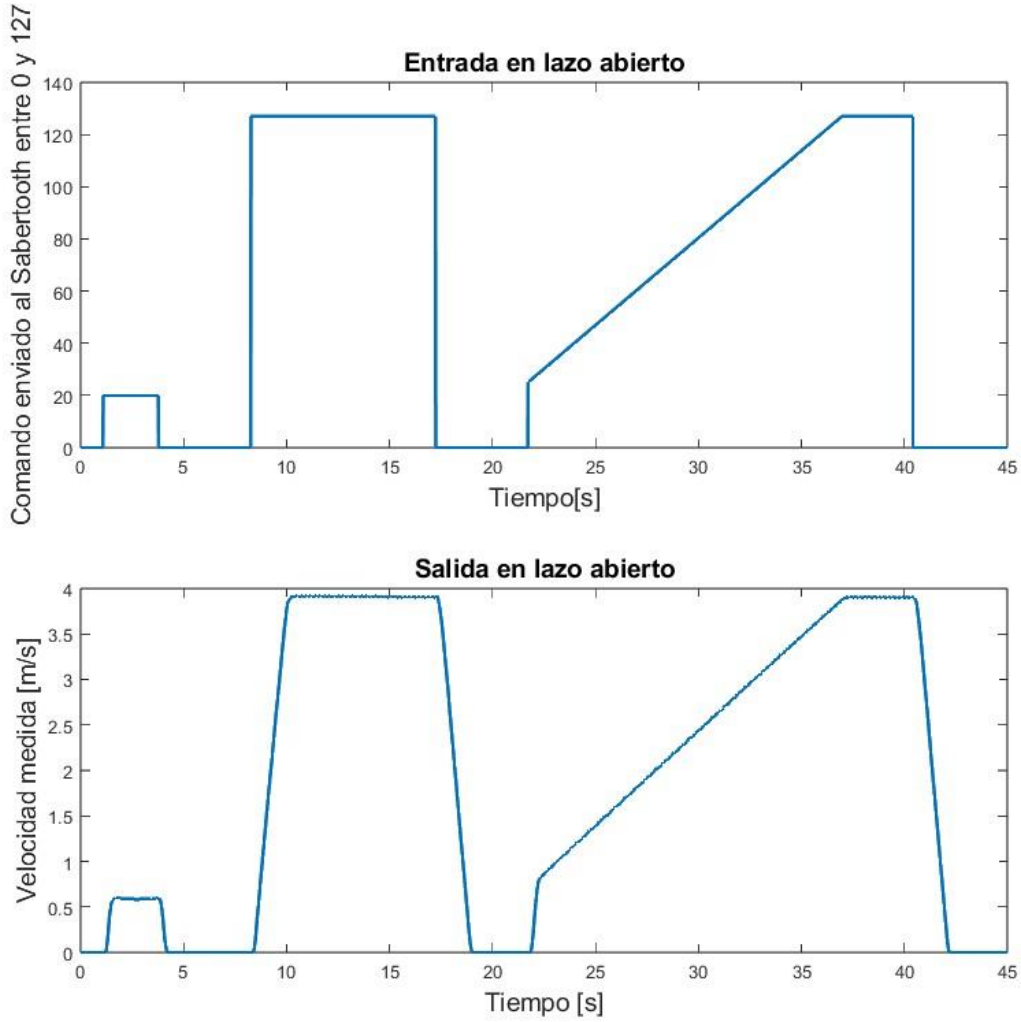


Figura 24. Experimento en lazo abierto sin carga en las ruedas

Como es de esperar, sin carga en las ruedas la respuesta es rápida y se ajusta en gran medida al comando enviado. La salida se satura cerca de los 4 m/s, por culpa principalmente de que esta es la máxima salida posible de nuestro controlador con la configuración actual del Sabertooth, que corresponde al comando con valor 127.

Esta velocidad máxima de 14 km/h resulta realmente excesiva para el recinto en el que vamos a realizar las pruebas, por lo que el controlador se sintonizará teniendo en cuenta velocidades más cómodas para navegar por interiores. Sobre todo, se tendrán en cuenta los 0.5m/s (1.8 km/h) y 1m/s (3.6 km/h).

Una vez obtenida la respuesta de nuestro sistema ante el experimento diseñado, es necesario realizar una identificación de éste. Esta identificación se ha realizado

mediante la filosofía *blackbox*, la cual asume que no conocemos nada acerca del sistema que estamos identificando. La filosofía opuesta a ésta es la conocida como w*hitebox*, mediante la cual el modelo es identificado y parametrizado mediante ecuaciones diferenciales obtenidas de los propios principios físicos que gobiernan el sistema. La combinación de ambos es habitual y se denomina g*reybox*.

El software utilizado para llevar a cabo esta tarea de identificación *blackbox* es *Matlab*, específicamente haciendo uso de las herramientas que ofrece la *System Identification Toolbox*.

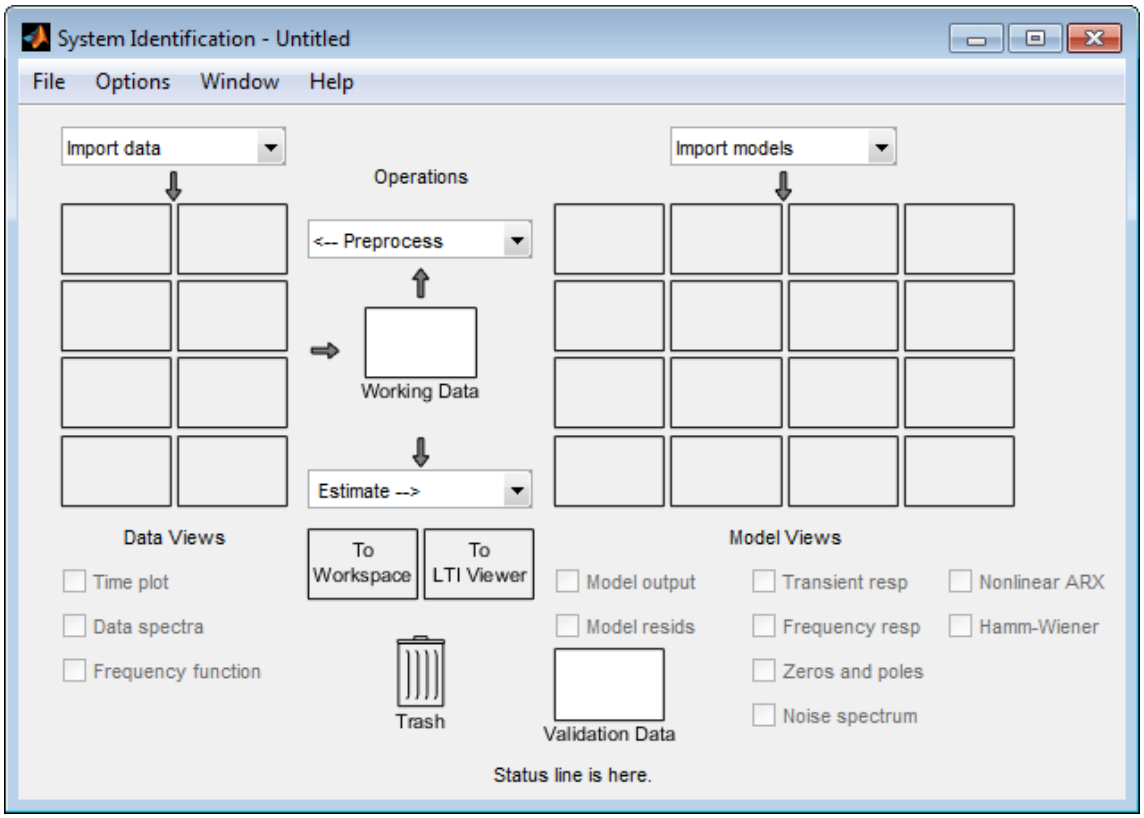


Figura 25. Ventana principal de la System Identification App

Esta aplicación sugiere distintos modelos en base a los datos de entrada-salida proporcionados por el usuario. Mediante un variado número de algoritmos reduce el error determinado por una función de coste, y aporta modelos de función de transferencia,

espacio de estados, etc. pero también algunos más avanzados para sistemas no lineales, como aquellos con estructura Hammerstein-Wiener o ARX.

Las siguientes estructuras (*Tabla 1*) han sido estimadas mediante la aplicación, y su porcentaje de ajuste con respecto a los datos de validación es ilustrado en la *Figura 26*:

Tabla 1. Detalles de los modelos propuestos, primera identificación.

Nombre	Estructura	Porcentaje de ajuste
tf1	Función de transferencia con 2 polos y ningún cero	91.04%
tf2	Función de transferencia con 2 polos y 1 cero	91.20%
ss1	Espacio de estados de orden 3	90.43%
ss2	Espacio de estados de orden 4	90.54%

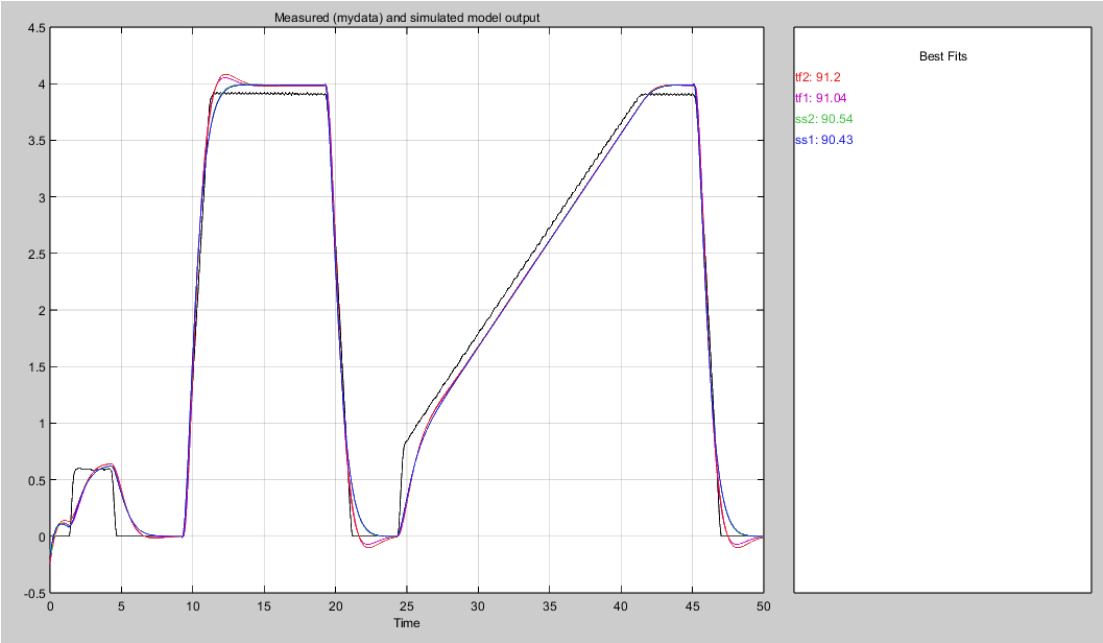


Figura 26. Ajuste de distintos modelos para el experimento de lazo abierto (tf1: 91.04%, tf2: 91.2%, ss1: 90.43%, ss2: 90.54%)

Las funciones de transferencia ofrecen resultados más satisfactorios en cuanto a lo que el error se refiere, pero presentan un sobrepasamiento ante las entradas que no está presente en los modelos basados en espacio de estados como podemos observar en la *Figura 27*, la cual hace énfasis en la salida para la segunda entrada escalón del experimento junto a los distintos modelos propuestos. Aunque ninguno de los sistemas ha sido capaz de replicar las condiciones iniciales satisfactoriamente, la respuesta ante la segunda entrada escalón parece bastante satisfactoria, aunque con cierto desplazamiento del valor de referencia debido a las no linealidades introducidas por la saturación de la entrada. Las oscilaciones en la salida real se deben a la inercia de las ruedas al estar elevadas sin carga ninguna.

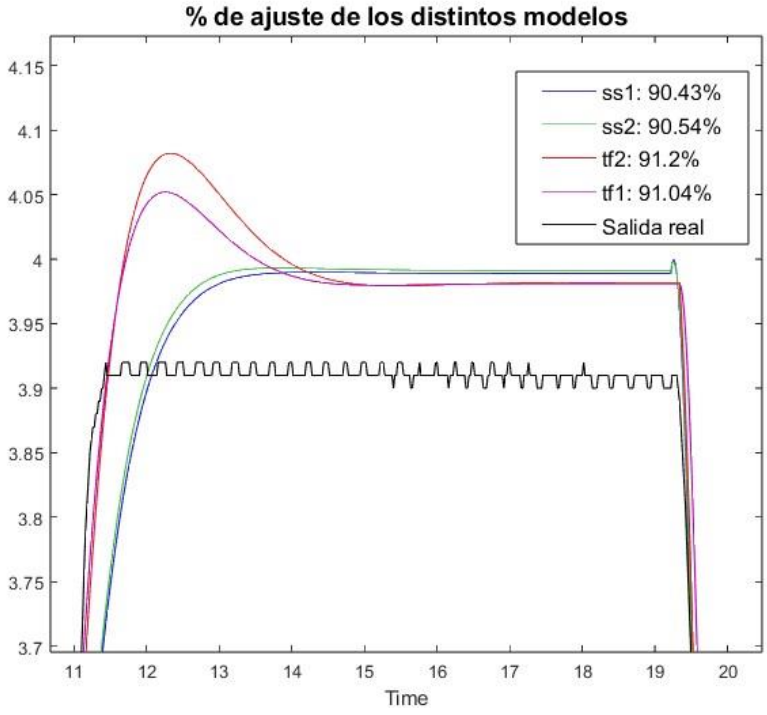


Figura 27. Porcentaje de ajuste de los distintos modelos en la segunda entrada escalón.

Dado que en este caso los modelos de espacio de estados propuestos son de tercer orden, se ha optado por utilizar tf2 al ser una función de transferencia de segundo orden, lo cual facilitará la sintonización del controlador.

Por lo tanto, el primer modelo identificado tiene la siguiente función de transferencia:

$$tf2(s) = \frac{0.007325s + 0.07676}{s^2 + 2.384s + 2.448} * e^{-0.12s}$$

5.4 Sintonización inicial del controlador

Se han utilizado dos alternativas para obtener los distintos parámetros del controlador. Una consiste en el uso de la aplicación *PIDTuner* incluida con Matlab, que permite una rápida sintonización de la planta identificada en el mismo entorno de trabajo, basándose principalmente en la velocidad y la robustez que busquemos. La segunda alternativa son las reglas de sintonización "AMIGO[4]" [9].

Las reglas "AMIGO" ofrecen una alternativa más actual a las famosas reglas de Ziegler-Nichols, en las cuales se basan. Ayudan a determinar parámetros de sintonización en controladores PI para una gran multitud de procesos de manera genérica, tomando ventaja de la facilidad de estimación de los parámetros necesarios para estimar los valores del controlador mediante estas reglas.

La sintonización mediante la aplicación *PIDTuner* se realiza visualmente, ajustando la velocidad y la robustez del sistema. Mediante prueba y error obtenemos la respuesta mostrada en la *Figura 28*. Procuramos evitar el sobrepasamiento, ya que no nos interesa que la silla de ruedas exhiba comportamientos bruscos al enviarle comandos. Tampoco es realmente necesario que el tiempo de estacionamiento sea lo más pequeño posible, ya que una gran aceleración de la silla resulta indeseable para el usuario.

[4] Approximate M-Constrained Integral Gain Optimization

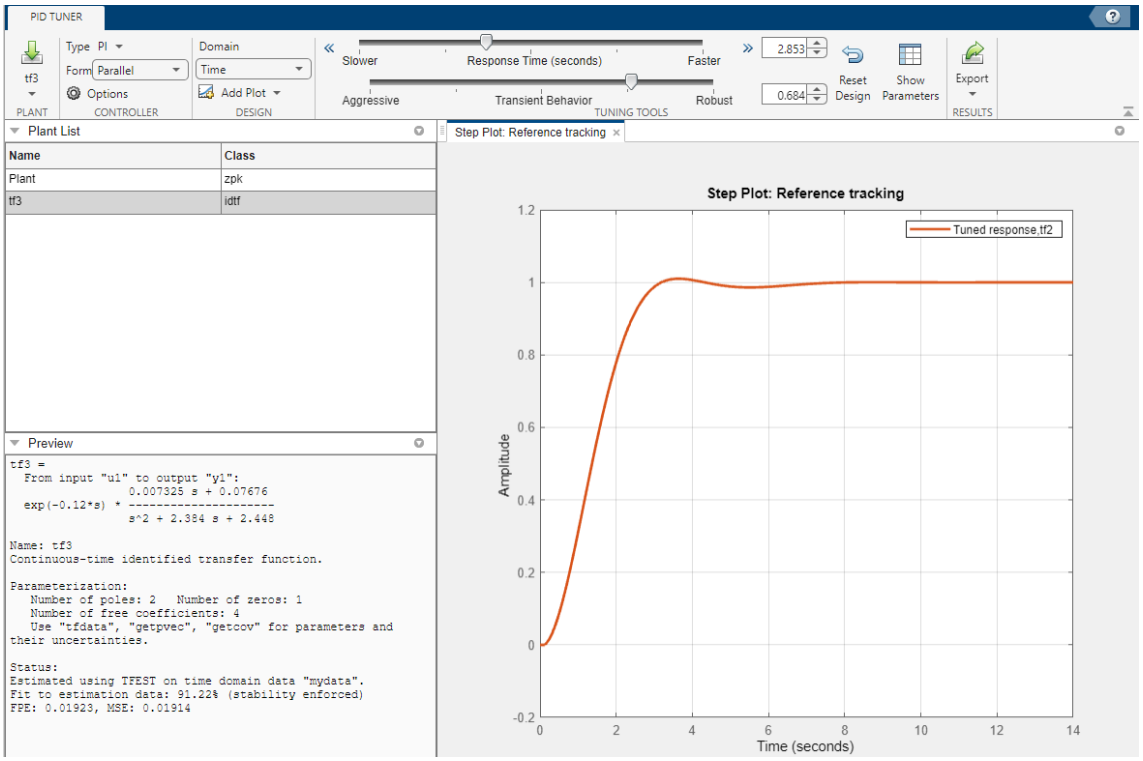


Figura 28 Ventana principal de la aplicación PIDTuner. Sintonización del controlador PI para la planta estimada.

Por lo tanto, los valores del controlador PI obtenidos mediante este método son los siguientes:

$$Kp = 12, \qquad Ki = 22$$

Las reglas AMIGO dependen esencialmente de tres parámetros: la ganancia estática, el tiempo muerto aparente, y la constante de tiempo aparente. Nosotros nos referiremos al parámetro de la ganancia estática como “K”, para evitar confusión con la ganancia proporcional del controlador, teniendo en cuenta que [9] utiliza “Kp” para referirse a la ganancia estática, y “K” para la ganancia proporcional del controlador.

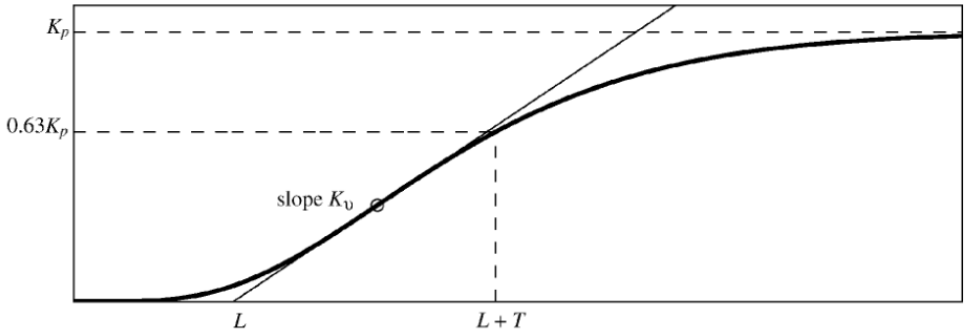


Figura 29. Estimación de ganancia estática "Kp", tiempo muerto aparente "L" y constante de tiempo aparente "T" para una entrada escalón. [9]

En la *Figura 29* se ilustra cómo obtener los parámetros necesarios para aplicar las reglas AMIGO y, en la *Figura 30*, se ha hecho una estimación de dichos parámetros en la planta que hemos identificado con anterioridad. Los parámetros estimados son, por lo tanto:

$$Kp = 0.0313, \qquad L = 0.12, \qquad T = 1.06$$

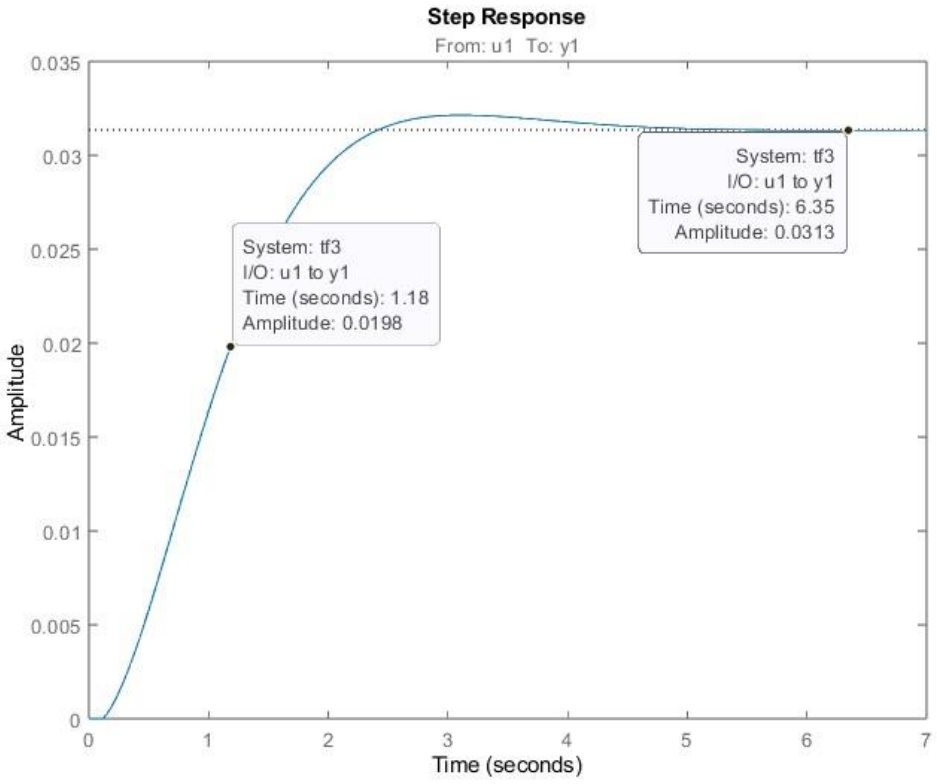


Figura 30. Estimación de los parámetros que caracterizan la respuesta de la planta ante la entrada escalón unitario.

Las reglas AMIGO son las siguientes:

$$K_p = \begin{cases} \dfrac{0.35}{K_v L} - \dfrac{0.6}{K}, & for\ L < \dfrac{T}{6} \\ \dfrac{0.25}{KL}, & for\ \dfrac{T}{6} < L < T \\ \dfrac{0.1T}{KL} + \dfrac{0.15}{K}, & for\ T < L \end{cases}$$

Ecuación 7. Obtención de la ganancia proporcional mediante las reglas AMIGO

$$T_i = \begin{cases} 7L, & for\ L < 0.11T \\ 0.8T, & for\ 0.11T < L < T \\ 0.3L + 0.5T, & for\ T < L \end{cases}$$

Ecuación 8. Obtención del tiempo integral mediante las reglas AMIGO

"Ti" es fácilmente convertible a "Ki" mediante la siguiente relación: $K_i = \frac{K}{T_i}$

En esta primera estimación usando este método, obtenemos los siguientes valores para el controlador PI:

$$Kp = 80, \qquad Ki = 94$$

5.5 Resultados de la primera sintonización

Las siguientes pruebas se han realizado con una persona dirigiendo la silla de ruedas a través de un espacio abierto, implementando en el controlador las ganancias obtenidas durante la primera sintonización en vacío. Al variar tanto las condiciones, lo que se busca de los controladores propuestos es un punto de partida para poder evaluar el comportamiento de la silla de ruedas ante distintos valores para las ganancias del controlador. Las muestras en formato .CSV se han obtenido con un tiempo de muestreo de 50 ms.

En la *Figura 31* observamos un sobrepasamiento de aproximadamente el 60% para el controlador con ganancias *Kp = 12* y *Ki = 20*. Añadido a este sobrepasamiento también tenemos un comportamiento oscilatorio que, aunque aparentemente estable, no resulta nada deseable.

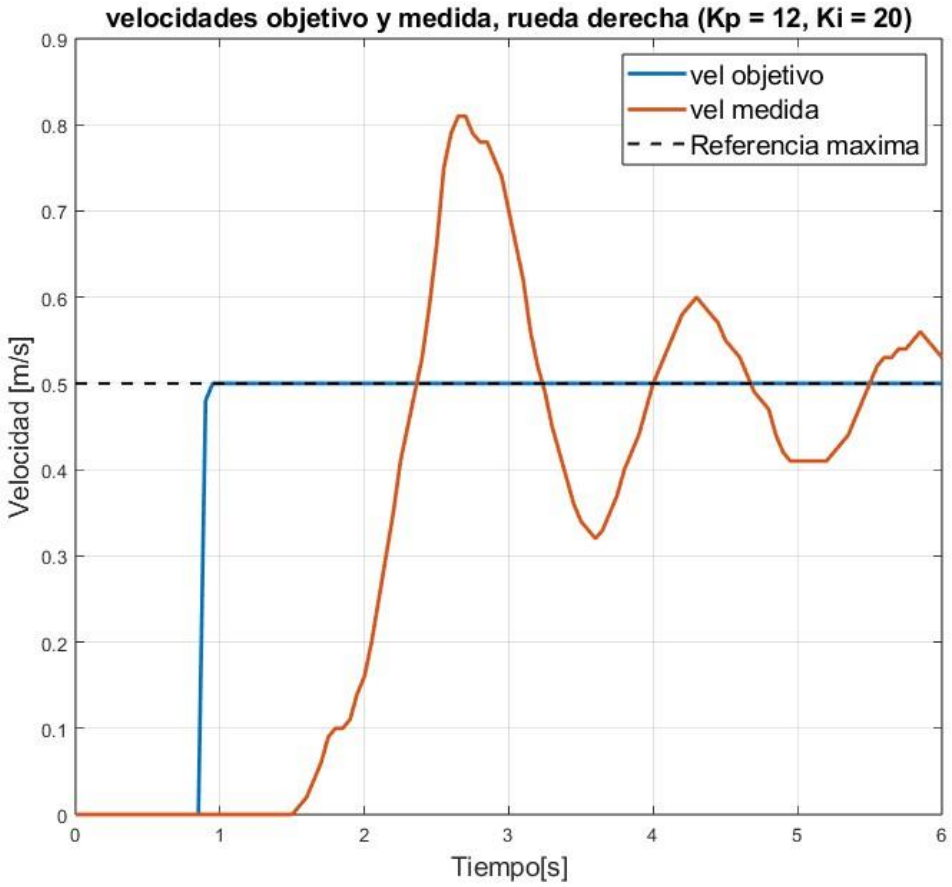


Figura 31 Velocidades objetivo y medida de la rueda derecha obtenidas durante las pruebas reales (Kp = 12, Ki = 20)

Los resultados obtenidos de las pruebas realizadas con el segundo controlador con ganancias $Kp = 80$ y $Ki = 94$ también han resultado insatisfactorios. Con unos valores tan elevados de las ganancias la respuesta del sistema es desmedida, con un sobrepasamiento de aproximadamente el 75% para la consigna de 1 m/s como se puede apreciar en la *Figura 32*.

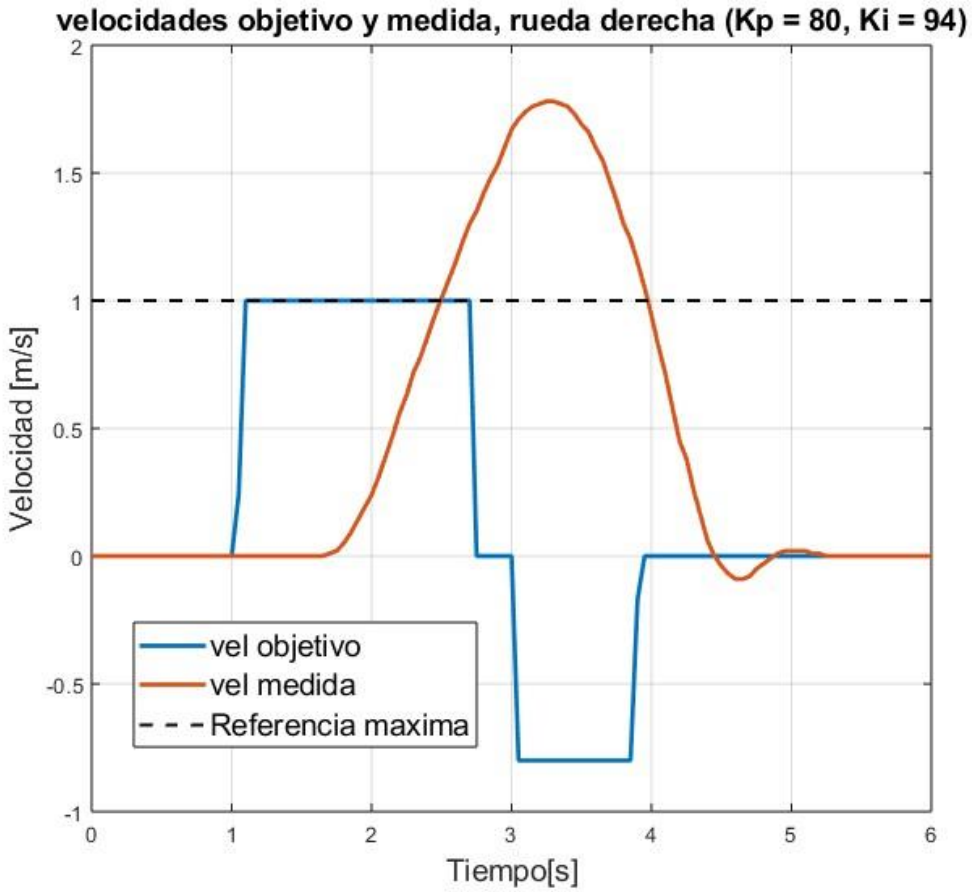


Figura 32. Velocidades objetivo y medida de la rueda derecha obtenidas durante las pruebas reales (Kp = 80, Ki = 94)

De estos resultados podemos extraer algunas conclusiones, especialmente del primer controlador:

- La ganancia integral debería ser inferior a *Ki = 20* para evitar el comportamiento oscilatorio.
- La ganancia proporcional debería ser bastante inferior a *Kp = 80* para evitar sobrepasamientos considerables.
- Tal y como apreciamos en la *Figura 33*, la acción del controlador está dominada por la acción integral en el primer controlador propuesto. Por lo tanto, la ganancia *Kp = 12* es insuficiente, especialmente en consignas de velocidad inferiores. Cabe mencionar que la puesta a cero de la acción integral tras el primer pico de la acción del controlador es producto de uno de los métodos *anti-windup* utilizado, el cual

borra el error integral una vez se pasa cierto umbral que determina el movimiento de la silla, tal y como se detalla en el apartado 5.2 Anti windup.

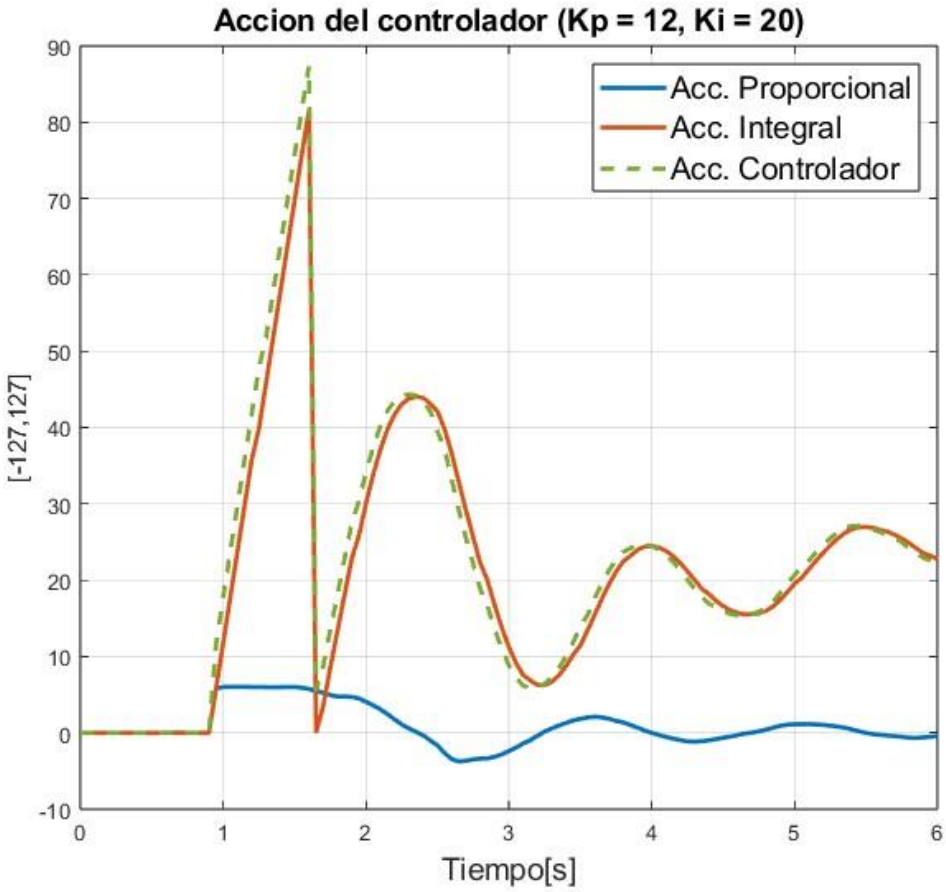


Figura 33. Acción del primer controlador propuesto (Kp = 12, Ki = 20)

5.6 Segunda identificación del sistema

Como hemos podido observar en la primera iteración de la sintonización del controlador PI, las condiciones de carga y el comportamiento de las ruedas guía de la silla dificultan la elección de los parámetros. Por lo tanto, se ha realizado una nueva identificación, realizando también un experimento en lazo abierto, pero con más limitaciones, dado que al tener que excitar lo máximo posible el sistema para obtener una buena identificación, las condiciones eran algo más inseguras.

En la siguiente *Figura 34* podemos observar la entrada escalón que se ha pretendido emular mediante el joystick, y la velocidad alcanzada antes de tener que frenar la silla. Como se ha mencionado anteriormente, se ha pretendido excitar el sistema hasta la saturación del actuador, de modo que la ganancia estática de la identificación fuera lo más cercana a la real posible.

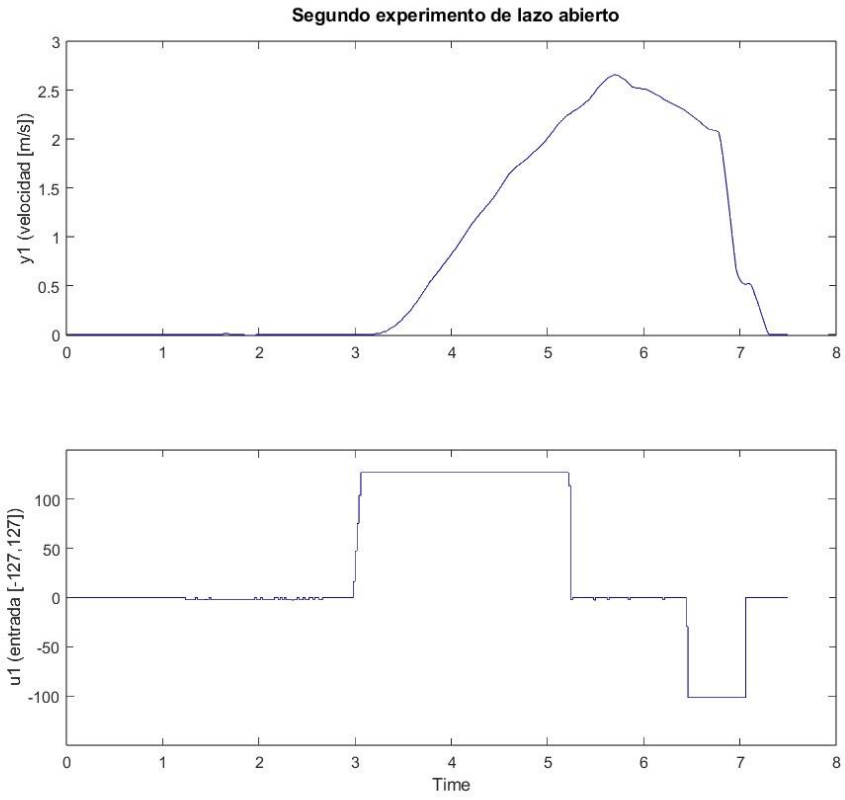


Figura 34. Segundo experimento de lazo abierto. Salida del sistema y1 en [m/s] en la primera gráfica, y entrada u1 en la segunda.

La estructura, nombres y porcentaje de ajuste de los distintos modelos propuestos por MATLAB se ofrece en la *Tabla 2*.

Siguiendo el mismo flujo de trabajo que en la primera identificación, se ha hecho uso nuevamente de la *System Identification Toolbox*, y se ha procurado imitar la misma estructura obtenida en la primera identificación de dos polos, pero también se ha probado a eliminar el cero de la estructura en tf2, sin embargo, la diferencia entre ambos modelos no ha sido significativa (*Figura 35*). En este caso, el modelo de espacio de estados propuesto no ha sido capaz de ajustarse a los datos, incluso siendo de tercer orden. Se ha optado por escoger el siguiente modelo ofrecido por tf1:

$$tf1(s) = \frac{0.0058s + 0.0457}{s^2 + 2.944s + 1.587} * e^{-0.32s}$$

Tabla 2. Detalles de los modelos propuestos, segunda identificación.

Nombre	**Estructura**	**Porcentaje de ajuste**
tf1	Función de transferencia con 2 polos y 1 cero	82.73%
tf2	Función de transferencia con 2 polos y ningún cero	82.42%
ss1	Espacio de estados de orden 3	65.88%

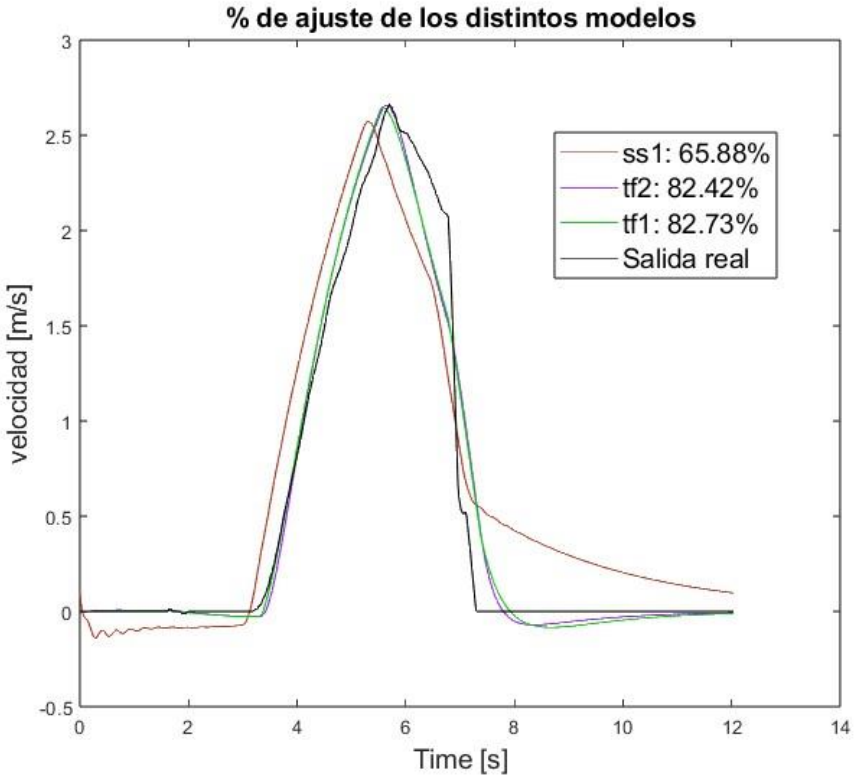


Figura 35. Porcentaje de ajuste de los distintos modelos en la segunda identificación.

5.7 Segunda sintonización

Una vez obtenida la nueva identificación de la planta se han seguido las mismas pautas que en la primera sintonización: se ha hecho uso de la aplicación PIDTuner y también de las reglas de sintonización "AMIGO".

Utilizando PIDTuner y priorizando un sobrepasamiento lo más pequeño posible obtenemos las siguientes ganancias para el controlador:

$$Kp = 33, \qquad Ki = 22$$

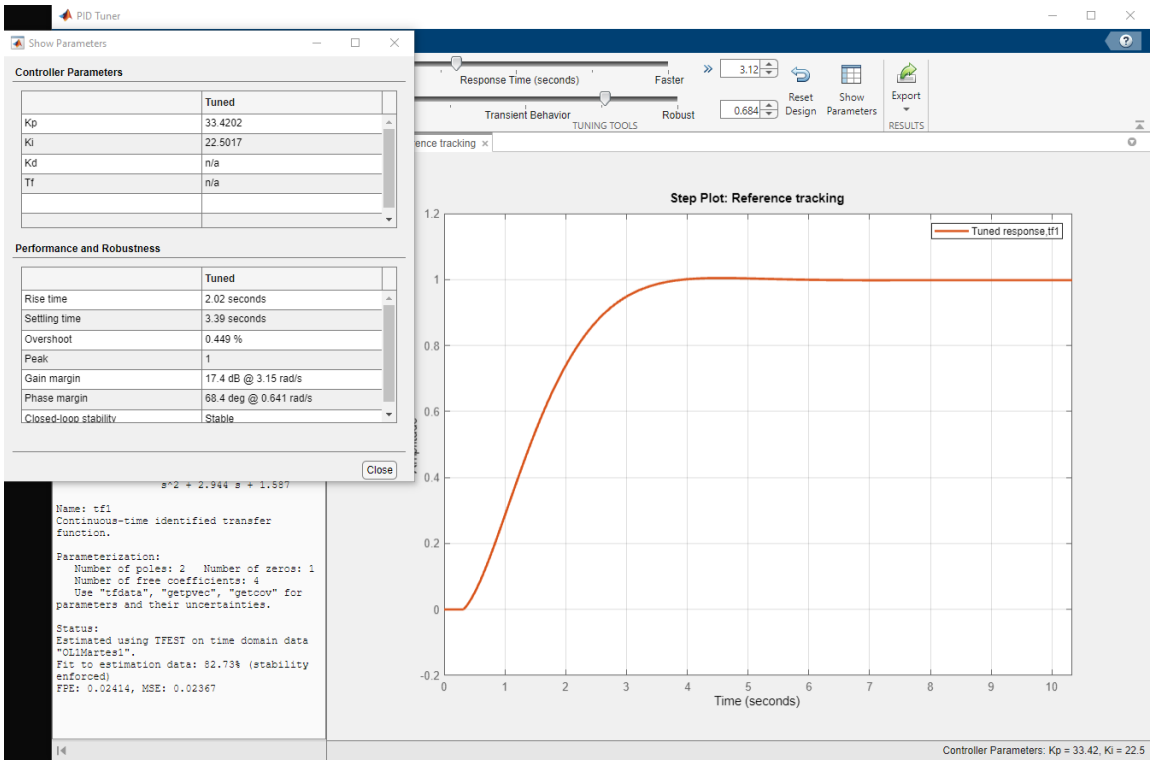


Figura 36 Sintonización del controlador PI para la segunda planta estimada.

Para aplicar las reglas "AMIGO" volvemos a analizar la respuesta del sistema identificado ante una entrada escalón unitario. Utilizando el tiempo muerto aparente identificado y la ganancia estática y constante de tiempo obtenidas de la respuesta al escalón unitario tenemos los siguientes valores:

$$K = 0.0288, \qquad L = 0.32, \qquad T = 1.78$$

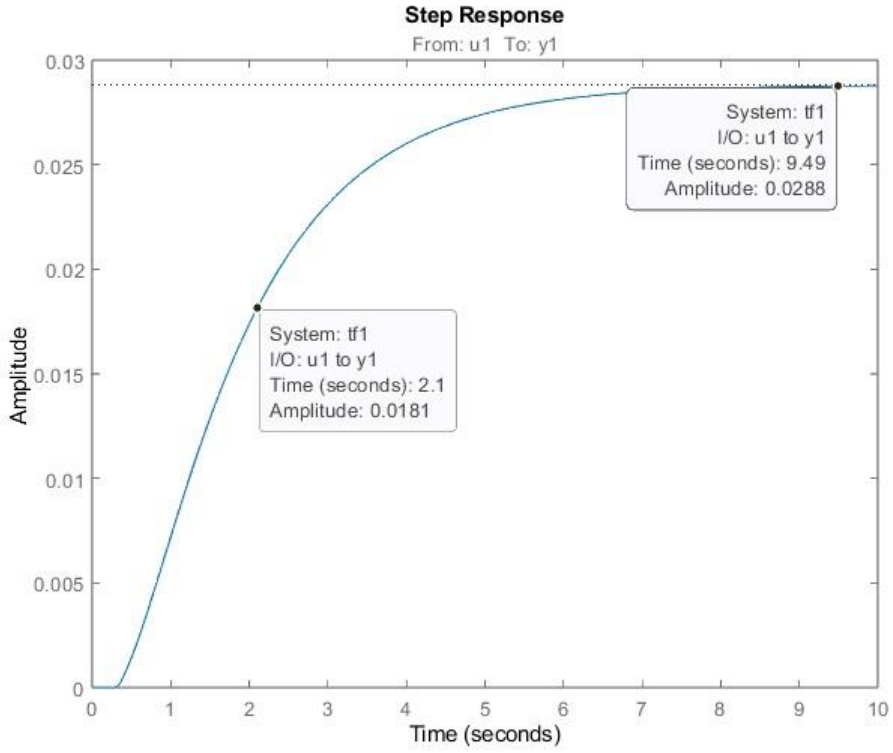


Figura 37. Entrada escalón unitario para el segundo sistema identificado. Estimación de los parámetros que caracterizan la respuesta.

Aplicando las reglas nuevamente obtenemos:

$$Kp = 27, \qquad Ki = 19$$

Que, a diferencia de la primera identificación, son valores bastante aproximados a los obtenidos mediante el PIDTuner. Las reglas "AMIGO" procuran asegurar la estabilidad del sistema para un gran rango de procesos. Al aumentar la relación L/T, denominada controlabilidad, la estabilidad puede verse comprometida al utilizar ganancias de tal magnitud como las sugeridas en la primera sintonización.

Al ser los valores tan similares, las pruebas se realizarán con los siguientes valores:

$$Kp = 30, \qquad Ki = 20$$

5.8 Resultados de la segunda sintonización

Las pruebas realizadas se han llevado a cabo en las mismas condiciones que en el apartado 5.5 Resultados de la primera sintonización. Con el nuevo controlador obtenido y el conocimiento previo del comportamiento del sistema esperamos un comportamiento oscilatorio al tener *Ki = 20*. Este es efectivamente el caso tal y como se muestra en la *Figura 38*. Sin embargo, la respuesta parece converger más rápidamente hacia la consigna, tal y como se esperaba de implementar una ganancia proporcional mayor a *Kp = 12*.

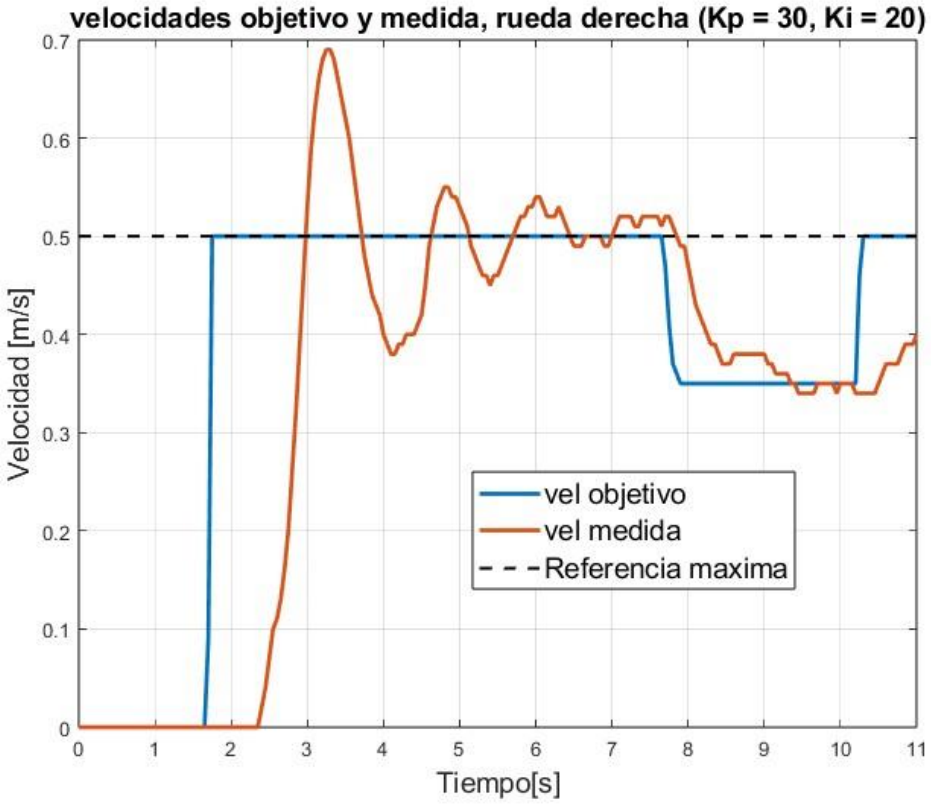


Figura 38. Velocidades objetivo y medida de la rueda derecha obtenidas durante las pruebas reales (Kp =30, Ki = 20)

Para reducir el comportamiento oscilatorio de la respuesta, reducimos la ganancia integral hasta *Ki = 10*. Los resultados se muestran en la *Figura 39*. Observamos que el comportamiento oscilatorio ha sido reducido en gran medida, al menos para velocidades bajas. El intervalo de tiempo comprendido entre los cuatro y seis segundos

aproximadamente es un intento de rotación diferencial pura. En este caso la consigna de velocidad no se alcanza ya que la rotación pura depende, en gran medida, de la orientación de las ruedas guía delanteras lo cual dificulta la labor de nuestro controlador. Un análisis más detallado se realiza en la sección 7.2 Ruedas guía.

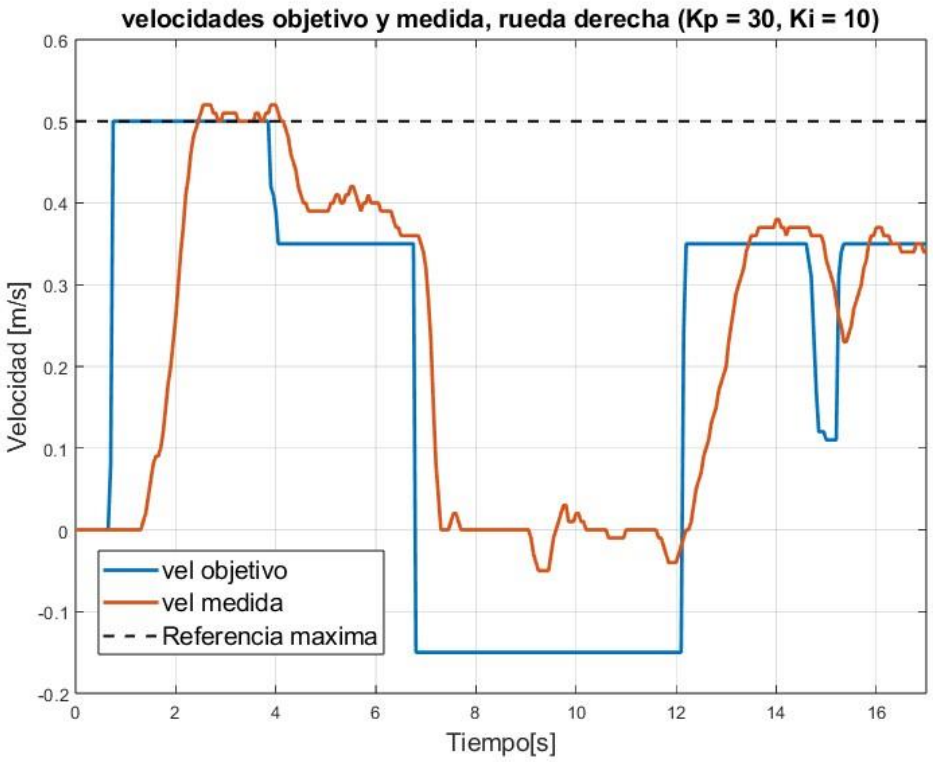


Figura 39. Velocidades objetivo y medida de la rueda derecha obtenidas durante las pruebas reales (Kp =30, Ki = 10), consigna máxima de 0.5 [m/s]

Para velocidades mayores, la acción integral sigue siendo demasiado brusca en algunas situaciones, tal y como podemos observar en la *Figura 40*. Vuelve a haber un sobrepasamiento debido a la acción integral, por lo que reduciremos la ganancia integral manteniendo la ganancia proporcional. Sin embargo, esto puede tener consecuencias indeseables en velocidades más bajas. Por ello se propondrán dos controladores distintos, los cuales se discutirán en el siguiente apartado.

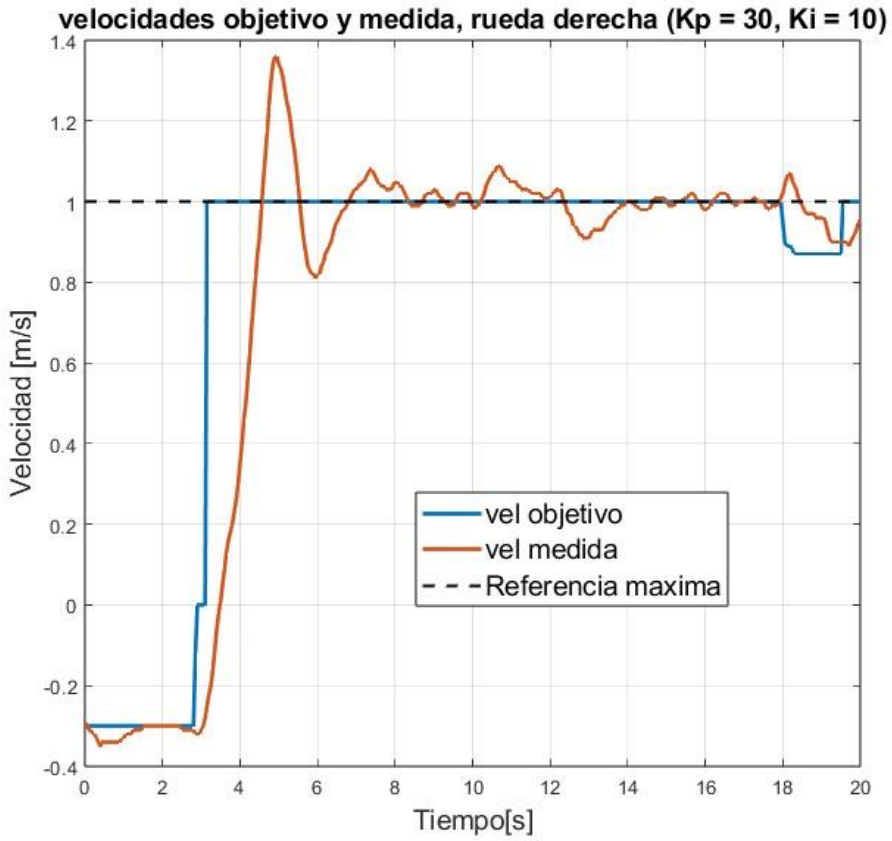


Figura 40. Velocidades objetivo y medida de la rueda derecha obtenidas durante las pruebas reales (Kp = 30, Ki = 10), consigna máxima de 1 [m/s]

5.9. Ajuste final de los valores y propuesta de controlador

Utilizando la implementación del selector de velocidades máximas, incluiremos dentro de esta selección distintos controladores en función de la velocidad máxima seleccionada.

Los controladores propuestos son:

Tabla 3. Controladores propuestos para las distintas velocidades

Velocidad máxima	**Kp**	**Ki**
0.5 [m/s]	50	10
1 [m/s] y superior	30	5

Para el primer controlador se ha optado por incrementar la acción proporcional al limitarlo solo a velocidades inferiores a 0.5 [m/s]. Este valor se ha hallado después de varias pruebas con ganancias superiores a 30 pero inferiores a 70. La conducción a estas velocidades resulta satisfactoria y lo suficientemente controlable como para manejar la silla de ruedas en espacios más reducidos.

En la *Figura 41* podemos ver los resultados de estos dos controladores propuestos. Se evitan en gran medida los sobrepasamientos, que ocurrirán de manera puntual ante cambios de la consigna muy bruscos si se ha acumulado algo de error integral, pero que el anti-windup por lo general es capaz de contener. Se propone el mismo controlador para la velocidad máxima de 1 [m/s] que para las superiores implementadas, ya que por cuestiones de seguridad no se ha hecho un estudio más extensivo de estas otras selecciones de velocidad.

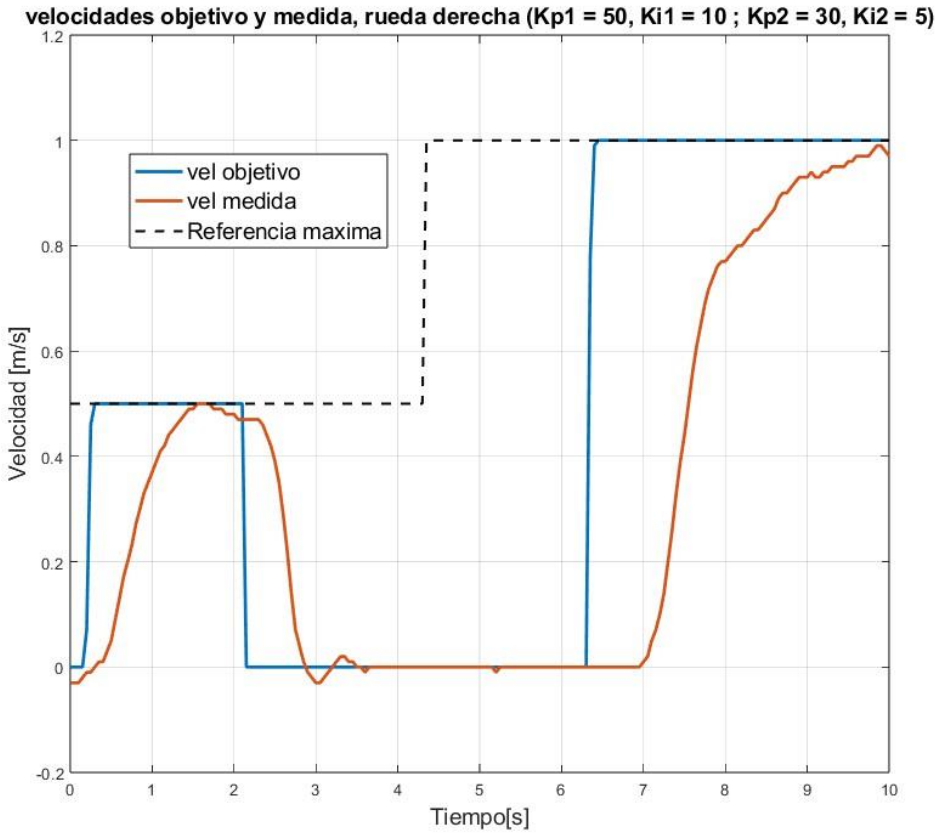


Figura 41. Velocidades objetivo y medida de la rueda derecha obtenidas durante las pruebas reales (Kp1 =50, Ki1 = 10; Kp2 = 30, Ki2 = 5), distintas referencias máximas.

La ganancia Kp1 = 50 en el primer controlador ofrece una respuesta rápida ante cambios en la consigna, que era lo que se buscaba al aumentar la ganancia inicial Kp = 30. Así, como se ha mencionado anteriormente, la silla de ruedas responde con buena precisión incluso en curvas, como se pretende mostrar en la *Figura 42*.

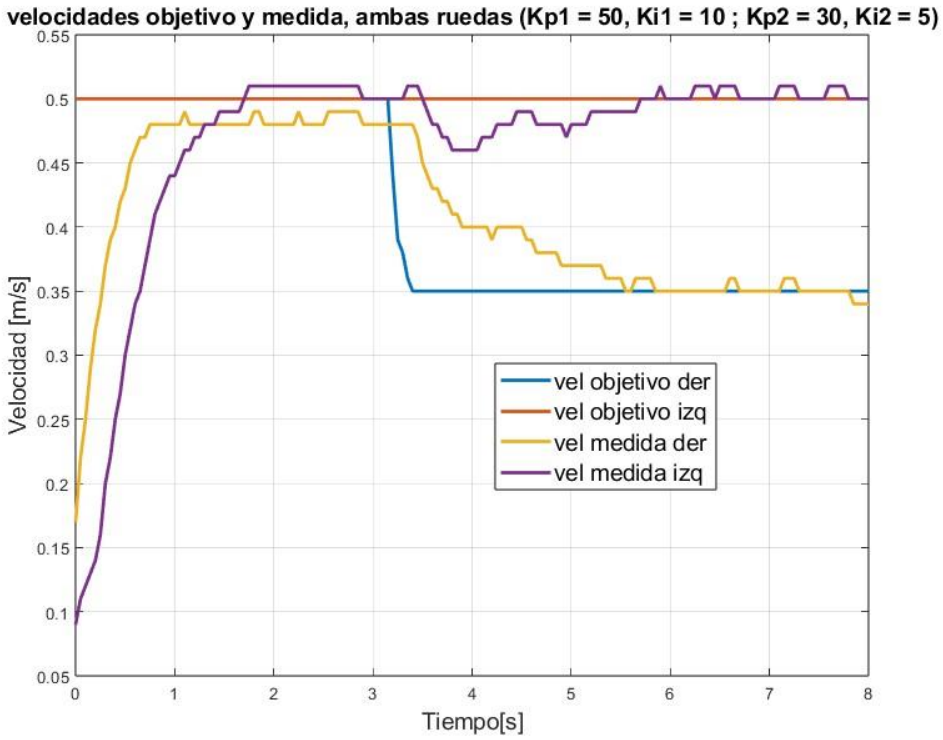


Figura 42. Velocidades objetivo y medida de ambas ruedas durante una curva obtenidas durante las pruebas reales (Kp1 =50, Ki1 = 10; Kp2 = 30, Ki2 = 5), consigna máxima de 0.5 [m/s]

6. Implementación del modo automático

ROS, o *Robot Operating System*, es un kit de desarrollo de código abierto orientado a aplicaciones robóticas, el cual aporta un marco de trabajo con un nivel de abstracción propio de un sistema operativo, pero sin llegar a ser uno realmente. En otras palabras, aporta las herramientas y librerías necesarias para acelerar el desarrollo de las aplicaciones robóticas. [10]

Los nodos son la pieza central del funcionamiento de ROS, permiten procesar la información recibida desde mensajes con información de sensores, actuadores, estados, etc. a través de los tópicos, que son "buses" de datos. Por lo general, los nodos no saben con qué parte del sistema se están comunicando, para ello, los nodos que estén interesados en cierta información deben suscribirse a los tópicos que les sean relevantes. De la misma manera, los nodos publican estos mensajes a través de tópicos para que otros nodos puedan interactuar con la información embebida en éstos.

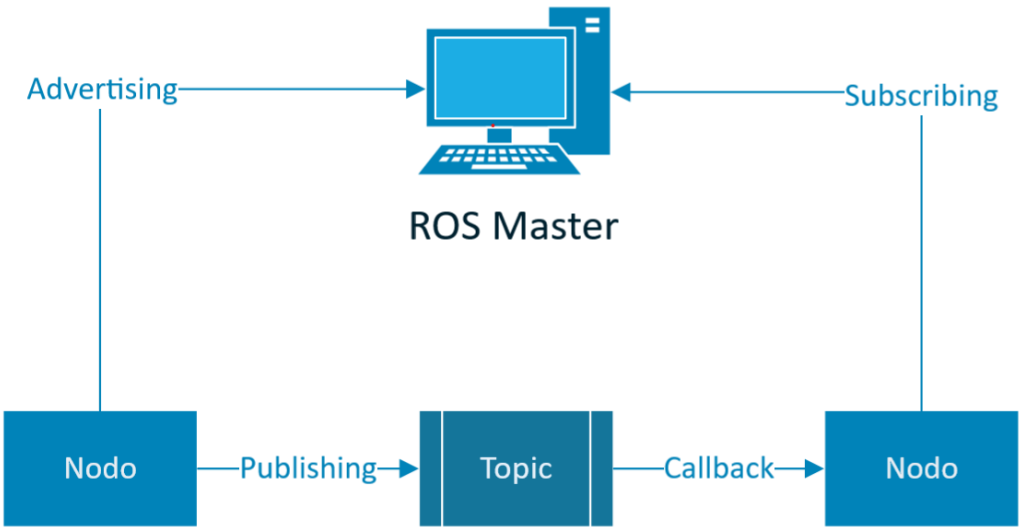


Figura 43. Típico modelo ROS. (Recreación de la ilustración en [11])

Como se muestra en la *Figura 44*, para poder recibir comandos desde un ordenador se ha hecho uso de la función "*CmdVel"* la cual convierte en velocidades

objetivo para cada una de las ruedas los comandos introducidos en el ordenador en forma de velocidad lineal y rotación. Además, se quiere enviar la información de la pose y la velocidad de la silla de ruedas al ordenador para poder monitorizarla y visualizarla en tiempo real. Esto se lleva a cabo mediante la publicación de los mensajes "pose" y "twist", cuya información depende de la adquisición de datos que realizamos a través de los encoders. Tanto el cálculo de la pose como de la velocidad a partir de las cuentas de los encoders se lleva a cabo durante la interrupción principal de muestro cada 20 milisegundos. Así nos aseguramos de que los valores obtenidos son lo más cercanos a la realidad dentro de lo posible con nuestra configuración.

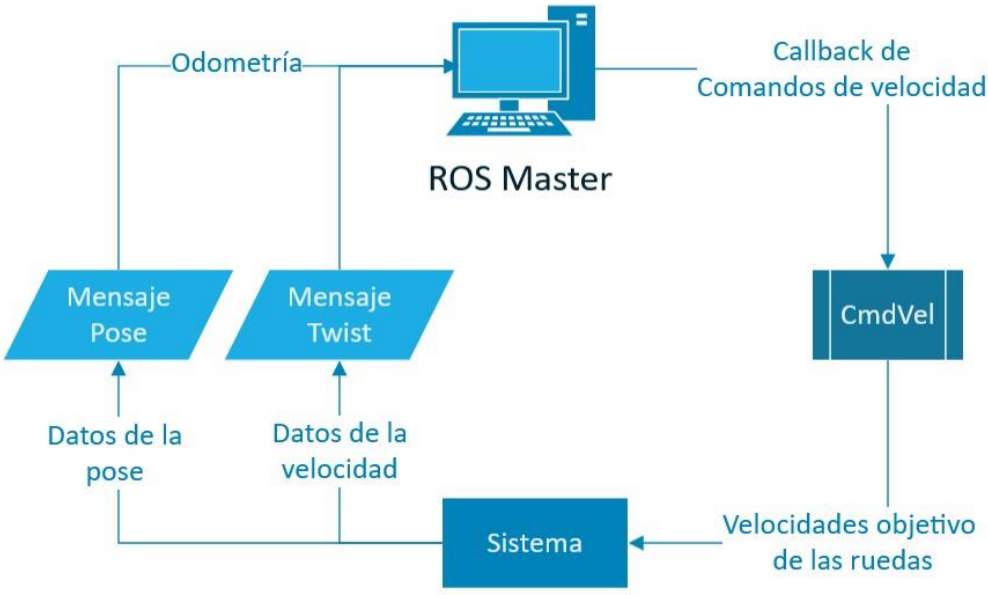


Figura 44. Flujo de datos en el modo automático propuesto.

7. Problemas encontrados

7.1 Frenos

Inicialmente, se pretendía utilizar el modo "*Brakes*" del controlador Sabertooth para las salidas P1 y P2, de modo que la acción de los frenos proviniera del propio Sabertooth. El software *Describe* incluye la personalización de este modo de funcionamiento, incluyendo el tiempo que tarda el controlador en soltar los frenos tras recibir un comando y el tiempo que tarda en volver a ponerlos tras no recibirlos. Por motivos que no han podido ser determinados, el controlador Sabertooth, ante ciertas situaciones, no era capaz de desbloquear los frenos aun cuando estaba recibiendo comandos. Entre estas situaciones se encuentran:

- Al recibir comandos de rotación pura.
- Cuando el cambio en la velocidad era muy repentino. Por ejemplo: enviando la máxima consigna de velocidad desde el reposo.
- Arbitrariamente durante las pruebas de los frenos en situaciones que no pudieron ser identificadas con claridad.

Al realizar las primeras pruebas en vacío, se pensó que este comportamiento tan particular podía deberse al freno regenerativo que viene implementado en el controlador. Para determinar si el problema era del freno regenerativo o de los propios frenos, se instalaron unos LED que indican si el freno está alimentado a 24V (quitado) o si por el contrario no había sido retirado. Cuando ocurrían este tipo de comportamientos los LED indicaban que el freno no había sido retirado completamente, por lo que se buscaron otras posibles causas.

Si los motores estaban acaparando toda la potencia suministrada, era posible que una caída de tensión, aunque pequeña, pudiera ocasionar este comportamiento errático del transistor (normalmente de tecnología FET) que controla los frenos. Cualquier desviación considerable de los 24V necesarios para conectar el freno podría ser el motivo de que no estuvieran siendo retirados completamente. Para intentar solucionarlo se conectó un condensador capaz de mantener este voltaje de manera estable, de forma que una caída de tensión puntual no ocasionase un cambio en el voltaje suministrado al freno.

Aunque se observó una mejoría en algunas de las situaciones descritas con anterioridad, el problema persistía.

Temiendo que el problema estuviera en el transistor que cierra el circuito de los frenos, se probó a controlarlos directamente con un relé. Dicho relé respondería a la señal enviada por las salidas de control de los frenos P1 y P2, y cerraría el circuito de la misma manera que lo haría inicialmente la electrónica original. Las pruebas realizadas con este nuevo circuito implementado y el condensador instalado con anterioridad resultaron insatisfactorias ya que el comportamiento errático de los frenos seguía presente.

Al no poder identificar la causa del problema, se ha optado por controlar los frenos desde el propio Arduino sin hacer uso de las salidas específicas del Sabertooth. Para ello se ha desarrollado código que emulase el comportamiento ideal del Sabertooth, y se ha utilizado un relé dependiente de la señal emitida por el Arduino. Para aislar completamente los frenos de los 24V en caso de que no estuvieran siendo utilizados, el relé al estar abierto conecta ambos frenos directamente a tierra. En caso contrario los frenos son alimentados.

Mediante esta estrategia, e implementando el código apropiado para enviar la señal de control al relé, los problemas relacionados con el freno fueron solucionados satisfactoriamente.

7.2 Ruedas guía y rotación pura

En la *Figura 45* vemos que la silla con el controlador con ganancias Kp = 30, y Ki = 10 es capaz de comenzar a rotar, pero no sin pasar antes por un largo intervalo de tiempo en el que la acción integral llega a saturar el controlador, pero sin garantizar la rotación en algunas situaciones. El mismo comportamiento se repite para los dos controladores propuestos en 5.9. Ajuste final de los valores y propuesta de controlador. Las oscilaciones iniciales de la velocidad medida se deben en gran parte al método anti-windup implementado: cada vez que la velocidad alcanza cierto valor la acción integral es eliminada, dificultando aún más la rotación. Para procurar solventar este problema se ha procurado ajustar los valores de los umbrales del anti-windup. Sin embargo, no se ha conseguido cambiar este comportamiento de manera significativa.

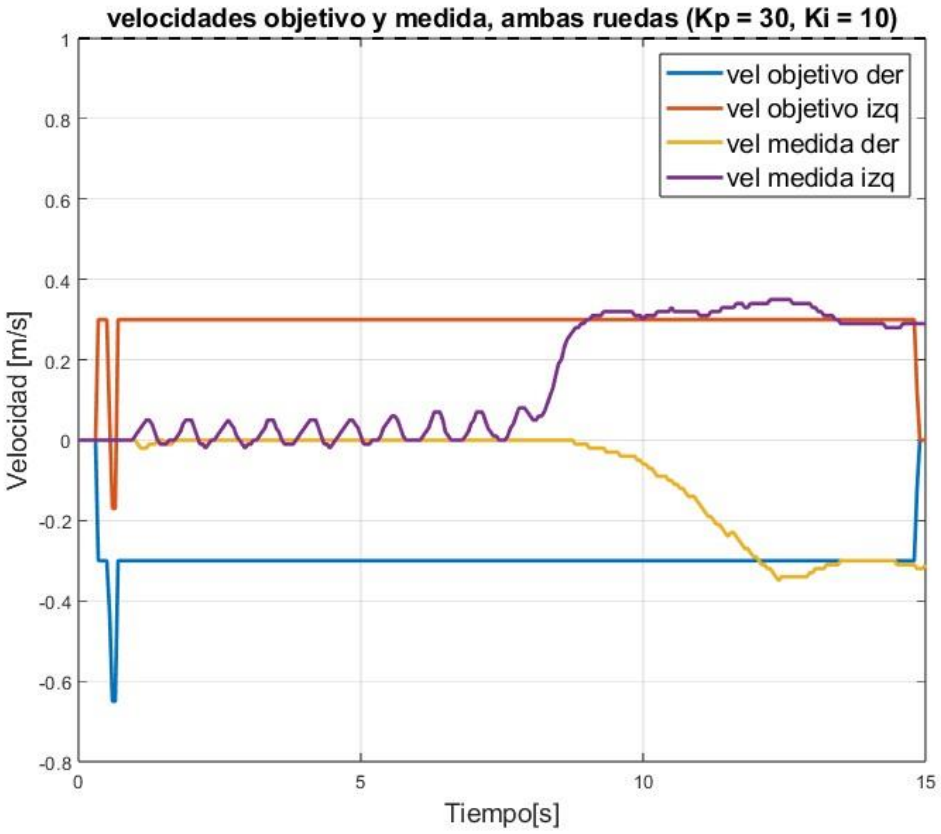


Figura 45. Velocidades objetivo y medida de ambas ruedas para una rotación pura (Kp = 30, Ki = 10)

Se ha observado que tras una reorientación de las ruedas guías después de una breve traslación junto con algo de rotación puede facilitar la rotación de la silla, como es previsible. En la *Figura 46* tenemos un ejemplo de este comportamiento: una consigna de "curva en marcha atrás" permite reorientar las ruedas lo necesario para luego comenzar la rotación en un breve periodo de tiempo. En la *Figura 45* ocurre algo similar, pero las ruedas guía no fueron lo suficientemente reorientadas como para permitir una rotación inmediata.

Por ello se ha propuesto una modificación al algoritmo de mapeo del joystick, que detecta si el usuario pretende rotar en el sitio y entonces realiza automáticamente la reorientación de las ruedas.

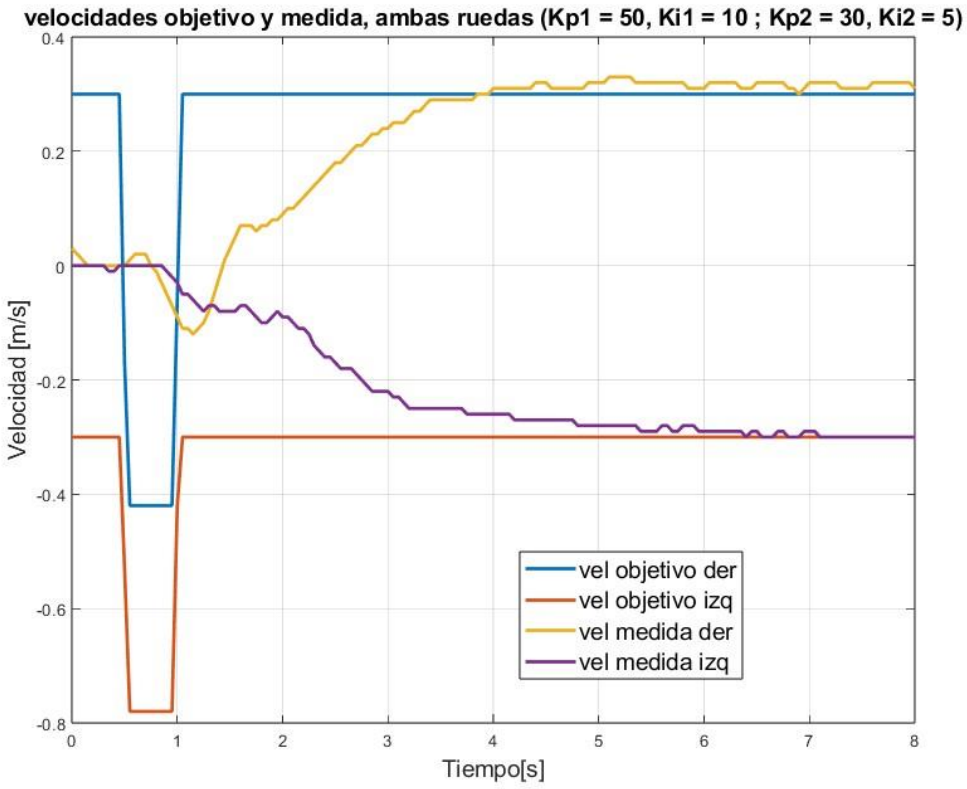


Figura 46. Velocidades objetivo y medida de ambas ruedas para una rotación pura tras una reorientación de las ruedas guía delanteras. (Kp1 = 50, Ki1 = 10; Kp2 = 30, Ki2 = 5).

Idealmente, con este algoritmo la silla se comportaría como en los resultados expuestos en la *Figura 47*. Sin embargo, no ha sido posible que la silla se comportase de esta manera para todos los escenarios posibles, por lo que el algoritmo no resulta todo lo fiable que debería ser para poder realizar rotaciones puras desde cualquier orientación de las ruedas guía delanteras.

La solución más fiable que se ha encontrado es una nueva modificación que convierte la rotación pura en una rotación pivotada sobre una de las ruedas, lo que introduce un radio de giro bastante indeseable en el control automático. En este modo de control por ordenador normalmente se asume que se está controlando un robot con la habilidad de girar sobre sí mismo, por lo que este tipo de rotaciones pivotadas introducirían un error considerable en los algoritmos de control de posición, por ejemplo.

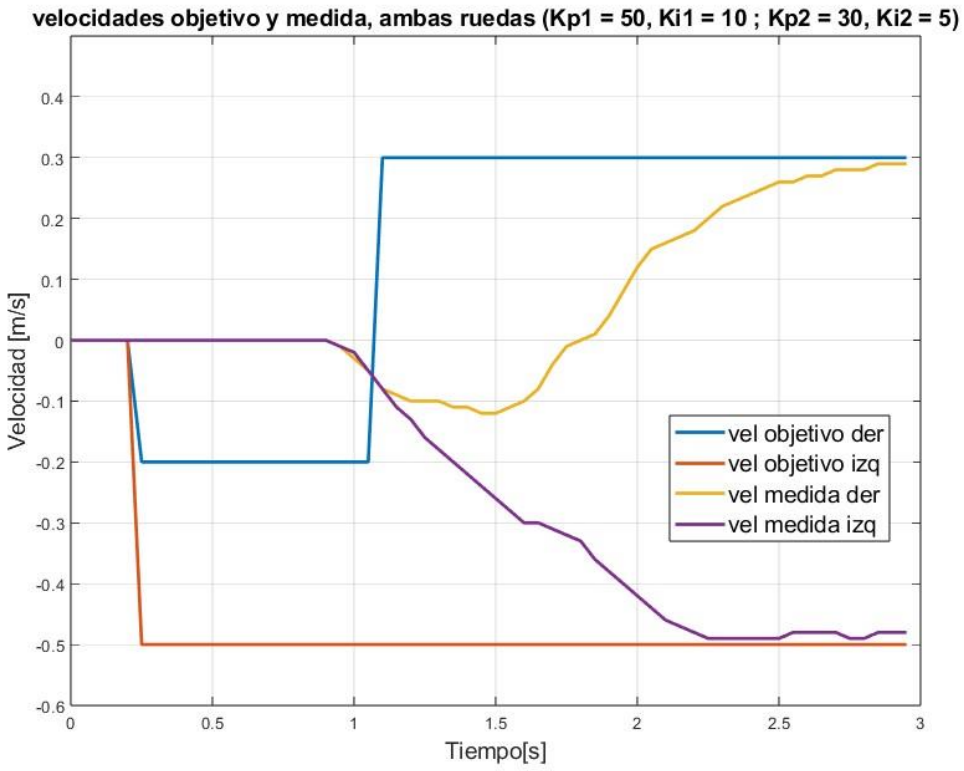


Figura 47. Velocidades objetivo y medida de ambas ruedas para una rotación pura mediante el algoritmo propuesto para orientar las ruedas delanteras (Kp1 = 50, Ki1 = 10; Kp2 = 30, Ki2 = 5).

8. Conclusiones y líneas futuras

El controlador PI implementado es capaz de realizar el control de velocidad correctamente y de manera segura, evitando sobrepasamientos y limitando comportamientos erráticos mediante los algoritmos *anti-windup* implementados. Existe margen de mejora para los controladores obtenidos, sobre todo en velocidades superiores a las estudiadas, en las que el control resulta más complicado. Para las velocidades con las que se han realizado la mayoría de las pruebas, la silla responde adecuadamente, permitiendo a un usuario su uso incluso en interiores.

Las identificaciones de la planta del sistema realizadas pueden verse ampliamente mejoradas, y, por lo tanto, los valores de sintonización obtenidos pueden ajustarse más a la realidad. Existen experimentos de lazo abierto más complejos que los llevados a cabo en este proyecto, y las herramientas de identificación utilizadas permiten aproximaciones más acertadas del sistema, incluyendo estructuras de modelo no lineales que pueden representar mejor este tipo de sistemas.

El uso de las reglas AMIGO y las herramientas de sintonización de controladores de MATLAB garantizan en gran medida la estabilidad de los controladores propuestos, y los resultados así parecen corroborarlo. Sin embargo, un estudio más avanzado de la estabilidad sería necesario para garantizar la seguridad del usuario, ya que los valores finales se han ajustado mediante prueba y error.

Para no depender tanto del algoritmo de *anti-windup* propuesto, añadir la acción derivativa al controlador, completándolo como PID, puede ofrecer buenos resultados. Con la inclusión de la acción derivativa se podría observar una mejora de los márgenes de estabilidad del sistema y disminuiría los cambios bruscos en la velocidad debidos a la acción integral.

Los problemas expuestos en 7.2 Ruedas guía y rotación pura impiden una correcta implementación del modo automático, ya que el comportamiento de la silla en algunos escenarios no resulta adecuado para su control mediante el ordenador, dificultando una futura implementación de algoritmos de navegación automática. Un nuevo enfoque, o una mejora del algoritmo propuesto podrían subsanar considerablemente estos problemas, así como un nuevo ajuste de los algoritmos *anti-*

windup que dificultan la rotación, pero que resultan imprescindibles para controlar la acción integral en el resto de las situaciones.

Tal y como se propone en [12], la implementación de un algoritmo de MPC[5] que priorice la seguridad del usuario, estableciendo límites en las aceleraciones y velocidades máximas, es una vertiente sumamente atractiva para este tipo de control orientado a la seguridad, especialmente cuando se tienen en cuenta las necesidades de este tipo de usuario. Este artículo, además, concluye que el uso de controladores PI para este tipo de control puede resultar subóptimo a la hora de limitar las mencionadas velocidades y aceleraciones máximas.

9. Conclusions and future prospects

The PI controller implemented can perform speed control correctly and safely, avoiding overshoots and limiting erratic behaviour by means of the *anti-windup* algorithms implemented. There is room for improvement for the controllers obtained, especially at speeds higher than those studied, where control is more complicated. For the speeds at which most of the tests were carried out, the chair responds adequately, allowing a user to use it even indoors.

The system floor plan identifications made can be greatly improved, and therefore the tuning values obtained can be more closely matched to reality. There are more complex open-loop experiments than those carried out in this project, and the identification tools used allow more accurate approximations of the system, including non-linear model structures that can better represent such systems.

The use of AMIGO rules and MATLAB's controller tuning tools largely guarantee the stability of the proposed controllers, and the results seem to corroborate this. However, a more advanced study of the stability would be necessary to ensure user safety, as the final values have been adjusted by trial and error.

[5] Model Predictive Control

In order not to rely so much on the proposed anti-windup algorithm, adding the derivative action to the controller, making it complete as PID, may offer good results. With the inclusion of the derivative action, an improvement of the stability margins of the system could be observed and would decrease the abrupt changes in speed due to the integral action.

The problems outlined in 7.2 Guiding wheels and pure rotation prevent a correct implementation of the automatic mode, as the behaviour of the chair in some scenarios is not suitable for computer control, making future implementation of automatic navigation algorithms difficult. A new approach, or an improvement of the proposed algorithm could considerably overcome these problems, as well as a new adjustment of the anti-windup algorithms that hinder the rotation, but which are essential to control the integral action in the rest of the situations.

As proposed in [12], the implementation of an MPC algorithm that prioritizes user safety by setting limits on accelerations and maximum speeds is an extremely attractive approach to this type of safety-oriented control, especially when the needs of this type of user are considered. This article also concludes that the use of PI controllers for this type of control may be suboptimal in limiting the maximum speeds and accelerations.

10. Presupuesto

Componentes

	Unidades	Coste unitario	Coste Total
Silla de ruedas Vermeiren Forest 3	1	5.368,00 €	5.368,00 €
Arduino MEGA 2560 rev 3	1	33,60 €	33,60 €
Sabertooth 2x32	1	112,38 €	112,38 €
Encoder incremental HEDS - 9140	2	37,78 €	75,56 €
Joystick analógico	1	2,02 €	2,02 €
Interruptor KN3(c) -202	1	1,70 €	1,70 €
Selector rotatorio	1	2,83 €	2,83 €
Componentes electrónicos varios	-	20,00 €	20,00 €
Cableado	-	20,00 €	20,00 €
Subtotal			**5.636,09 €**

Mano de obra

	Horas totales	Coste unitario	Coste Total
Ingeniero Junior	180	15,00 €	2.700,00 €
Subtotal			**2.700,00 €**

Coste total

	Coste Total
Subtotal Componentes	5.636,09 €
Subtotal Mano de Obra	2.700,00 €
Total	**8.336,09 €**

Referencias

[1] *Sabertooth dual 32A motor driver*. Available: https://www.dimensionengineering.com/products/sabertooth2x32.

[2] *Arduino*. Available: https://www.arduino.cc/.

[3] B. Joshi, R. Shrestha and R. Chaudhary, *Modeling, Simulation and Implementation of Brushed DC Motor Speed Control using Optical Incremental Encoder Feedback.* 2014.

[4] L. Fehr, W. E. Langbein and S. B. Skaar, "Adequacy of power wheelchair control interfaces for persons with severe disabilities: A clinical survey," *J. Rehabil. Res. Dev.*, vol. 37, *(3)*, pp. 353-360, 2000.

[5] *Librería Encoder de Paul Stroffegen*. Available: https://github.com/PaulStoffregen/Encoder.

[6] *Processing Grapher*. Available: https://wired.chillibasket.com/processing-grapher/.

[7] B. M. Faria *et al*, "Intelligent wheelchair manual control methods: A usability study by cerebral palsy patients," in . DOI: 10.1007/978-3-642-40669-0_24.

[8] G. Franklin, D. Powell and A. Emami-Naeini, *Feedback Control of Dynamic Systems, Global Edition.* 2019Available: https://ebookcentral.proquest.com/lib/[SITE_ID]/detail.action?docID=5770170.

[9] T. Hagglund and K. J. Astrom, "Revisiting The Ziegler-Nichols Tuning Rules For Pi Control," *Asian Journal of Control*, vol. 4, *(4)*, pp. 364-380, 2002. . DOI: 10.1111/j.1934-6093.2002.tb00076.x.

[10] *ROS: Home*. Available: https://www.ros.org/.

[11] *ROS 101: An Intro to the Robot Operating System*. Available: https://www.designnews.com/gadget-freak/ros-101-intro-robot-operating-system.

[12] M. Yuan *et al*, "Safety-based Speed Control of a Wheelchair using Robust Adaptive Model Predictive Control," 2022. . DOI: 10.48550/arxiv.2210.02692.

ANEXOS

Universidad de La Laguna

ANEXO I: CÓDIGO

VSCODE_PRINT_SCRIPT_TAGS

Folder

4 printable files

PID.h
Sabertooth.cpp
Sabertooth.h
SillaVermeiren-TFG.ino

PID.h

```
/* PID */

class PID {

  public:

  // Variables
  double kp, ki, output;
  double error_P, error_I;

  // Constructor
  PID(double x, double y) {
    kp = x;
    ki = y;
  }

};
```

Sabertooth.cpp

```
/*
Arduino Library for SyRen/Sabertooth Packet Serial
Copyright (c) 2012-2013 Dimension Engineering LLC
http://www.dimensionengineering.com/arduino

Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
ANY
```

```
13  SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
    WHATSOEVER
14  RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF
    CONTRACT,
15  NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
    WITH THE
16  USE OR PERFORMANCE OF THIS SOFTWARE.
17  */
18
19  #include "Sabertooth.h"
20
21  Sabertooth::Sabertooth(byte address)
22    : _address(address), _port(SabertoothTXPinSerial)
23  {
24
25  }
26
27  Sabertooth::Sabertooth(byte address, SabertoothStream& port)
28    : _address(address), _port(port)
29  {
30
31  }
32
33  void Sabertooth::autobaud(boolean dontWait) const
34  {
35    autobaud(port(), dontWait);
36  }
37
38  void Sabertooth::autobaud(SabertoothStream& port, boolean dontWait)
39  {
40    if (!dontWait) { delay(1500); }
41    port.write(0xAA);
42  #if defined(ARDUINO) && ARDUINO >= 100
43    port.flush();
44  #endif
45    if (!dontWait) { delay(500); }
46  }
47
48  void Sabertooth::command(byte command, byte value) const
49  {
50    port().write(address());
51    port().write(command);
52    port().write(value);
53    port().write((address() + command + value) & B01111111);
54  }
55
56  void Sabertooth::throttleCommand(byte command, int power) const
57  {
58    power = constrain(power, -126, 126);
59    this->command(command, (byte)abs(power));
60  }
```

```
61
62  void Sabertooth::motor(int power) const
63  {
64    motor(1, power);
65  }
66
67  void Sabertooth::motor(byte motor, int power) const
68  {
69    if (motor < 1 || motor > 2) { return; }
70    throttleCommand((motor == 2 ? 4 : 0) + (power < 0 ? 1 : 0), power);
71  }
72
73  void Sabertooth::drive(int power) const
74  {
75    throttleCommand(power < 0 ? 9 : 8, power);
76  }
77
78  void Sabertooth::turn(int power) const
79  {
80    throttleCommand(power < 0 ? 11 : 10, power);
81  }
82
83  void Sabertooth::stop() const
84  {
85    motor(1, 0);
86    motor(2, 0);
87  }
88
89  void Sabertooth::setMinVoltage(byte value) const
90  {
91    command(2, (byte)min(value, 120));
92  }
93
94  void Sabertooth::setMaxVoltage(byte value) const
95  {
96    command(3, (byte)min(value, 127));
97  }
98
99  void Sabertooth::setBaudRate(long baudRate) const
100 {
101 #if defined(ARDUINO) && ARDUINO >= 100
102   port().flush();
103 #endif
104
105   byte value;
106   switch (baudRate)
107   {
108   case 2400:           value = 1; break;
109   case 9600: default: value = 2; break;
110   case 19200:          value = 3; break;
111   case 38400:          value = 4; break;
```

```
  case 115200:         value = 5; break;
  }
  command(15, value);

#if defined(ARDUINO) && ARDUINO >= 100
  port().flush();
#endif

  // (1) flush() does not seem to wait until transmission is complete.
  //     As a result, a Serial.end() directly after this appears to
  //     not always transmit completely. So, we manually add a delay.
  // (2) Sabertooth takes about 200 ms after setting the baud rate to
  //     respond to commands again (it restarts).
  // So, this 500 ms delay should deal with this.
  delay(500);
}

void Sabertooth::setDeadband(byte value) const
{
  command(17, (byte)min(value, 127));
}

void Sabertooth::setRamping(byte value) const
{
  command(16, (byte)constrain(value, 0, 80));
}

void Sabertooth::setTimeout(int milliseconds) const
{
  command(14, (byte)((constrain(milliseconds, 0, 12700) + 99) / 100));
}

```

Sabertooth.h

```
/*
Arduino Library for SyRen/Sabertooth Packet Serial
Copyright (c) 2012-2013 Dimension Engineering LLC
http://www.dimensionengineering.com/arduino

Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
ANY
SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER
```

```
14  RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF
    CONTRACT,
15  NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
    WITH THE
16  USE OR PERFORMANCE OF THIS SOFTWARE.
17  */
18
19  #ifndef Sabertooth_h
20  #define Sabertooth_h
21
22  #if defined(ARDUINO) && ARDUINO >= 100
23  #include <Arduino.h>
24  typedef Stream SabertoothStream;
25  #else
26  #include <WProgram.h>
27  #include <Arduino.h>
28  typedef Print SabertoothStream;
29  #endif
30
31  #if defined(USBCON)
32  #define SabertoothTXPinSerial Serial1 // Arduino Leonardo has TX->1 on
    Serial1, not Serial.
33  #else
34  #define SabertoothTXPinSerial Serial
35  #endif
36  #define SyRenTXPinSerial SabertoothTXPinSerial
37
38  /*!
39  \class Sabertooth
40  \brief Controls a %Sabertooth or %SyRen motor driver running in Packet
    Serial mode.
41  */
42  class Sabertooth
43  {
44  public:
45    /*!
46    Initializes a new instance of the Sabertooth class.
47    The driver address is set to the value given, and the Arduino TX
    serial port is used.
48    \param address The driver address.
49    */
50    Sabertooth(byte address);
51
52    /*!
53    Initializes a new instance of the Sabertooth class.
54    The driver address is set to the value given, and the specified serial
    port is used.
55    \param address The driver address.
56    \param port    The port to use.
57    */
58    Sabertooth(byte address, SabertoothStream& port);
59
```

```
60  public:
61    /*!
62    Gets the driver address.
63    \return The driver address.
64    */
65    inline byte address() const { return _address; }
66
67    /*!
68    Gets the serial port.
69    \return The serial port.
70    */
71    inline SabertoothStream& port() const { return _port; }
72
73    /*!
74    Sends the autobaud character.
75    \param dontWait If false, a delay is added to give the driver time to
    start up.
76    */
77    void autobaud(boolean dontWait = false) const;
78
79    /*!
80    Sends the autobaud character.
81    \param port     The port to use.
82    \param dontWait If false, a delay is added to give the driver time to
    start up.
83    */
84    static void autobaud(SabertoothStream& port, boolean dontWait =
    false);
85
86    /*!
87    Sends a packet serial command to the motor driver.
88    \param command The number of the command.
89    \param value   The command's value.
90    */
91    void command(byte command, byte value) const;
92
93  public:
94    /*!
95    Sets the power of motor 1.
96    \param power The power, between -127 and 127.
97    */
98    void motor(int power) const;
99
100   /*!
101   Sets the power of the specified motor.
102   \param motor The motor number, 1 or 2.
103   \param power The power, between -127 and 127.
104   */
105   void motor(byte motor, int power) const;
106
107   /*!
```

```
108   Sets the driving power.
109   \param power The power, between -127 and 127.
110   */
111   void drive(int power) const;
112
113   /*!
114   Sets the turning power.
115   \param power The power, between -127 and 127.
116   */
117   void turn(int power) const;
118
119   /*!
120   Stops.
121   */
122   void stop() const;
123
124 public:
125   /*!
126   Sets the minimum voltage.
127   \param value The voltage. The units of this value are driver-specific
      and are specified in the Packet Serial chapter of the driver's user
      manual.
128   */
129   void setMinVoltage(byte value) const;
130
131   /*!
132   Sets the maximum voltage.
133   Maximum voltage is stored in EEPROM, so changes persist between power
      cycles.
134   \param value The voltage. The units of this value are driver-specific
      and are specified in the Packet Serial chapter of the driver's user
      manual.
135   */
136   void setMaxVoltage(byte value) const;
137
138   /*!
139   Sets the baud rate.
140   Baud rate is stored in EEPROM, so changes persist between power
      cycles.
141   \param baudRate The baud rate. This can be 2400, 9600, 19200, 38400,
      or on some drivers 115200.
142   */
143   void setBaudRate(long baudRate) const;
144
145   /*!
146   Sets the deadband.
147   Deadband is stored in EEPROM, so changes persist between power cycles.
148   \param value The deadband value.
149                 Motor powers in the range [-deadband, deadband] will be
      considered in the deadband, and will
150                 not prevent the driver from entering nor cause the driver
      to leave an idle brake state.
151                 0 resets to the default, which is 3.
```

```
152   */
153   void setDeadband(byte value) const;
154
155   /*!
156   Sets the ramping.
157   Ramping is stored in EEPROM, so changes persist between power cycles.
158   \param value The ramping value. Consult the user manual for possible
    values.
159   */
160   void setRamping(byte value) const;
161
162   /*!
163   Sets the serial timeout.
164   \param milliseconds The maximum time in milliseconds between packets.
    If this time is exceeded,
165                       the driver will stop the motors. This value is
    rounded up to the nearest 100 milliseconds.
166                       This library assumes the command value is in units
    of 100 milliseconds. This is true for
167                       most drivers, but not all. Check the packet serial
    chapter of the driver's user manual
168                       to make sure.
169   */
170   void setTimeout(int milliseconds) const;
171
172 private:
173   void throttleCommand(byte command, int power) const;
174
175 private:
176   const byte        _address;
177   SabertoothStream& _port;
178 };
179
180 #endif
181
```

SillaVermeiren-TFG.ino

```
1  /* Silla Vermeiren */
2
3  // Autor: Carlos Menéndez Perdomo
4  // Fecha: 13/07/2023
5
6  // 1: izq, 2: der; desde el punto de vista del usuario
7
8  // LIBRERIAS
9  #include "Sabertooth.h"
10 #include "PID.h"
11 #include <Encoder.h>
12 #include <TimerOne.h>
```

```
13
14  // LIBRERIAS ROS
15  #include <ros.h>
16  #include <geometry_msgs/Twist.h>
17  #include <geometry_msgs/PoseStamped.h>
18  #include <geometry_msgs/TwistStamped.h>
19  #include <geometry_msgs/Quaternion.h>
20
21
22  // OBJETOS
23
24  // Declaracion objeto Sabertooth
25  // Address para packetized serial mode
26  /* DIP wizard para referencia:
27  http://www.dimensionengineering.com/datasheets/SabertoothDIPWizard
    /start.htm */
28  Sabertooth Silla(128, Serial1);
29
30  // Declaracion objeto Encoder
31  // Usar pines de interrupcion de Arduino Mega
32  // Interrupt pins Mega: 2, 3, 18, 19, 20, 21
33  Encoder Encoder1(2, 3);
34  Encoder Encoder2(20, 21);
35
36
37  // OBJETOS ROS
38
39  ros::NodeHandle nh; /* Create the ROS node handle */
40  geometry_msgs::Twist msg;
41
42  void cmdVelCb(const geometry_msgs::Twist &msg);  // callback para el
    topic /cmd_vel
43
44  // Messages
45  geometry_msgs::PoseStamped pose_msg;  // mensaje para publicar pose
46  geometry_msgs::Twist twist_msg;       // mensaje para publicar velocidad
47  //geometry_msgs::TwistStamped twistStam_msg;
48  //geometry_msgs::Twist twist_msg;
49
50
51  // Publishers
52  ros::Publisher posePub("/pose", &pose_msg);
53  ros::Publisher twistPub("/twist", &twist_msg);
54  //ros::Publisher twistStampPub("/twistStamped", &twistStam_msg);
55
56
57  // Subscribers
58  ros::Subscriber<geometry_msgs::Twist> cmdVelSub("/cmd_vel", &cmdVelCb);
59
60
61  // Declaracion objeto PID
```

```
62  // PID (kp, ki)
63  PID PID1(0.0, 0.0);
64  PID PID2(0.0, 0.0);
65
66  // MATRIZ PID CON LAS GANANCIAS DE LOS CONTROLADORES PROPUESTOS
67  double gains[][2] = {{ 50.0, 10.0 }, { 30.0, 5.0 }};
68
69  // JOYSTICK
70  #define x_axis A0
71  #define y_axis A1
72
73  // FRENOS
74  #define frenos 10
75
76  // SELECTOR Y AUTOMATICO
77  #define Sel1 A8
78  #define Sel2 A9
79  #define pinAUTO A3
80  int leds[] = { A7, A6, A5, A4 };
81
82
83  // VARIABLES GLOBALES
84
85  // VARIABLES PARA MUESTREO Y PID
86  double c1, c2, w1, w2, v1, v2;  // c = cuentas, w = velocidad angular
    [rad/s], v = velocidad lineal [m/s]
87
88  double ts = 20.0;  // sample time [ms]
89
90  int i = 0;  // indice para arrays
91  const int window_size = 10; // tamaño ventana
92  const double window_interval = (ts / 1000.0) * (double)window_size;  //
    intervalo ventana [s]
93  double window1[window_size]; // arrays para guardar lecturas encoder
94  double window2[window_size];
95
96  int MAXoutput = 127;  // maximo output del controlador
97  double MAXref = 0.0;  // maxima referencia de velocidad [m/s] (maximo
    ~4) 14 km/h
98
99  double v_target1 = 0.0;  // velocidad objetivo [m/s]
100 double v_target2 = 0.0;
101
102 double v_target1temp = 0.0;
103 double v_target2temp = 0.0;
104
105 // VARIABLES PARA JOYSTICK
106
107 double x_mapped = 0.0;
108 double y_mapped = 0.0;
109 unsigned long UltimoJoystick = 0;
```

```
110
111 // ANTIWINDUP
112
113 int threshold = 0;  // umbral de cuentas para considerar que nos movemos
114
115 // VARIABLES PARA FRENOS
116
117 unsigned long TIEMPOPONFRENO = 1000;
118 unsigned long TIEMPOQUITAFRENOS = 500;
119 unsigned long tiempoComandoNoCero = 0;
120 unsigned long tiempoComandoFreno = 0;
121
122 bool frenoDesactivado = false;
123 bool cambiandoEstadoFrenos = false;
124
125 // VARIABLES PARA SELECTOR Y AUTOMATICO
126 byte seleccion = 0;
127 bool selERROR = false;
128 bool AUTO = false;
129
130 // VARIABLES PARA ODOMETRIA
131 double wheelTrack = 0.5;
132 unsigned long ultimoMensaje = 0;
133
134 double dt, dleft, dright, dx, dy, dxy_ave, dth, vxy, vth, vx, vy, th, x,
    y;
135 unsigned long lastMotorCommand = 0;
136
137 // VARIABLES PARA ROTACION PURA
138 bool flag1 = false;
139 bool flag2 = false;
140
141
142
143 // FUNCIONES
144
145 int SpeedToTicks(float v) {
146   double cpr = 4096.0;
147   double PID_UPDATE_RATE = ts * 1000.0;
148   double wheelDiameter = 0.34;
149
150   return int(v * cpr / ((PID_UPDATE_RATE)*PI * wheelDiameter));
151 }
152
153 /* The callback function to convert Twist messages into motor speeds */
154 void cmdVelCb(const geometry_msgs::Twist &msg) {
155   float x = msg.linear.x;    // m/s
156   float th = msg.angular.z;  // rad/s
157   float spd_left, spd_right;
158
159
```

```
  if (AUTO == true) {



    lastMotorCommand = millis(); /* Reset the auto stop timer */

    if ((x == 0) && (th == 0)) {
      //moving = 0;
      noInterrupts();
      v_target1 = 0.0;
      v_target2 = 0.0;
      interrupts();
      return;
    }


    //    moving = 1; /* Indicate that we are moving */

    if (x == 0) {  // Turn in place
      spd_right = th * wheelTrack / 2.0;
      spd_left = -spd_right;
    } else if (th == 0) {  // Pure forward/backward motion
      spd_left = spd_right = x;
    } else {  // Rotation about a point in space
      spd_left = x - th * wheelTrack / 2.0;
      spd_right = x + th * wheelTrack / 2.0;
    }

    /* Set the target speeds in meters per second */
    noInterrupts();
    v_target1 = spd_left;
    v_target2 = spd_right;
    interrupts();
  }
}

void MuestreoYPID() {


  double R1 = 0.17;  // radio rueda [m]
  double R2 = 0.17;
  double enc_resolution = 4096.0;  // 1024*4 Quadrature Encoder

  float ticksPerMeter = enc_resolution / (PI * 2.0 * R1);

  //noInterrupts();
  // Actualizamos lectura en la posicion i
  window1[i] = Encoder1.read();
  window2[i] = Encoder2.read();


```

```
211   if (i == window_size - 1) {  // Si estamos al final del array
212     c1 = window1[i] - window1[0];
213     c2 = window2[i] - window2[0];
214     i = 0;
215
216   } else {  // caso general
217     c1 = window1[i] - window1[i + 1];
218     c2 = window2[i] - window2[i + 1];
219     i++;
220   }
221
222   // Ahora calculamos la velocidad
223   /*
224   w = 2pi*n / N*t
225   n = number of pulses/incremento de cuentas
226   N = pulses per rotation/cuentas por revolucion
227   t = sampling period
228   sampling_rate = 1/t --> sampling_rate = 1/(ts*1000)
229   */
230
231   w1 = (2.0 * PI * c1) / (enc_resolution * window_interval);
232   w2 = (2.0 * PI * c2) / (enc_resolution * window_interval);
233
234   // velocidad lineal [m/s]
235
236   v1 = w1 * R1;
237   v2 = w2 * R2;
238
239   // PID
240
241   PID1.error_P = (v_target1 - v1);  // error proporcional
242   PID2.error_P = (v_target2 - v2);
243
244   if (fabs(PID1.output) < MAXoutput) { // limitacion de la accion
    integral
245     PID1.error_I += PID1.error_P * (double)window_interval;
246   }
247   if (fabs(PID2.output) < MAXoutput) {
248     PID2.error_I += PID2.error_P * (double)window_interval;
249   }
250
251   // Si nos movemos, reseteamos el error acumulado
252
253   if ((fabs(c1) > double(threshold)) && (fabs(c1) < 4.0 *
    double(threshold))) {
254     PID1.error_I = 0.0;
255   }
256
257   if ((fabs(c2) > double(threshold)) && (fabs(c2) < 4.0 *
    double(threshold))) {
258     PID2.error_I = 0.0;
```

```
  }

  if ((fabs(v_target1) < 0.1) && (fabs(v_target2) < 0.1)) {
    PID1.error_I = 0.0;
    PID2.error_I = 0.0;
  }

  // output controlador
  PID1.output = PID1.kp * PID1.error_P + PID1.ki * PID1.error_I;
  PID2.output = PID2.kp * PID2.error_P + PID2.ki * PID2.error_I;


  // Clamping

  PID1.output = constrain(PID1.output, -MAXoutput, MAXoutput);
  PID2.output = constrain(PID2.output, -MAXoutput, MAXoutput);


  // ROS

  dt = window_interval;

  dleft = c1 / ticksPerMeter;
  dright = c2 / ticksPerMeter;

  /* Compute the average linear distance over the two wheels */
  dxy_ave = (dleft + dright) / 2.0;

  /* Compute the angle rotated */
  dth = (dright - dleft) / wheelTrack;

  /* Linear velocity */
  vxy = dxy_ave / dt;

  /* Angular velocity */
  vth = dth / dt;

  /* How far did we move forward? */
  if (dxy_ave != 0) {

    dx = cos(dth) * dxy_ave;
    dy = -sin(dth) * dxy_ave;

    x += (dx * cos(th) - dy * sin(th)) * dt;
    y += (dx * sin(th) + dy * cos(th)) * dt;
  }


  if (dth != 0) {
    th += dth;
  }
```

```
310 }
311
312 void Joystick() {
313
314   if (AUTO == true) {
315     return;
316   }
317
318   // (!) it takes about 100 microseconds (0.0001 s) to read an analog
    input,
319   // so the maximum reading rate is about 10,000 times a second.
320
321   double xread = analogRead(x_axis);
322   double yread = analogRead(y_axis);
323
324   // Mapeamos valores de 0 a 1023 a valores normalizados en [-1,1]
325   // map(value, fromLow, fromHigh, toLow, toHigh) SOLO USA ENTEROS
326
327   // up   = 1023,   middle = 512,    down = 0
328   // 3m/s = 1023,   0m/s   = 512,   -3m/s = 0
329
330   // x_mapped = (x - fromLow) * (toHigh - toLow) / (fromHigh - fromLow)
    + toLow;
331   // https://reference.arduino.cc/reference/en/language/functions
    /math/map/
332
333   double fromLow = 0.0;
334   double fromHigh = 1023.0;
335   double toLow = -1.0;  // normalizar valores entre -1 y 1
336   double toHigh = 1.0;
337
338   // DEADZONE
339
340   int deadzone = 20;
341
342   if ((xread >= 512 - deadzone) && (xread <= 512 + deadzone)) {
343     x_mapped = 0.0;
344   } else {
345     x_mapped = (xread - fromLow) * (toHigh - toLow) / (fromHigh -
    fromLow) + toLow;
346   }
347
348   if ((yread >= 512 - deadzone) && (yread <= 512 + deadzone)) {
349     y_mapped = 0.0;
350   } else {
351     y_mapped = (yread - fromLow) * (-toHigh + toLow) / (fromHigh -
    fromLow) - toLow;
352   }
353
354   // DIRECCIONES
355
356   double cpoint = 0.2;
```

```
357   double u1 = 0.5;
358   double u2 = 0.25;
359   double nx = 0.0;
360
361   if (x_mapped > cpoint) {
362     nx = u1 * cpoint + (x_mapped - cpoint) * u2;
363   } else if (x_mapped < -cpoint) {
364     nx = -u1 * cpoint + (x_mapped + cpoint) * u2;
365   } else {
366     nx = u1 * x_mapped;
367   }
368
369   v_target1temp = y_mapped + nx;
370   v_target2temp = y_mapped - nx;
371
372   fromLow = -1.0;
373   fromHigh = 1.0;
374
375   if (y_mapped < 0.0) {  // Limitador marcha atras
376     toLow = -MAXref * 0.6;
377     toHigh = MAXref * 0.6;
378   } else {
379     toLow = -MAXref;
380     toHigh = MAXref;
381   }
382
383   // ROTACION PURA
384   // Algoritmo propuesto para orientar las ruedas y realizar una
    rotación pura
385
386   if ((v_target1temp != -v_target2temp) || (v_target1temp == 0.0)) {
387     flag1 = false;
388     flag2 = false;
389   }
390
391   if ((v_target1temp == -v_target2temp)) {
392
393     fromLow = -MAXref;
394     fromHigh = MAXref;
395
396     if ((v_target1temp < 0.0)) {
397       v_target1temp = -0.5;
398       v_target2temp = -0.2;
399
400       flag2 = false;
401       if (fabs(c1) > 5.0 * threshold) {
402         flag1 = true;
403       }
404     }
405
406     if ((v_target1temp > 0.0)) {
```

```
      v_target1temp = -0.2;
      v_target2temp = -0.5;
      flag1 = false;
      if (fabs(c2) > 5.0 * threshold) {
        flag2 = true;
      }
    }

    if ((flag1 == true) && (flag2 == false)) {
      v_target1temp = -0.5;
      v_target2temp = 0.5;
    }

    if ((flag2 == true) && (flag1 == false)) {
      v_target1temp = 0.5;
      v_target2temp = -0.5;
    }
  }

  // Mapeamos valores de -1 a 1 a valores normalizados en [-MAXref,MAXref]

  noInterrupts();

  v_target1 = (v_target1temp - fromLow) * (toHigh - toLow) / (fromHigh - fromLow) + toLow;
  v_target2 = (v_target2temp - fromLow) * (toHigh - toLow) / (fromHigh - fromLow) + toLow;

  v_target1 = constrain(v_target1, -MAXref, MAXref);
  v_target2 = constrain(v_target2, -MAXref, MAXref);

  interrupts();

}

void Frenos() {

  double comando1, comando2;  // Comandos de los motores

  noInterrupts();
  comando1 = PID1.output;
  comando2 = PID2.output;
  interrupts();

  if (frenoDesactivado == true) {

    if ((fabs(comando1) > 1.0) || (fabs(comando2) > 1.0)) {
      tiempoComandoNoCero = millis();
    }

```

```
455       if ((millis() - tiempoComandoNoCero) > TIEMPOPONFRENO) {
456         digitalWrite(frenos, LOW);
457         frenoDesactivado = false;
458         Silla.motor(1, 0);
459         Silla.motor(2, 0);
460         return;
461       }
462
463       Silla.motor(1, comando1);
464       Silla.motor(2, comando2);
465
466
467
468     } else {
469
470       Silla.motor(1, 0);
471       Silla.motor(2, 0);
472
473       if ((fabs(comando1) > 1.0) || (fabs(comando2) > 1.0)) {
474
475         digitalWrite(frenos, HIGH);
476
477         if (cambiandoEstadoFrenos == false) {
478           cambiandoEstadoFrenos = true;
479           tiempoComandoFreno = millis();
480         }
481
482
483         if ((millis() - tiempoComandoFreno) > TIEMPOQUITAFRENOS) {
484           cambiandoEstadoFrenos = false;
485           frenoDesactivado = true;
486         }
487       } else {
488         cambiandoEstadoFrenos = false;
489       }
490     }
491   }
492
493   void SelectorVelocidad() {
494
495     if (digitalRead(pinAUTO) == HIGH) {
496       AUTO = true;
497     } else {
498       AUTO = false;
499     }
500
501
502     bitWrite(seleccion, 0, digitalRead(Sel1));
503     bitWrite(seleccion, 1, digitalRead(Sel2));
504
505     switch (seleccion) {
```

```
506
507       case 0:
508         MAXref = 0.5;
509         selERROR = false;
510         threshold = 20;
511         // de matriz PID
512         PID1.kp = gains[0][0];
513         PID1.ki = gains[0][1];
514         PID2.kp = gains[0][0];
515         PID2.ki = gains[0][1];
516
517         //Serial.println("Velocidad 1: 0.5 m/s");
518         break;
519
520       case 1:
521         MAXref = 1.0;
522         selERROR = false;
523         threshold = 10;
524         PID1.kp = gains[1][0];
525         PID1.ki = gains[1][1];
526         PID2.kp = gains[1][0];
527         PID2.ki = gains[1][1];
528
529         //Serial.println("Velocidad 2: 1.0 m/s");
530         break;
531
532       case 2:
533         MAXref = 2.0;
534         selERROR = false;
535         threshold = 10;
536         PID1.kp = gains[1][0];
537         PID1.ki = gains[1][1];
538         PID2.kp = gains[1][0];
539         PID2.ki = gains[1][1];
540         //Serial.println("Velocidad 3: 2.0 m/s");
541         break;
542
543       case 3:
544         MAXref = 3.0;
545         selERROR = false;
546         threshold = 10;
547         PID1.kp = gains[1][0];
548         PID1.ki = gains[1][1];
549         PID2.kp = gains[1][0];
550         PID2.ki = gains[1][1];
551         //Serial.println("Velocidad 4: 3.0 m/s");
552         break;
553
554       default:
555         selERROR = true;
556         break;
```

```
  }

  if ((selERROR == false) && (AUTO == false)) {

    // encender los leds correspondientes

    for (int j = 0; j < 4; j++) {

      digitalWrite(leds[i], (i <= seleccion));
    }

  } else if (AUTO == true) {

    for (int j = 0; j < 4; j++) {

      digitalWrite(leds[i], LOW);
    }

  } else {

    // parpadear los 4 leds si hay error

    for (int j = 0; j < 4; j++) {

      digitalWrite(leds[j], (millis() % 2000 > 1000));
    }
  }
}


void EnviaMensaje() {

  if ((millis() - ultimoMensaje) > 100) {


    /* Represent the rotation as a quaternion */
    geometry_msgs::Quaternion quaternion;
    quaternion.x = 0.0;
    quaternion.y = 0.0;
    quaternion.z = sin(th / 2.0);
    quaternion.w = cos(th / 2.0);


    pose_msg.header.stamp = nh.now();
    pose_msg.pose.position.x = x;
    pose_msg.pose.position.y = y;
    pose_msg.pose.position.z = 0;
    pose_msg.pose.orientation = quaternion;
    posePub.publish(&pose_msg);

    twist_msg.linear.x = vxy;
```

```
    twist_msg.angular.z = vth;

    twistPub.publish(&twist_msg);

    ultimoMensaje = millis();
  }
}
void encoderReset() {  // Encoders a 0

  Encoder1.write(0);
  Encoder2.write(0);
}

void PIDReset() {  // PID a 0

  PID1.error_P = 0;
  PID1.error_I = 0;

  PID2.error_P = 0;
  PID2.error_I = 0;

  PID1.output = 0;
  PID2.output = 0;
}

void setup() {

  // PINS

  pinMode(x_axis, INPUT);
  pinMode(y_axis, INPUT);

  pinMode(frenos, OUTPUT);

  pinMode(Sel1, INPUT);
  pinMode(Sel2, INPUT);

  pinMode(pinAUTO, INPUT);

  for (int k = 0; k < 4; k++) {
    pinMode(leds[k], OUTPUT);
  }

  // SERIAL
  Serial.begin(57600);  // Serial USB
  Serial1.begin(9600);  // Serial1: 19(RX), 18(TX)

  // CLEAR ALL
  encoderReset();
  PIDReset();

```

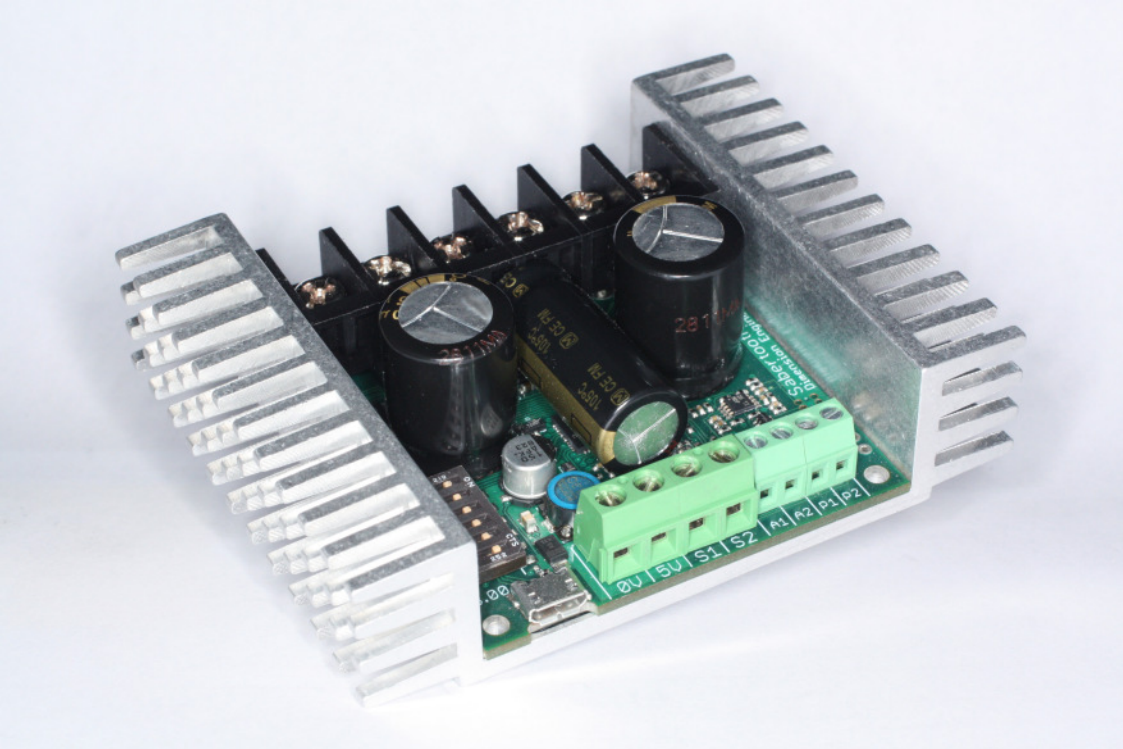
```
659   // INTERRUPCION MUESTREO Y PID
660   Timer1.initialize(ts * 1000.0);        //20 ms (micros)
661   Timer1.attachInterrupt(MuestreoYPID);  //llama a muestreoYPID() cada
    ts
662
663   /* Initialize the ROS node */
664   nh.initNode();
665
666   /* Advertise the Odometry publisher */
667   nh.advertise(twistPub);
668   nh.advertise(posePub);
669   /* Activate the Twist subscriber */
670   nh.subscribe(cmdVelSub);
671
672 }
673
674
675 void loop() {
676
677   if (millis() - UltimoJoystick > 50) {
678     Joystick();
679     UltimoJoystick = millis();
680   }
681
682   Frenos();
683   SelectorVelocidad();
684   EnviaMensaje();
685
686   if (((millis() - lastMotorCommand) > 5000) && (AUTO == true)) {
687     noInterrupts();
688     v_target1 = 0.0;
689     v_target2 = 0.0;
690     interrupts();
691   }
692
693   nh.spinOnce();
694 }
```

Universidad de La Laguna

ANEXO II: DOCUMENTACIÓN SABERTOOTH

Sabertooth 2x32

Dimension Engineering



Sabertooth 2x32 is a dual channel motor driver capable of supplying 32 amps to two motors, with peak currents up to 64 amps per motor. It can be operated from radio control, analog, TTL serial or USB inputs. It uses regenerative drive and braking for efficient operation. A variety of operating modes including tank style mixing and automatic calibration allow most projects to work immediately out of the box. In addition to the standard operating modes, Sabertooth 2x32 features additional signal inputs and power outputs, as well as enhanced configuration options. User-defined operating modes allow for custom operation, such as switching between radio control and computer-driven inputs on the fly, emergency stops or front panel control overrides. The auxiliary power outputs can be configured to allow the Sabertooth 2x32 to operate from a power supply without a parallel battery, or automatic control of electromagnetic brakes. When combined with Dimension Engineering's Kangaroo x2 motion control module, the Sabertooth 2x32 can be used for closed-loop position or speed control with encoder or analog feedback. The state of the driver can be monitored in real time using the USB port in any operating mode, making debugging your project faster and easier. Sabertooth 2x32 is more flexible, robust and powerful than previous motor drivers, while also being easier to use.

Contents

Features

Dual motor driver with mixed and independent options

Sabertooth 2x32 will drive two motors at up to 32A continuous and 64A peak each. These can be mixed together for a tank drive type vehicle, or run independently. It can be controlled by analog voltages, R/C transmitters, TTL serial commands, USB, or a combination of these signals.

USB input

Every third generation Sabertooth motor driver comes with USB standard. This makes operating from a PC or advanced microcontroller as easy as plugging in the cable. Windows drivers are included with the DEScribe PC software, and no driver is required for Linux. USB is also used with to set options, create custom operating modes, monitor the system, and update the firmware with new features.

Auxiliary inputs and outputs

Sabertooth 2x32 has two additional 8A power outputs, which can be set up to operate electromagnetic brakes on motors, act as a voltage clamp to protect power supplies, or power other medium duty loads. Sabertooth 2x32 also has extra signal inputs and extra serial ports which enable better control options.

User-created operating modes for custom applications

We often hear that a motor driver would be perfect if it only had one more input or a minor operating change. Third generation Sabertooth motor drivers like 2x32 have user-scriptable operating modes, which allow you to mix and match analog, R/C, serial and USB inputs, create custom output functions and handle tasks automatically. You can even switch control from one input type to another. Many jobs that would have taken an additional microcontroller can now be handed with just the Sabertooth 2x32.

Self-tuning PID control using the optional Kangaroo x2 expansion board

When used with the optional Kangaroo x2 expansion board, Sabertooth 2x32 works with quadrature encoder or potentiometer feedback for speed or position control. Because it is self-tuning, you can skip the hours or days of work getting your PID coefficients dialed in.

High resolution Synchronous regenerative drive with ultrasonic switching frequency

Sabertooth 2x32 features high resolution inputs and over 4000 output speeds for the smoothest control on the market. The switching frequency is over 29 kHz, so there's no annoying motor whine. The outputs use synchronous rectification for high efficiency and low heat generation, as well as regenerative drive to save energy and extend battery life.

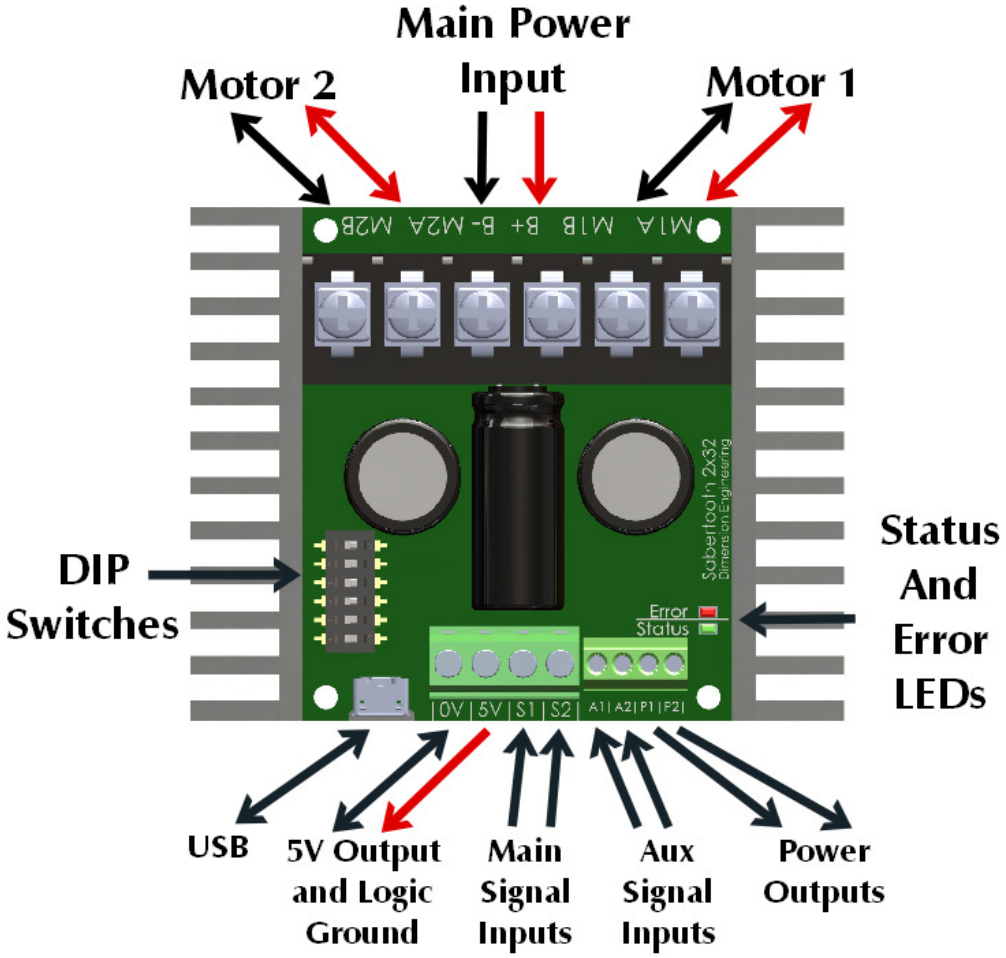
Adjustable current limit, ramp rates and thermal protection

The current and temperature limit of the Sabertooth 2x32 can be set for each motor channel. This can be used to protect the mechanism of your device, as well as protect itself.

Easy mounting and setup

Sabertooth 2x32 features a new heat sink CNC machined from a single block of aluminum. Mounting is accomplished with included 4-40 hardware. All connections are by screw terminals, so no soldering or special cables are required, other than the included micro USB cable.

Overview



Main Power Input: Connect to a 6V-33.6V Battery or Power Supply.

Motor 1 and Motor 2: Connect Motor 1 to the M1A and M1B. Connect Motor 2 to M2A and M2B.

DIP Switches: These are used to set the operating mode and options. Can be changed while operating.

USB: A standard Micro USB port. Connect to a PC or other USB host to control, monitor or modify.

Logic Ground: The 0V logic ground is connected internally to B-.

5V Output: 5V is a regulated 5 volt output. You can use it to power additional circuitry up to 1 amp.

Main Signal Inputs: Connect your main analog, R/C or serial signals here.

Aux Signal Inputs: These may be used for additional control. Optional in most modes.

Power Outputs: These connect to voltage clamp resistors, electromagnetic brakes, field windings, or other moderate power loads. 8A max current per channel.

Status and Error LEDs: These glow and flash to indicate the status of the Sabertooth 2x32.

Specifications

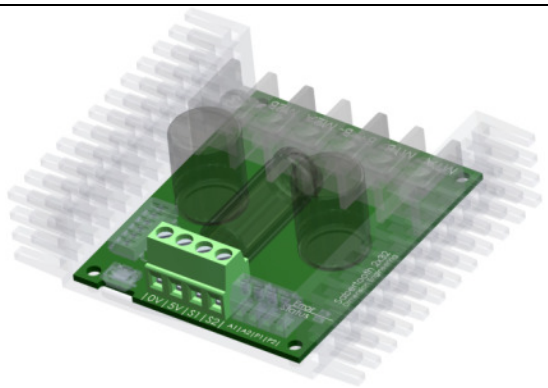
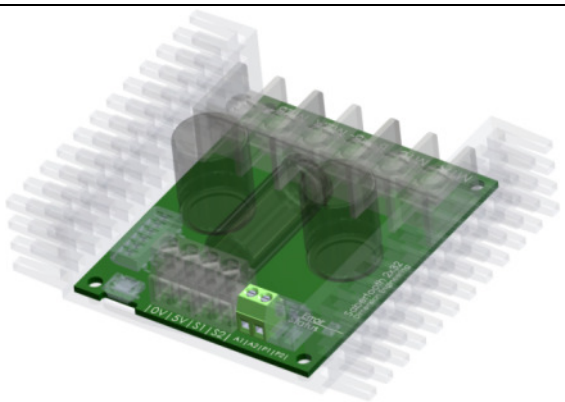
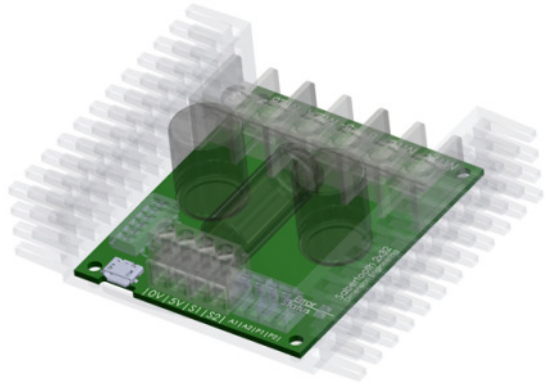
	Mechanical specifications		
Dimensions	2.75 x 3.5 x 1.0 inches (70mm x 90mm x 26mm)		
Weight	4.5 ounces (125 grams)		
	Minimum	Typical	Maximum
Wire size, battery	16 gauge	10 gauge	10 gauge
Wire size, motors	16 gauge	12 gauge	10 gauge
Wire size, signal	28 gauge	24 gauge	18 gauge
Operating temperature	0F (-20C)	70F (25C)	160F (70C)[1]
	Electrical Characteristics		
	Minimum	Typical	Maximum
Input voltage, B+ and B-	6.0 Volts	12 or 24 Volts	33.6 Volts
Continuous output current, M1 and M2	-	-	32 amps[1]
Peak output current, M1 and M2	-	-	64 amps[2]
Output voltage, M1 and M2	-95% of input voltage (average)	-	95% of input voltage (average)
Voltage, P1 and P2	0V	-	Input voltage +.3V
Output current, P1 and P2			8 amps, sink only
Output voltage, 5V	4.85	5.0	5.15
Output Current, 5V	-	-	1A
Input voltage, S1 and S2	-.3V	0V to 5V	12V[3]
Input voltage, A1 and A2	-.3V	0V to 5V	12V[3]
Output Voltage, S2 and A2	0V		3.5V

[1] Maximum continuous output current derates linearly above 40C ambient. At 70C ambient the maximum continuous output current is 10 amps per channel.

[2] Can be reduced by software setting

[3] Stress rating only, signals over 5 volts will be read as 5 volts by the Sabertooth 2x32

Inputs

Main Signal Inputs The main inputs are labeled **S1** and **S2**. They can be set for analog, R/C or serial communications. When using serial, S1 is the RX port and S2 is the TX port.	 Main signal inputs are labeled **S1** and **S2**
Auxiliary Inputs The auxiliary inputs are labeled **A1** and **A2**.They can be set for analog, R/C or serial communications. When using serial, A1 is the RX port and A2 is the TX port.	 Auxiliary signal inputs are labeled **A1** and **A2**
USB Input The Sabertooth connects via a standard micro USB plug. A windows driver installs with the DEScribe PC software. Linux and Mac will work out of the box with no driver. The USB port is also used to communicate with DEScribe to configure the Sabertooth 2x32.	 USB input. The USB input is not labeled.

Sabertooth 2x32 has four logic inputs S1, S2, A1 and A2. The analog input range is 0 to 5 volts. Digital signals can be 3, 3.3 or 5v logic. All third generation drivers such as Sabertooth 2x32 include a USB port. The USB port can be used for control from a PC or embedded processor like a Raspberry Pi. It can also be used to monitor the inputs and outputs, set up user modes and custom settings, and update the firmware.

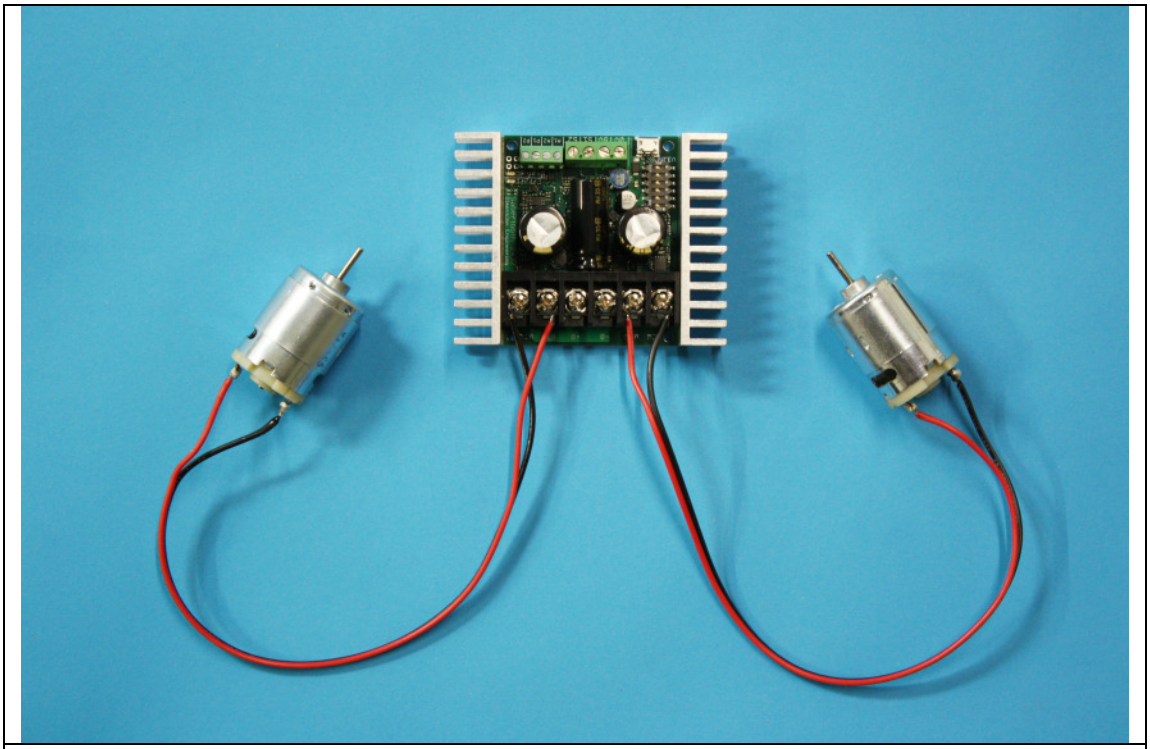
Outputs

Sabertooth 2x32 is a dual motor driver, and can drive two motors at up to 32 amps continuous and 64 amps peak current per channel. In addition, there are two 8 amp auxiliary outputs and two 20 milliamp indicator outputs.

Motor Outputs

Sabertooth 2x32's M1 and M2 motor outputs are 12 bit, synchronous regenerative motor drives. They have a switching frequency of 30 kHz for silent operation. Each channel has a programmable current limit. It is also possible to disable regeneration to drive loads like lamps or anodizing tanks, or completely disable the outputs and braking to allow motors to freewheel.

Typical motors used with a Sabertooth 2x32 include fractional horsepower permanent magnet motors, wheelchair type motors, scooter type motors and brush type cordless tool motors. Most two wire, permanent magnet motors can be made to work. Sabertooth motor drives have also been used to drive large solenoids, lamps, heaters, coolers, electromagnets, shakers and transducers. To minimize heating and losses, use as large a wire to your motors as is practical. 12 gauge is typical for short wiring runs.



Two small motors connected to the Sabertooth 2x32's Motor Outputs.

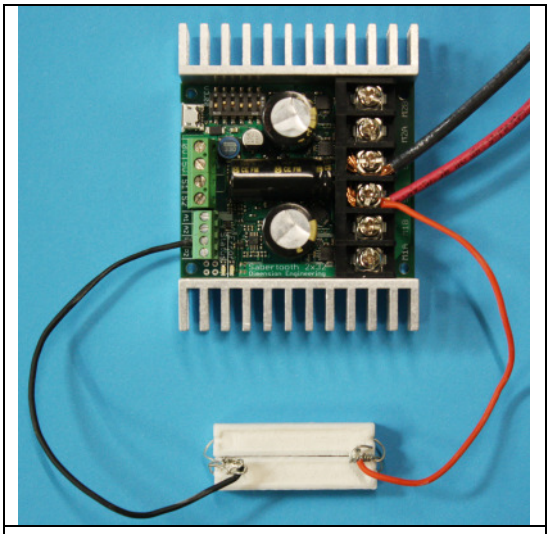
Indicator Outputs

Two of the signal inputs, S2 and A2, can be used as 20 milliamp, 3.3 volt outputs. This is typically used to drive LEDs for remote control panels or signal back to microcontrollers. The indicator outputs are

secondary functions of the S2 and A2 inputs. When in indicator mode, these pins have a built-in 220 ohm series resistor, so they can drive loads directly.

Power Outputs

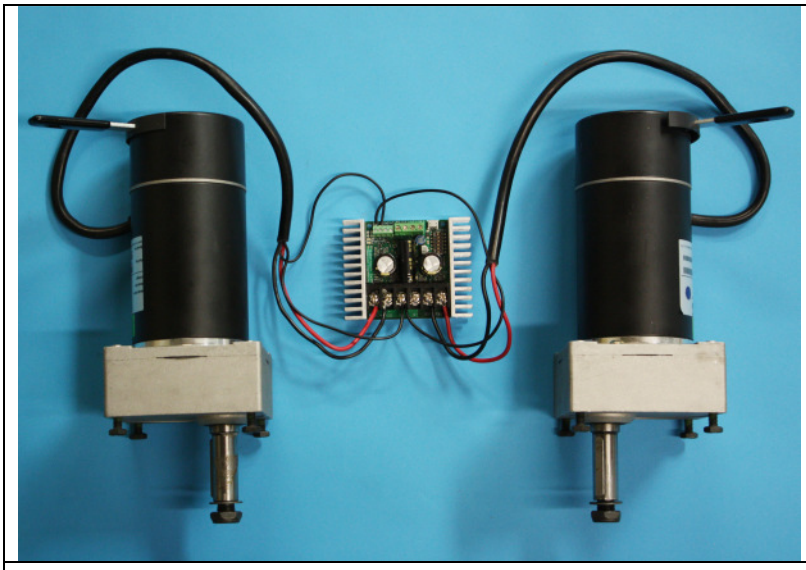
These open collector outputs can sink up to 8 amps of current each. The power outputs can be configured as Voltage Clamps, Brakes or Controllable Outputs.



Power output connected to a resistor and configured as a voltage clamp

Voltage Clamp

With a typical regenerative driver, it is difficult to operate from a power supply, because while braking there is nowhere for the regenerated energy to dissipate. This can lead to power supply shutdown or damage. By connecting the power outputs to a resistor pack (sold separately, or construct your own) the Sabertooth 2x32 can operate from a power supply without an additional battery or circuitry. The resistor's value should be calculated to provide the typical motor current or 8 amps, whichever is less. Voltage clamp mode is selected using the Power Outputs tab in the DEScribe software. Voltage Clamp is the default behavior for both the P1 and P2 outputs.



Power outputs connected to electromagnetic brakes.

Brakes

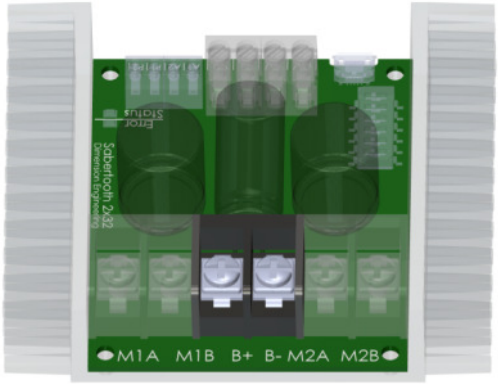
The power outputs can also be used to operate electromagnetic brakes. Systems such as wheelchairs often have brakes to prevent rolling away when power is removed. Brakes are also used in CNC machines and automation to hold alignment when power is removed or reduce power consumption. In brakes mode, the brakes automatically disengage when motion is commanded, and engage when the motor stops. When using Brakes mode, the P1 output is linked to the M1 motor output, and the P2 output is linked to the M2 motor output. With the brake timing changed, brakes mode can also be used to control the field in a shunt wound or separately excited motor. Brakes mode must be selected and configured from the Power Outputs tab using the DEScribe software.

Controllable Output

Finally, the power outputs can be used as additional variable power outputs. These could be used to drive fans, solenoids, valves, heaters or similar medium power devices. This is primarily used with serial or USB inputs, or custom User Mode programs. Controllable Output must be selected in DEScribe.

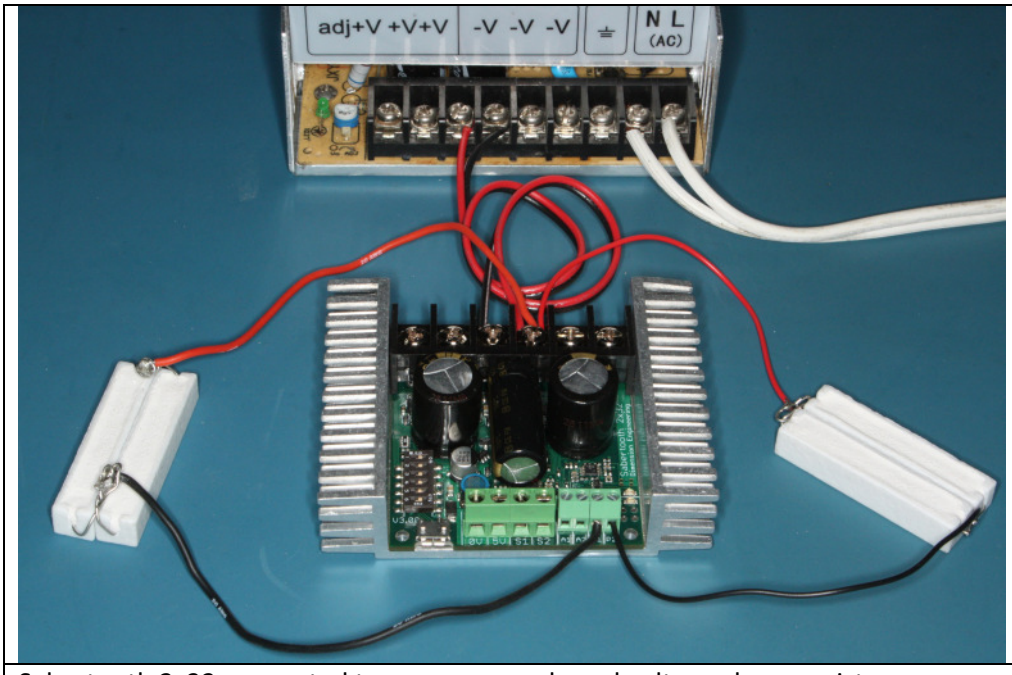
Input Power

Sabertooth 2x32 can be used with power supplies or batteries. Input power is connected to the center power terminals labeled B+ and B-. The input voltage range of the Sabertooth 2x32 is 6.0V to 33.6V. The input current is dependent on the motors being used and the load placed upon them. The input current can be limited by reducing the current limit of one of both motor channels.



The main power inputs are labeled B+ and B-

Power Supply



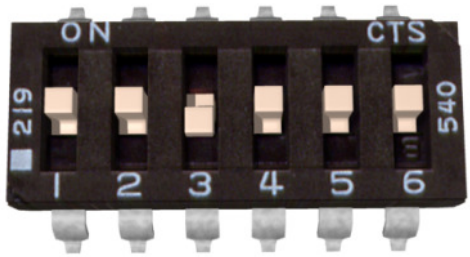
Sabertooth 2x32 connected to a power supply and voltage clamp resistors.

Sabertooth 2x32 can be used with a suitably sized power supply. Power supplies used with a Sabertooth 2x32 should have output voltages between 7V and 30V. It is important to make sure the power supply being used can produce enough current for the load. Power Supply mode is selected by setting DIP switch 3 to the ON position. One major improvement in the third generation Sabertooth motor drivers is the addition of the Power outputs P1 and P2. These can be used with an appropriate resistor to

dissipate the regenerative current generated when the device slows down or stops. This allows the use of power supplies without batteries or capacitors to absorb the regenerated energy. A calculator on the Dimension Engineering website will assist with choosing appropriate resistor packs. At least one of the power outputs should be configured as a voltage clamp when using a power supply.

Battery

Sabertooth 2x32 can be used with a variety of battery types and voltages. When running from a battery that can be damaged by deep discharge, such as a lithium polymer or lead acid battery, you should set DIP switch 3 to the battery protect position, which is OFF. When in this position, the Status LED will blink out the number of detected cells, followed by a pause, in a repeated pattern. This is useful to verify that the Sabertooth has detected the correct number of cells. When the battery is depleted, the Sabertooth will stop driving the motors, and the Status and Error LEDs will both blink in sync. When using a battery, please ensure that it can handle the current draw the motors will supply. Because Sabertooth is a regenerative motor driver and will return energy to the battery when braking is commanded, only rechargeable batteries are recommended.



DIP switch 3 in the OFF position selects Battery Protect mode

With the DEScribe PC software, you can change what type of battery the Sabertooth is protecting. The default setting for battery protect is Lithium Polymer. The options for batteries are laid out in the following table. For each type, you can modify the minimum cell voltage.

Battery Type	Number of series cells in pack	Nominal Voltage
Lithium Polymer or Lithium Ion	2 to 8	7.4V to 29.6V
Lithium FerroPhosphate ($LiFePO_4$)	3 to 9	9.9V to 29.7V
Lead Acid	1 to 4 6V batteries or 1 to 2 12V	6V to 24V
Nickel Metal Hydride (NiMH)	6 to 20	7.2V to 24V
Nickel Cadmium (NiCd)	6 to 20	7.2V to 24V
Custom	1 to 20	-

By default, the number of cells is automatically detected. The Sabertooth assumes the battery is fully charged when it powers up. With high cell count battery packs that are partially discharged, this can be inaccurate. If you are not planning to use multiple voltages, you can specify the exact battery that will be used to avoid this problem.

Wiring

Power connections

As a general rule of thumb, you should use the thickest wire that is practical to make power connections, especially on the battery leads. Using undersized wire will lead to the wire getting hot, and can lead to elevated temperatures on the Sabertooth 2x32 as well.

The main power connections to the Sabertooth 2x32 are on the rear edge of the board. Connections are made to large black screw terminals. These terminals will accept 10 to 24 gauge wire. Using stranded wire it is possible to run twinned 10 gauge wire connections to the battery terminals. This is often a good idea if your design will be running both motors near or above the 32 amp continuous limit. For the motor connections, single 10 gauge wires should be sufficient for all applications.

Motor 1 connects to the M1A and M1B terminals. Motor 2 connects to the M2A and M2B terminals.

If you are using the power outputs as regenerative clamps, one side of the resistor pack connects to the P1 terminal, and the other side connects to the B+ terminal. You may find it easier to connect the positive side of the power input through a bus bar or in the wiring harness instead of at the Sabertooth itself. This is acceptable. By default the power outputs are set up for regenerative clamp.

If you are using the power outputs for electromagnetic brakes, the positive side of both brakes connect to the B+ terminal. The negative side of the M1 brake connects to P1, and the negative side of the M2 brake connects to P2. Remember that to use brakes you must set the power outputs to brakes mode using the DEScribe PC software.

If you are using the power outputs at a voltage other than the main system voltage, such as to run a 5v cooling fan, connect the negative side of the device being powered to the P1 or P2 auxiliary power output, and the positive side to that supply voltage. If you are using an inductive load such as a motor from a different voltage, you will need to install a flyback diode across the device to prevent problems.

Signal connections

The signal connections, as well as the auxiliary power outputs, connect to the smaller green screw terminals on the front edge of the board. All the signal inputs can accept signals between 0V and 5V. In digital input modes, logic high can be between 2.7V and 5V. This allows for interface to boards using 3.3 or 2.7 volt microcontrollers without the need for level translators.

USB: The USB port is used for connection to a PC, tablet or advanced microcontroller. If only the USB port is connected, the internal logic circuitry of the Sabertooth will be active, but neither input nor output terminals will operate. This is useful to enable changing the operating options or updating the firmware without powering on the entire device. In a production setting, it is also useful to set up the Sabertooth 2x32 before installation.

0V: This is the main logic ground of the Sabertooth 2x32. It is internally connected to B-.

5V: This is a 5V output, and can supply up to 1A of current to devices such as receivers, potentiometers, microcontrollers or servos.

S1: This is a main signal input. Its functionality depends on the operating mode. Input voltages into S1 should be between 0V and 5V.

S2: This is a main signal input. Its functionality depends on the operating mode. Input voltages into S2 should be between 0V and 5V. S2 can also be set up as an Indicator type output.

A1: This is an auxiliary signal input. Its functionality depends on the operating mode. . Input voltages into A1 should be between 0V and 5V.

A2: This is an auxiliary signal input. Input voltages into A2 should be between 0V and 5V. A2 can also be set up as an Indicator type output.

Wiring Summary

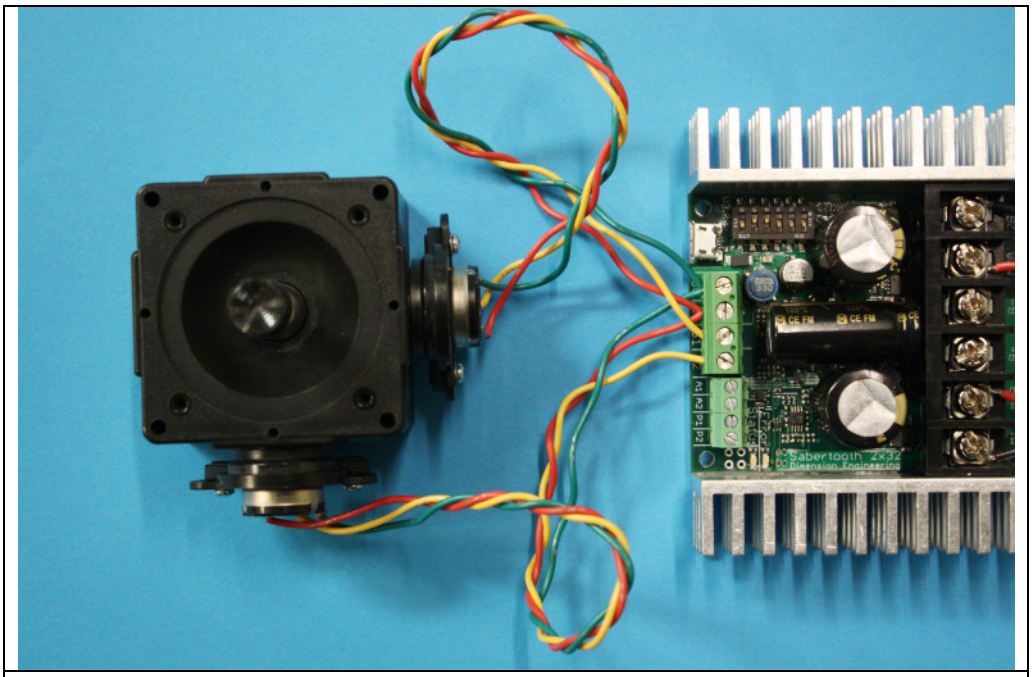
Name	Description	Voltage	Current
B+	Power input positive	6V – 33.6V	64A /128A peak
B-	Power input negative	0V only	64A /128A peak
M1A	Motor 1 output	0V – 33.6V	32A/64A peak
M1B	Motor 1 output	0V – 33.6V	32A/64A peak
M2A	Motor 2 output	0V – 33.6V	32A/64A peak
M2B	Motor 2 output	0V – 33.6V	32A/64A peak
0V	Logic ground	0V only. Tied internally to B-	2A
5V	5V power output	5V	1A
S1	Signal input 1	0V – 5V	1mA
S2	Signal input 2, serial TX	0V – 5V	1mA in, 20mA out
A1	Auxiliary input 1	0V – 5V	1mA
A2	Auxiliary input 2, serial TX	0V – 5V	1mA in, 20mA out
P1	Power output 1. Open collector current sink only.	0V – 33.6V	8A
P2	Power output 2. Open collector current sink only.	0V – 33.6V	8A
USB	Micro USB port		100mA max

Control Modes

Sabertooth 2x32 has four main control modes, plus a special User mode that can be used to create custom control modes. Control Modes are selected via the six DIP switches. DIP switches 1 and 2 select the control mode, and DIP switches 4, 5 and 6 select the options within each control mode.

Analog

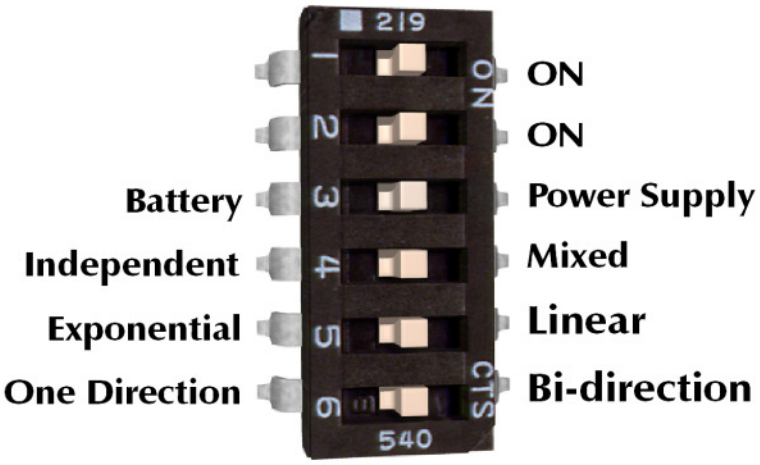
Analog control uses analog voltages to send commands to the Sabertooth 2x32. This is the simplest way to control a Sabertooth. These voltages can be generated by potentiometers, switches, joysticks or digital to analog converters. The input voltage range is 0 volts to 5 volts. Custom ranges can be set using the DEScribe PC software. In Analog the S1, S2, A1 and A2 terminals are analog inputs. They are internally pulled down to 0V if not connected to a signal.



An analog joystick connected to a Sabertooth 2x32

To use analog mode with a potentiometer or joystick, connect the negative terminals to 0V. Connect both positive terminals to 5V. Feed the signals into S1 and S2. If used, A1 and A2 are connected the same way.

Analog Mode DIP Switch Settings



DIP switch 4 selects between mixed and independent mode.

Mixed mode is selected by setting DIP switch 4 to the ON position. Mixed mode is primarily used for differential drive mobile vehicles. The signal going to S1 controls the forward/backwards speed of both motors. The signal going to S2 controls the right/left turning of both motors.

Independent mode is selected by setting DIP switch 4 to the OFF position as shown. In Independent mode, the speed of the motor connected to the M1 motor outputs is controlled by the analog signal sent to the S1 input, and the speed of the motor connected to the M2 output is controlled by the analog signal sent to the S2 input.

DIP switch 5 selects between linear and exponential control.

Linear control makes the speed of the output motor directly proportional to the input voltage. This is best for control systems and speed adjustment dials. Linear control is selected by setting DIP switch 5 to the ON position.

Exponential control makes the motors less responsive around the zero speed point. This is useful for fine control of small motions, while not losing the ability to go full speed. The exponential mapping is saved in EEPROM, and can be modified with the DEScribe PC software. Although this mode is called exponential, you can use the software to create other input to output functions as well.

DIP switch 6 selects between bidirection and single direction modes.

Bidirection control is selected by setting DIP switch 6 to the ON position. In this mode, an input signal of 2.5 volts is stopped. Voltages higher than 2.5 volts cause the motor to go forward, and voltages less than 2.5 volts cause the motor to go in reverse.

Single direction mode is selected by setting DIP switch 6 to the OFF position. In this mode, a command of 0V stops the motor and a command of 5V is full speed. The direction the motor runs can be controlled by the A1 and A2 inputs. If you connect a switch between 5V and A1, it will act as a forward/reverse

switch for the signal being fed into S1. If only a single direction is needed, only S1 and S2 need to be connected.

Speed ramping

In all analog modes except Single Direction, an analog signal sent to the A1 auxiliary input controls the ramp rate for both channels of the motor driver. This is useful to make gentle motions or limit peak currents due to acceleration. For example, a ride-on electric skateboard that started at full power immediately will throw its rider off, but one with a several second ramp rate is easy to ride. If this input is not connected, the input's internal pull-down will automatically set the fastest response. If adjustable ramping is not desired, leave the A1 input disconnected. The ramp rate can also be set using the DEScribe PC software.

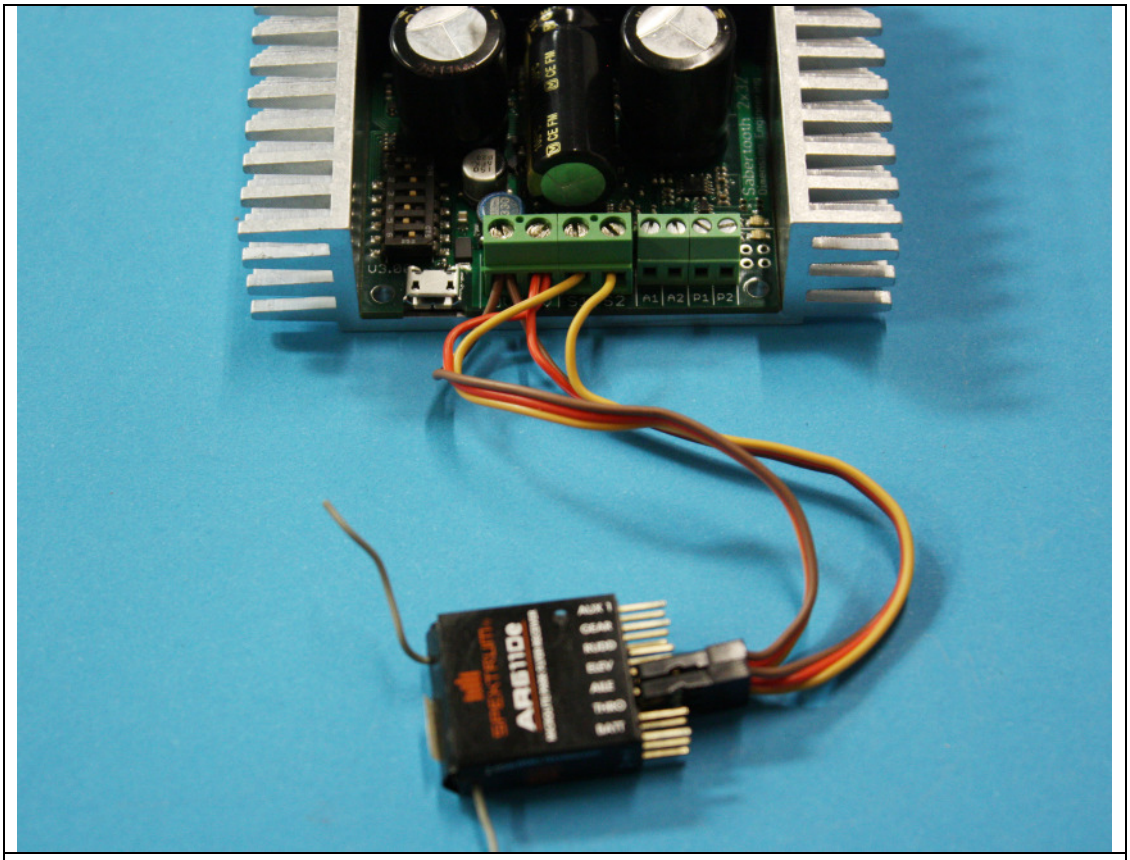
Maximum speed

In all analog modes except Single Direction Mode, a signal sent to the A2 auxiliary input sets the maximum speed for both motor outputs. This can be used for fine control, because as the maximum speed scales down, so do all other inputs. You might want to reduce the maximum speed for very precise control while inspecting a specimen, then turn the speed all the way back up to jog over to the next. If this input is not connected, the input's internal pull-down will automatically set the fastest speed, so if adjustable max speed is not desired, you should leave the A2 input disconnected.

Other analog control mode options, such as automatic calibration, are available by using the DEScribe PC software. You can also use DEScribe to create custom analog control modes.

Radio Control

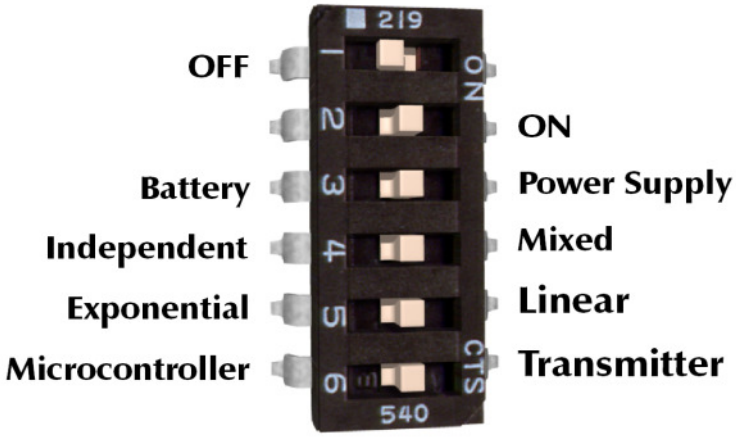
Radio Control uses R/C (servo) pulses to send commands to the Sabertooth 2x32. These signals are generated by R/C transmitters and receivers, or by microcontrollers. Anything that can generate servo signals can be used to drive a Sabertooth in Radio Control mode. In Radio Control mode the S1, S2, and A2 inputs are set up to read R/C pulses. The A1 input is set up as an analog input and can be used with a potentiometer for an adjustable ramp rate.



A radio receiver being used to control a Sabertooth 2x32

R/C signals are typically carried on servo pigtails, which are three wires. The brown wire is ground, and connects to 0V. The red wire connects to 5V. The orange or white wire carries the signal, and connects to S1, S2 or A2. Sabertooth 2x32 will power a receiver, so no separate receiver battery is required.

Radio Control Mode DIP Switch Settings



DIP switch 4 selects between mixed and independent mode.

Mixed mode is selected by setting DIP switch 4 to the ON position as shown. Mixed mode is primarily used for differential drive mobile robots. The signal going to S1 controls the forward/backwards speed of both motors. The signal going to S2 controls the right/left turning of both motors.

Independent mode is selected by setting DIP switch 4 to the OFF position as shown. In Independent mode, the speed of the motor connected to the M1 motor outputs is controlled by the R/C signal sent to the S1 input, and the speed of the motor connected to the M2 output is controlled by the R/C signal sent to the S2 input.

DIP switch 5 selects between linear and exponential control.

Linear control makes the speed of the output motor directly proportional to the input signal. This is best for control systems, as well as microcontrollers. Linear control is selected by setting DIP switch 5 to the ON position.

Exponential control makes the motors less responsive around zero speed. This is useful for fine control of small motions, while not losing the ability to go full speed. It is especially useful for differential drive robots and vehicles. The exponential mapping is saved in EEPROM, and can be modified with the DEScribe PC software.

DIP switch 6 selects between Transmitter and Microcontroller mode.

Transmitter mode is selected by setting DIP switch 6 to the ON position. Because all R/C transmitters use slightly different timing, transmitter mode automatically calibrates the stopped position at startup. The maximum and minimum signals are constantly recomputed. In this way, the Sabertooth learns the transmitter's signals every time it is powered up. In Transmitter mode, loss of signal from the receiver will stop the motor driver. Be advised that some receivers, especially 2.4 GHz receivers, will continue to put out servo signals even if communication with the transmitter is lost, so check the manual for your receiver's failsafe behavior.

Microcontroller mode is selected by turning DIP switch 6 OFF. In microcontroller mode, the input pulse ranges are fixed. A 1500us signal is stopped, 1000us is full reverse and 2000us is full forward. To allow for slower microcontrollers like Basic Stamps, by default there is no timeout in Microcontroller mode.

DEScribe can be used to override these settings and use other input ranges, as well as change the timeout behavior and timing. Using DEScribe, there is also a Saved calibration mode which learns the transmitter settings one time only, and a Joystick mode which only automatically calibrates the center position.

Speed ramping

In all R/C modes, an analog signal sent to the A1 auxiliary input controls the ramp rate for both channels of the motor driver. This is useful to make gentle motions or limit peak currents due to acceleration. For example, an R/C lawnmower that started at full power immediately might tear up soft grass. If this input is not connected, the input's internal pull-down will automatically set the fastest response. If adjustable ramping is not desired, leave the A1 input disconnected. The ramp rate can also be set using the DEScribe PC software.

Flip input

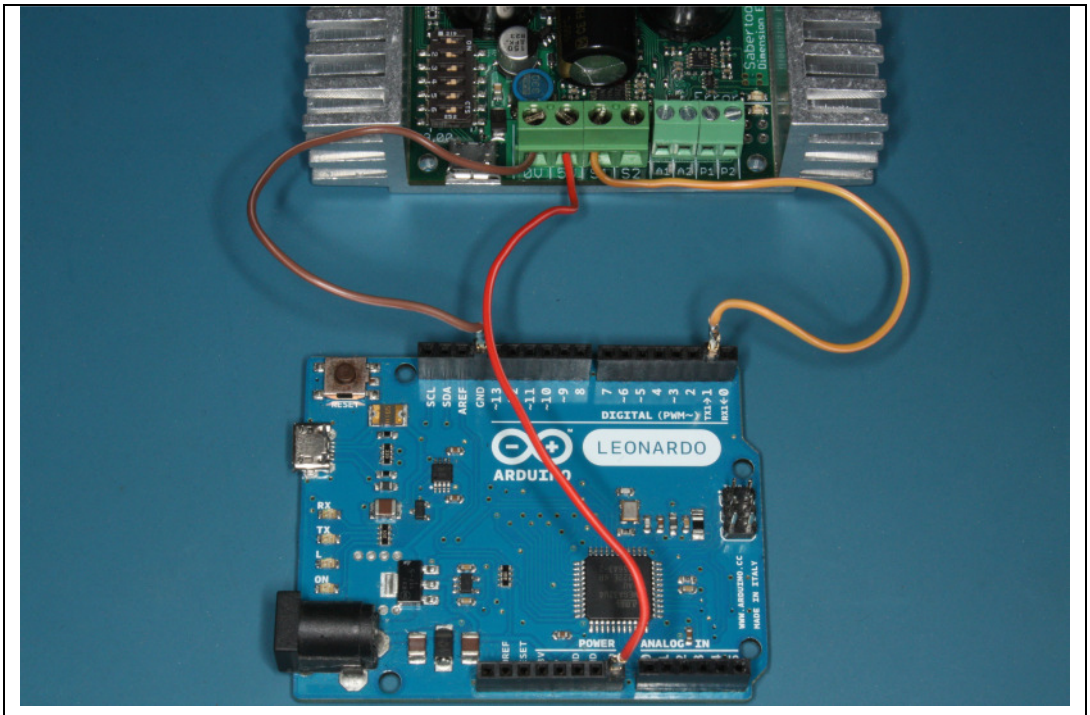
The auxiliary A2 input is also an R/C input. It is an optional Flip channel. In mixed mode, an R/C signal coming from a switch (such as the channel 5 gear switch on the transmitter) is used to select between normal and inverted mode. This is used for mobile robots that can run both right side up and upside down. If the robot is flipped upside down, flipping the gear switch will correct the steering.

Serial

Serial mode is used to control the Sabertooth 2x32 from a PC or microcontroller.

Setup

There are a variety of serial communication protocols that can be used with Sabertooth 2x32. By default, all of these modes use 9600 baud, 8N1 TTL serial levels. The baud rate can be changed using the DEScribe PC software.

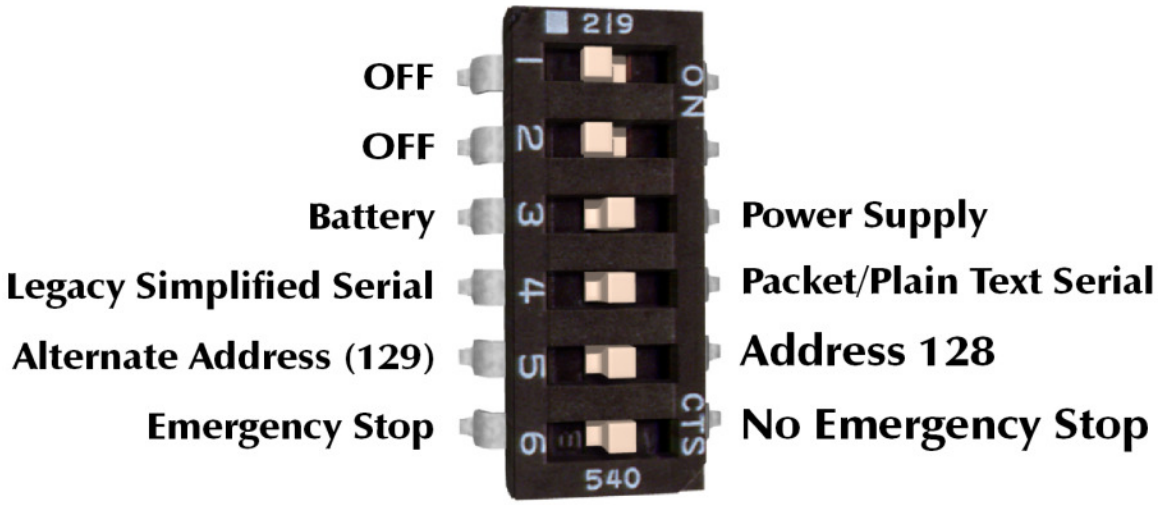


A Sabertooth 2x32 receiving serial commands from an Arduino microcontroller

Wiring

Wire the S1 connection on the Sabertooth 2x32 to the serial TX pin of your microcontroller. To use any command that reports data back from the Sabertooth 2x32, connect the S2 terminal to the Serial RX pin of your microcontroller. Connect the 0V terminal to the microcontroller's ground. The Sabertooth 2x32's 5V output may be used to provide power to the microcontroller.

Serial Mode DIP Switch Settings



DIP Switch 4 selects between packet/plain text serial and legacy simplified serial.

If DIP switch 4 is in the ON position, the Sabertooth 2x32 is listening for packet serial or plain text serial commands. It will automatically respond to either type of command.

If DIP switch 4 is in the OFF position, the Sabertooth 2x32 is listening for legacy simplified serial commands. This mode uses single-byte commands to command each motor, and is included primarily for compatibility. If you are developing a new design, we recommend using one of the newer command protocols.

DIP Switch 5 selects the packet serial address.

In packet serial mode, DIP switch 5 sets the packet serial address. The ON position sets address 128, and the OFF position sets the address to a user-definable alternate. By default, this is address 129, but it can be changed in DEScribe. Packet serial addresses are used to run multiple Sabertooth motor drivers on the same serial line.

If you are sending plain text serial commands, DIP switch 5 has no effect.

In legacy simplified serial, DIP switch 5 selects between 9600 baud (switch 5 ON) and a user-defined setting, which defaults to 2400 (switch 5 OFF)

DIP Switch 6 enables or disables emergency stops.

If DIP switch 6 is in the OFF position, emergency stops are enabled. This emergency stop is active low and internally pulled down. To enable the M1 output, connect the A1 terminal to 5V. To enable the M2 output, connect the A2 terminal to 5V. If these connections are broken and emergency stops are enabled, the motors will stop immediately. This might be used for a safety E-stop in a machine, to only allow motion while a dead-man switch is held, or to detect a disconnected control cable.

Plain Text Serial

Plain text serial mode uses ASCII formatted plain text strings to control the Sabertooth 2x32. This makes serial communications especially easy, as one can open a terminal window on a PC and start typing commands, or use a printf command on a microcontroller. Using Plain Text Serial mode allows full control of the Sabertooth 2x32. The tradeoff is plain text takes more data to send the same information, so the maximum update rate may be less. Plain text serial has an optional checksum, but by default does not require one. Data integrity is not assured unless DEScribe is used to make the checksum required.

All commands follow the same format. All commands consist of a two letter destination, followed by a colon, followed by the command and a newline (Enter key). For example, **M1: -532(Enter)**

Basic commands

Outputs are controlled by sending the destination address, followed by a colon, followed by the power setting, followed by a newline. Most settings take commands from -2047 to 2047. Commands outside this range will be ignored. If you are using a terminal program to command the Sabertooth, remember that the command only takes effect when you press the Enter key.

Destination address	Description
M1	Motor 1
M2	Motor 2
MD	Drive channel. Both motors .Forward/Backwards in Mixed Mode
MT	Turn channel. Both motors. Right/Left in Mixed Mode
P1	Power output 1
P2	Power output 2
R1	Ramp rate motor 1
R2	Ramp rate motor 2
Q1	Auxiliary variable 1
Q2	Auxiliary variable 2

Example	Result
M1: 2047\r\n	Motor 1 will go forward full speed.
M2: -1023\r\n	Motor 2 will go backwards at half speed.
M1: 0\r\n M2: 0\r\n	Motors 1 and 2 will stop.
MD: 0\r\n MT: -512\r\n	The robot will turn left at ¼ speed. Please note that to control using the mixed commands MD and MT, you must have sent both a turn and a drive command at some point. Once you have sent them both the first time, you only have to update the turn or drive commands as you desire, it is not necessary to always send them both.
P1: 2047\r\n	Power output 1 will be set to full power. Note that the power outputs are only controllable if they are set as additional output using the DEScribe software.

	Otherwise they will perform their normal voltage clamp or brake behavior and ignore serial input.
R2: 2047\r\n	This sets the speed ramping to the maximum amount. At this setting, it will take the Sabertooth approximately 8 seconds to go from a stop to full speed.
Q1: 1\r\n	By default the Q parameters control motor freewheeling. Setting a positive value will disable the motor channel. Freewheeling and shutdown are different. In a freewheel state the motor will act as though there is no power applied and the motor leads are floating. This makes the motor shaft easier to turn manually. In a shutdown state, the motor will act as if there is no power and the motor leads are connected together. This makes the motor shaft harder to turn manually. In user modes, the Q parameters are often used to change the operation of the program or activate special modes.

Note: \r\n is not visible in a terminal program, but must be included when using a function such as printf to designate the end of the message.

Get Commands

Values can be read back from the inputs and outputs using the get command. Note that when reading from the inputs, it will read according to the current pin configuration. The Sabertooth 2x32 will respond back in the same format: the address, followed by a colon and a space, followed by the value and a newline. Remember that when connected using USB, the get commands are always available, even if you are using R/C or analog mode. This can be helpful for debugging.

Command	Result	Examples
M1: get\r\n M2: get\r\n	Returns the duty cycle of the M1 or M2 output, from -2047 for full reverse to 2047 for full forward.	M1: get\r\n might return M1: -875\r\n
P1: get\r\n P2: get\r\n	Returns the duty cycle of the P1 output. -2047 is no output, and 2047 is full power. If the power output is set up for voltage clamp or brakes, this can use used to determine if the brakes or output clamp is engaged.	P1: get\r\n might return P1: -2047\r\n P2: get\r\n might return P2: 1776\r\n
S1: get\r\n S2: get\r\n	Returns the input value for the S1 or S2 inputs. In USB mode, by default these are set as analog inputs. In packet serial mode, they are serial RX and TX. By creating a user mode	S1: get\r\n might return S1: 0\r\n S2: get\r\n

	program, they can be set to any of the input types.	might return S2: 2012\r\n
A1: get\r\n A2: get\r\n	Returns the input value for the S1 or S2 inputs. By default these are analog in USB and serial modes.	A1: get\r\n might return S1: -2047\r\n

Extended Get commands

In addition to these values, outputs of the Sabertooth such as motor current and battery voltage can be read by using the extended get commands. Because each motor has its own current and temperature, the structure is slightly different. The destination address should be M1 or M2, and then the command letter is added. On the reply, the type of reading is added before the value. Some values, such as the battery voltage, are shared between both motors, so polling either M1 or M2 will return the same value.

Command	Result	Examples
M1: getb\r\n M2: getb\r\n	Returns the battery voltage in tenths of a volt. A battery reading of 12.5 volts will report as B125	M1: getb\r\n might return M1:B240\r\n
M1: getc\r\n M2: getc\r\n	Returns the motor current in tenths of an amp. Please note this is a noisy signal and may vary by up to several amps. This is normal. Positive current values mean energy is being drawn from the battery, and negative values indicate energy is being regenerated into the battery.	M1: getc\r\n might return M1:C320\r\n M2: getc\r\n might return M2:C-20\r\n
M1: gett\r\n M2: gett\r\n	Returns the temperature of the output transistors for this channel, in degrees C.	M1: gett\r\n might return M1:T30\r\n M2: gett\r\n might return M2:T85\r\n

Note: \r\n is not visible in a terminal program, but must be included when using a function such as scanf to designate the end of the message.

Additional commands

Command	Result	Examples
M1:shutdown\r\n M2:shutdown\r\n	Shuts off the motor output. Using the shutdown command will put the motor in a hard brake state.	M1: shutdown\r\n M2: shutdown\r\n
M1:startup\r\n M2:startup\r\n	Returns the motor channel from a shutdown state to normal operation.	M1: startup\r\n M2: startup\r\n

Packet Serial

Packet serial mode uses much the same commands as plain text serial mode, but does so in a more compact binary communication protocol with a reliable checksum or CRC. There are open source libraries for Arduino and C# and sample code for both on the Dimension Engineering website, as well as documentation on the protocol itself.

Legacy Packet Serial

For compatibility with existing programs, Sabertooth 2x32 supports the same 4 character packet serial commands as previous generation Sabertooth motor drivers. Using these commands only allow for 8 bits of motor output precision, and does not have a robust CRC as the new packet serial commands do, but they work acceptably in most applications. The Sabertooth 2x32 will automatically differentiate between the legacy and new commands, so the setup is the same as for Packet Serial. Using the legacy commands does not allow motor driver information such as battery voltage, temperatures or currents to be read back from the Sabertooth.

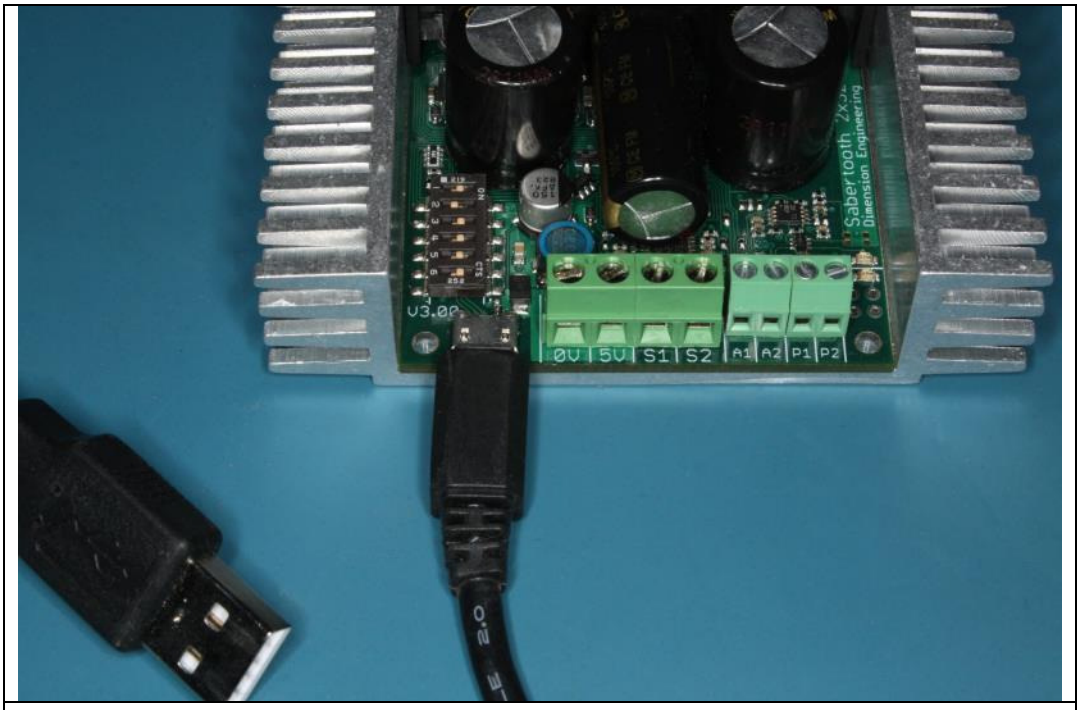
The legacy packet serial commands are documented in a separate application note on Dimension Engineering's website.

Legacy Simplified Serial

For compatibility with existing programs, Sabertooth 2x32 supports Legacy Simplified Serial mode, which is the same as Simplified Serial mode on previous generation motor drivers. The DIP switch settings are different, and the auxiliary inputs are not used.

USB Mode

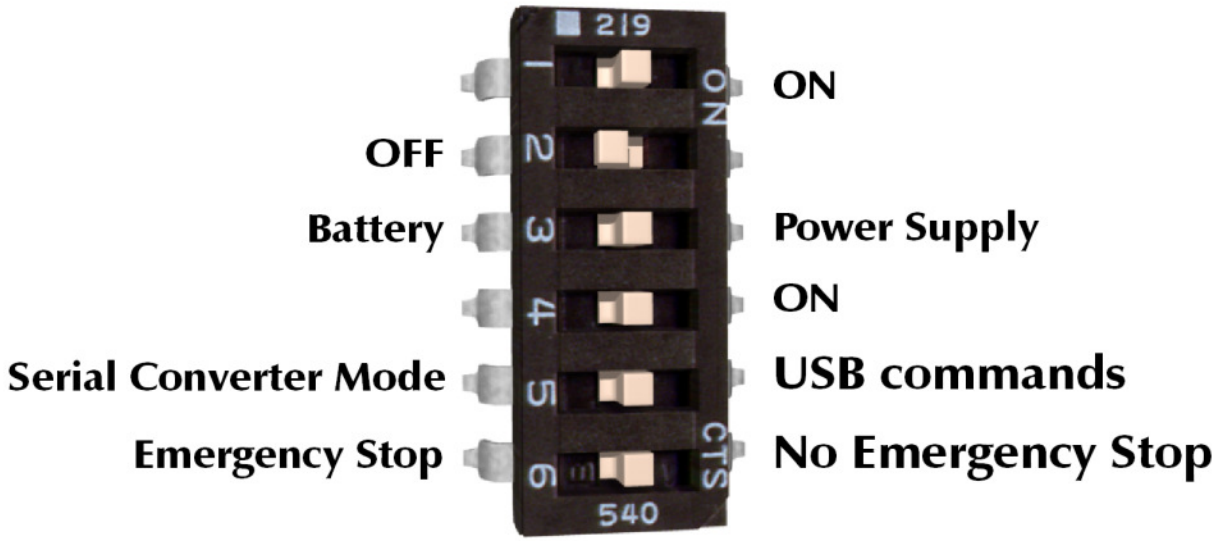
USB mode takes primary input from the USB port, using the same commands as the serial modes. USB mode can also use the Sabertooth 2x32 as a USB to serial converter, to relay commands from the PC to other serial devices, such as another Sabertooth motor driver or a Kangaroo x2 motion control card.



A Sabertooth 2x32 connected to a USB cable

Connect a USB Micro B cable to the USB port on the Sabertooth 2x32. Connect the USB cable to a PC, tablet or single board computer. Even though the signal inputs are not required for control in most USB Modes, they are read as analog inputs. This can be used as a general-purpose analog to digital converter for your system.

USB Mode DIP Switch Settings



DIP switch 4 in the ON position selects USB mode.

USB control uses the same commands as the serial modes. A Sabertooth 2x32 will appear to the operating system both as a serial port and as an HID device. Usually, the serial port is how you will send and receive commands from the Sabertooth. Dimension Engineering supplies an open-source C# library, as well as example code for other programming languages and platforms. For a description of the serial protocol and commands, see the Serial Mode section. USB is the easiest way to interface to PCs or advanced microcontrollers like a Raspberry Pi. Additionally, USB handles its own addressing, collision detection, connection and disconnection detection and checksums, so data corruption is less of a worry.

DIP switch 5 enables or disables USB serial converter mode.

In serial converter mode, you can use the USB port to both send commands to the Sabertooth 2x32 and also relay commands to other Dimension Engineering devices. In this mode, the host Sabertooth 2x32 becomes Packet Serial address 135, to be out of the way of the other devices.

The most common use of this mode is to control a Kangaroo x2 motion control module connected to the Sabertooth 2x32. Closed loop motion commands are sent to the Kangaroo x2. Closed loop positions and velocities are read from the Kangaroo x2 through the Sabertooth. Plug the Kangaroo x2 into the main terminal of the Sabertooth as shown in the Kangaroo X2 manual. Set the DIP switch settings on the Sabertooth to USB Serial Converter mode. Connect a wire from the A1 terminal of the Sabertooth 2x32 to the S2 terminal of the Kangaroo x2. Connect a second wire from the A2 terminal of the Sabertooth 2x32 to the S1 terminal of the Kangaroo x2.

Serial Converter mode can also be used to control additional Sabertooth or SyRen motor controllers. These should be set for packet serial mode, with appropriate packet serial addresses. Connect the A2

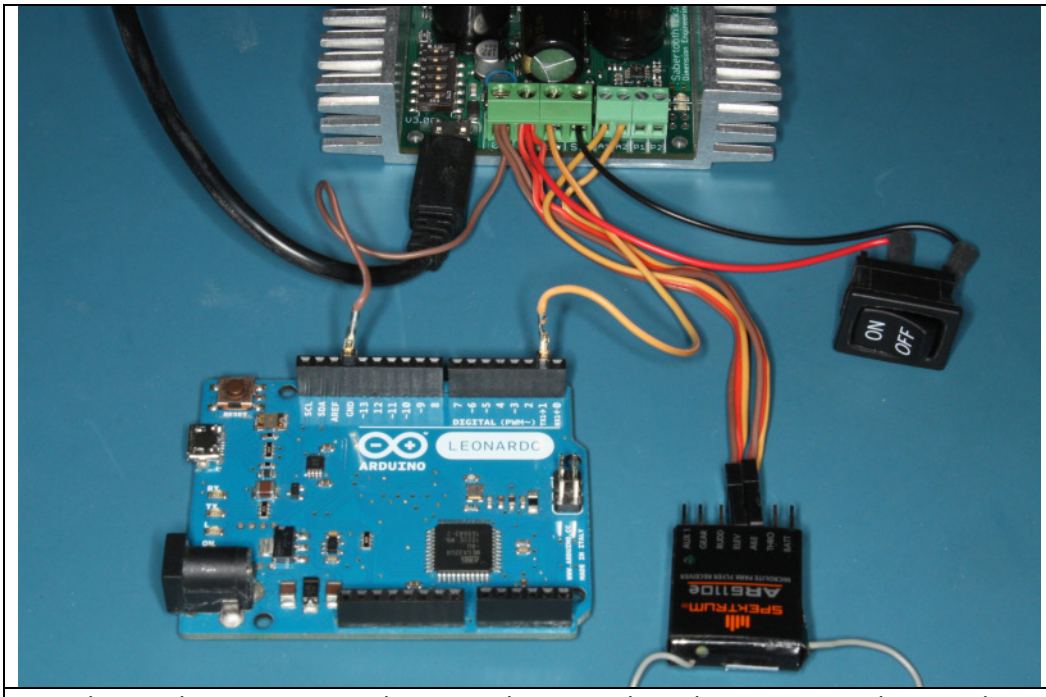
terminal of the Sabertooth 2x32 to the S1 terminal of the additional SyRen or Sabertooth drivers. Connect the A1 terminal to the S2 terminals to enable reading back data, if the device supports it.

DIP Switch 6 enables or disables emergency stops.

If DIP switch 6 is in the OFF position, emergency stops are enabled. This emergency stop is active low and internally pulled down. To enable the M1 output, connect the A1 terminal to 5V. To enable the M2 output, connect the A2 terminal to 5V. If these connections are broken and emergency stops are enabled, the motors will stop immediately. This might be used for a safety E-stop in a machine, to only allow motion while a dead man's switch is held, or to detect a disconnected control cable. Emergency stops provide a second way to shut down the machine if the host computer crashes or locks up, as well as reacting more quickly than a PC might. Hardware emergency stops are required for some safety regulations. Emergency stop is not available when using serial converter mode.

User Mode

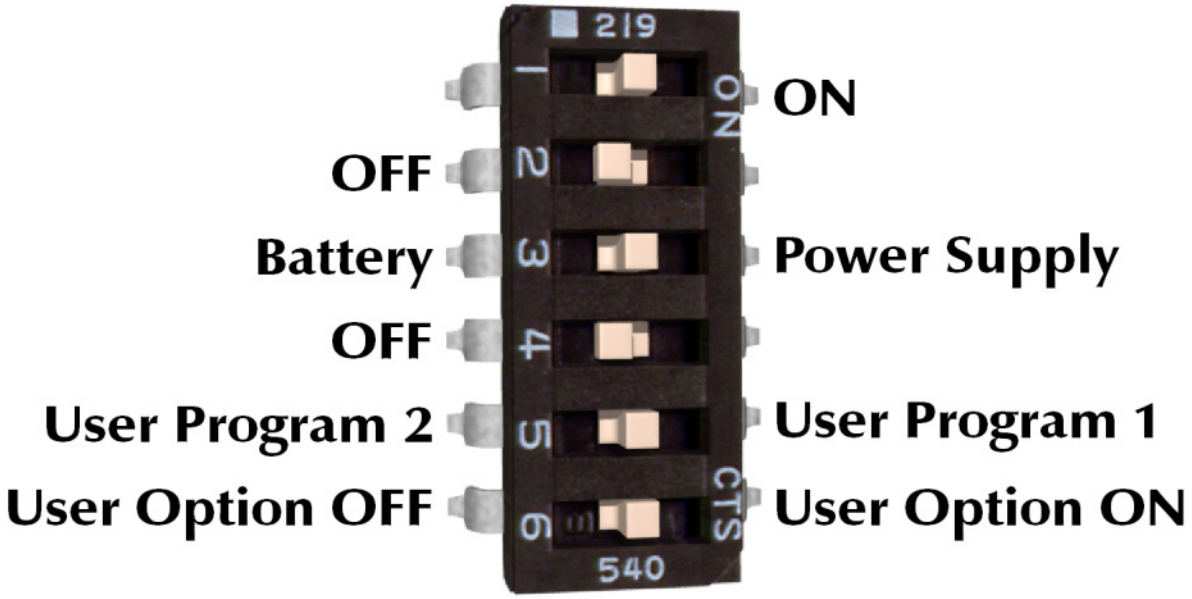
User modes are custom operating modes that are created inside DEScribe. They can combine input types and functions to create a Sabertooth tailored specifically for an application. This can cut down on the number of components needed for a product and enable more sophisticated functionality.



A Sabertooth 2x32 connected to a serial input and a radio receiver, with a switch to select the controlling device and USB monitoring.

In User Mode any of the inputs can be set as any of the input types. Each input should be connected according to its own type.

User Mode DIP Switch Settings



DIP switch 4 in the OFF position selects User Mode.

User modes are custom modes that can be reprogrammed with the DEScribe PC software.

DIP switch 5 selects between User Mode program 1 and User Mode program 2.

Each Sabertooth 2x32 can have two separate User Mode programs. If DIP switch 5 is in the ON position, User Mode program 1 is selected. If DIP switch 5 is in the OFF position, User Mode program 2 is selected.

DIP switch 6 is used inside the User Mode programs.

DIP switch 6 is left available to the creator of the User Mode program, to allow for in-program settings. For example, in a program that controls a conveyor belt, this might be used to select whether the belt runs to the right or to the left.

Default User mode programs

Sabertooth 2x32 ships with two default user mode programs. This is for additional out-of-the-box functionality, and to give programming examples that can be modified.

Default program 1: Serial Autopilot with R/C takeover

Default program 1 would be used with both a microcontroller and an R/C transmitter. Often, you want a robot to run autonomously, except for having the option of manual control for parking, maneuvering in tight spaces, or if the autopilot malfunctions.

S1 is set up as a serial receiver and is connected to an Arduino or similar.

S2 is set up as an R/C input with fixed calibration.

The R/C signal connected to S2 controls whether the motor driver responds to the serial input or the R/C inputs. This is connected to channel 5 on the transmitter, which is a two position switch. In one position, the Sabertooth will respond to the serial commands, in the other it will respond to the R/C signals.

A1 and A2 are set up as R/C inputs with automatic calibration and differential drive mixing.

A1 is the forward/back input and is connected to the elevator channel of the transmitter.

A2 is the right/left input and is connected to the aileron channel of the transmitter.

Default program 2: Analog with remote Indicator LEDs and current display

Default program 2 would be used when it is necessary to have the error indicator on a control panel. Often when building machinery the Sabertooth 2x32 is tucked deep inside a control cabinet, so the onboard LEDs are not visible. Also, for some applications it is useful to display the load current, to prevent loading a mechanism. The power outputs P1 and P2 are used to drive analog panel meters to display load current on each motor.

In this example, the M1 motor is set up bi-direction, with a potentiometer connected to S1 controlling speed and direction.

The M2 motor is set up for separate speed and direction controls. A potentiometer going to A1 controls the output speed, and a switch going between the 5V output and A2 controls the direction.

S2 is set to Indicator output and mirrors the error LED. If the error LED illuminates, an LED connected between S2 and 0V will also illuminate.

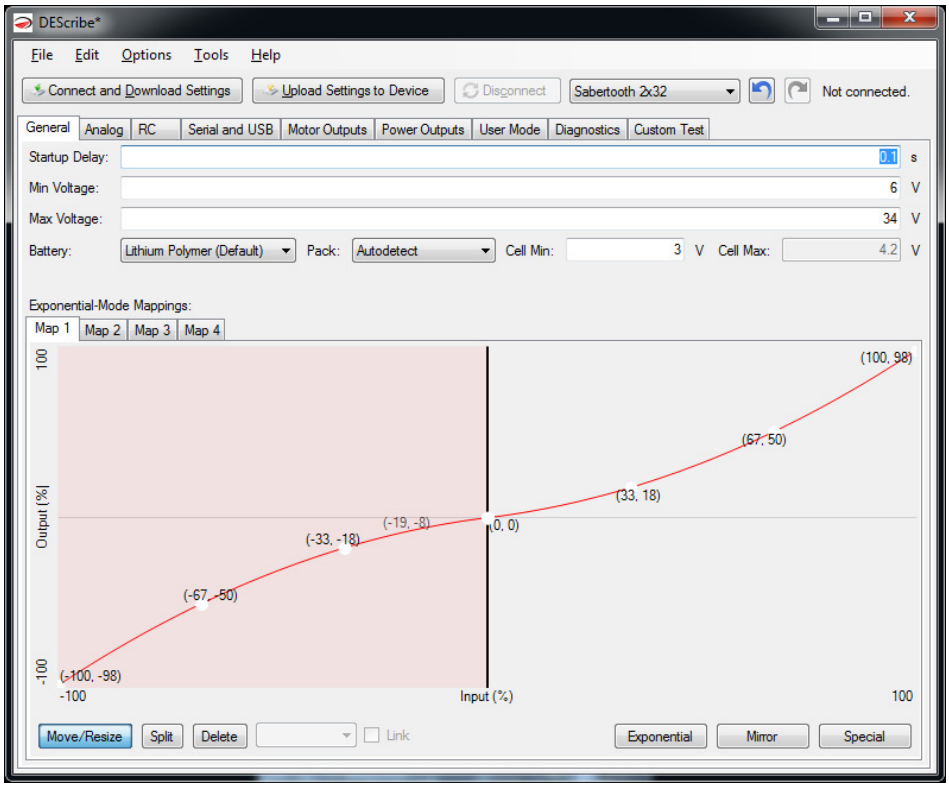
P1 is connected with a 1k pull-up resistor to the 5V output and set as a controllable output, driven by the internal M1 current reading. A 32 amp current reading will drive a full scale 5 volt output. Connect a 5V analog meter between P1 and 0V.

P2 is connected with a 1k pull-up resistor to the 5V output and set as a controllable output, driven by the internal M2 current reading. A 32 amp current reading will drive a full scale 5 volt output. Connect a 5V analog meter between P2 and 0V.

In addition to the example programs, User Mode equivalents for the other operating modes can also be found in the Examples folder.

DEScribe PC software

DEScribe is Dimension Engineering's PC software for adjusting, modifying, monitoring and updating motor drivers, as well as certain other products. It is compatible with Windows XP and newer operating systems. Third generation motor drivers like Sabertooth 2x32 are highly configurable. DEScribe is necessary to access many of the advanced features of these products. DEScribe is a free download, and can be found on Dimension Engineering's website.



Getting Started

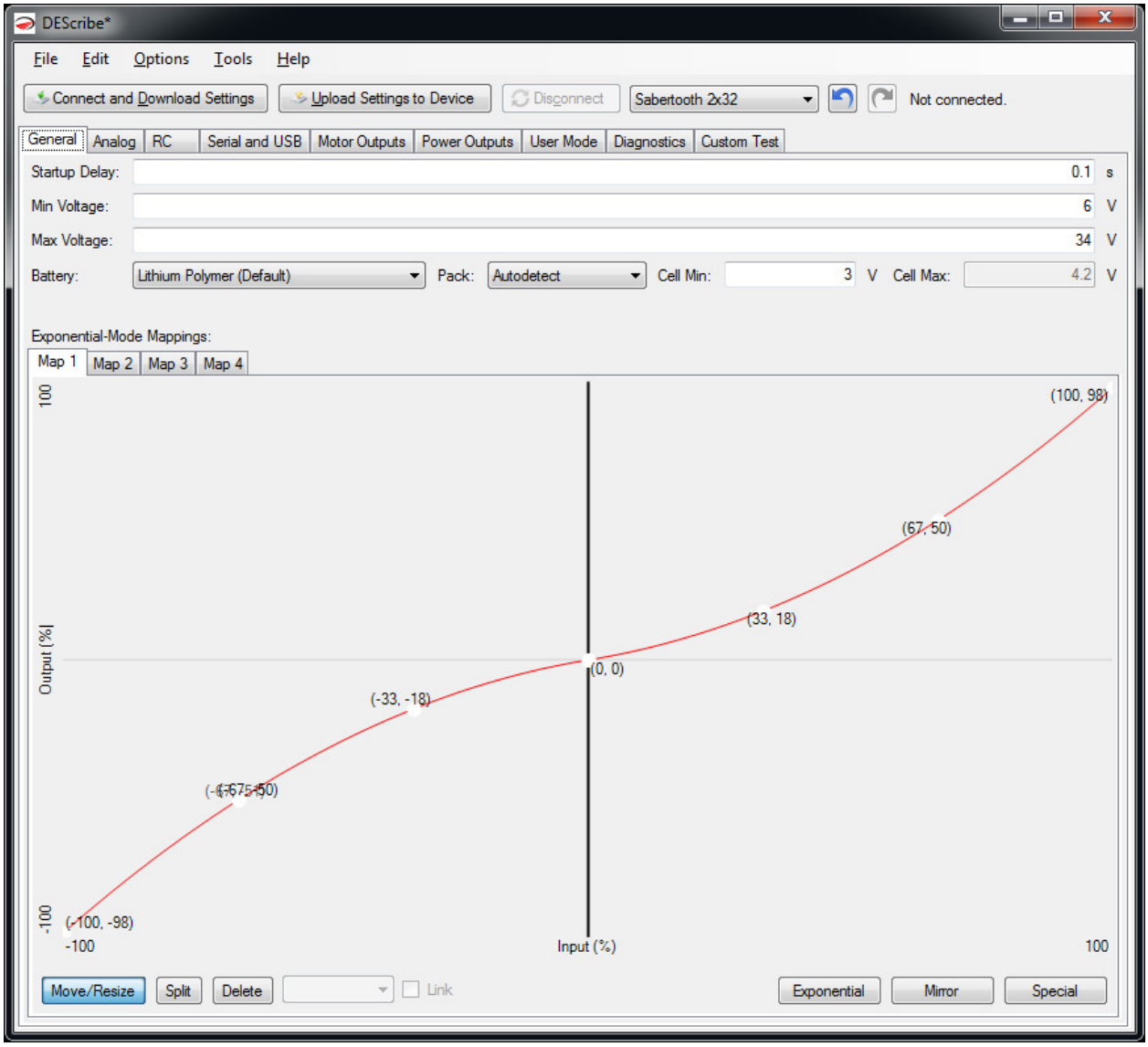
Your Sabertooth 2x32 came with a micro USB cable. Once you have installed DEScribe, connect one side to the Sabertooth 2x32 and the other side to the PC. You do not have to have a battery, motors or inputs connected, but you may if this is more convenient. Without a battery connected, the green Status LED and red Error LED will flash together when connected to USB. This is normal, and is indicating a low battery voltage. You may communicate with your Sabertooth 2x32 over USB in any operating mode, so you do not need to change the DIP switches.

With the Sabertooth 2x32 connected to USB, launch DEScribe. A quick start screen will appear. Click *Connect and Download Settings* to connect to the Sabertooth 2x32. Generally, you will want to start a DEScribe session by connecting to the Sabertooth. If you do not, then changes you already made could be lost. The *Connect* dialog box will appear. Click the button next to the Serial Port label, and select Sabertooth 2x32. Because Sabertooth 2x32 is a native USB device, no other options are necessary. Click

OK. DEScribe will download the current settings. Any time you want to save the modified settings to the Sabertooth, click the *Upload Settings to Device* button.

General tab

The General tab contains settings that affect the Sabertooth 2x32 in all modes and for all outputs.



Startup Delay

Startup Delay sets how long the Sabertooth 2x32 waits after power is applied to begin operation. This delay is to allow host microcontrollers to boot, receivers to start and analog voltages to stabilize.

Min Voltage and Max Voltage

Min Voltage and Max Voltage set the input voltages within which the Sabertooth 2x32 will operate normally. If the input voltage is outside this range, all motor and power outputs will be off.

Battery

Battery selects the type of battery used and the number of cells. It is also possible to change the low voltage cutoff settings here. These settings go into effect when DIP switch 3 is in the OFF position. See the Battery section for more details.

Exponential-Mode Mappings

Sabertooth 2x32 can have up to four maps which modify the ratio of input signal to output signal. These can be used to add a deadband, change the responsiveness of the system, make the Sabertooth only turn the motors one way, or other uses. Each exponential map is defined by a series of curves. Each curve is defined by control points. By clicking and dragging the control points, you can change the shape of the curve. Which exponential map is used for each output is selected with DIP switches and/or settings in that operating mode's tab in DEScribe.

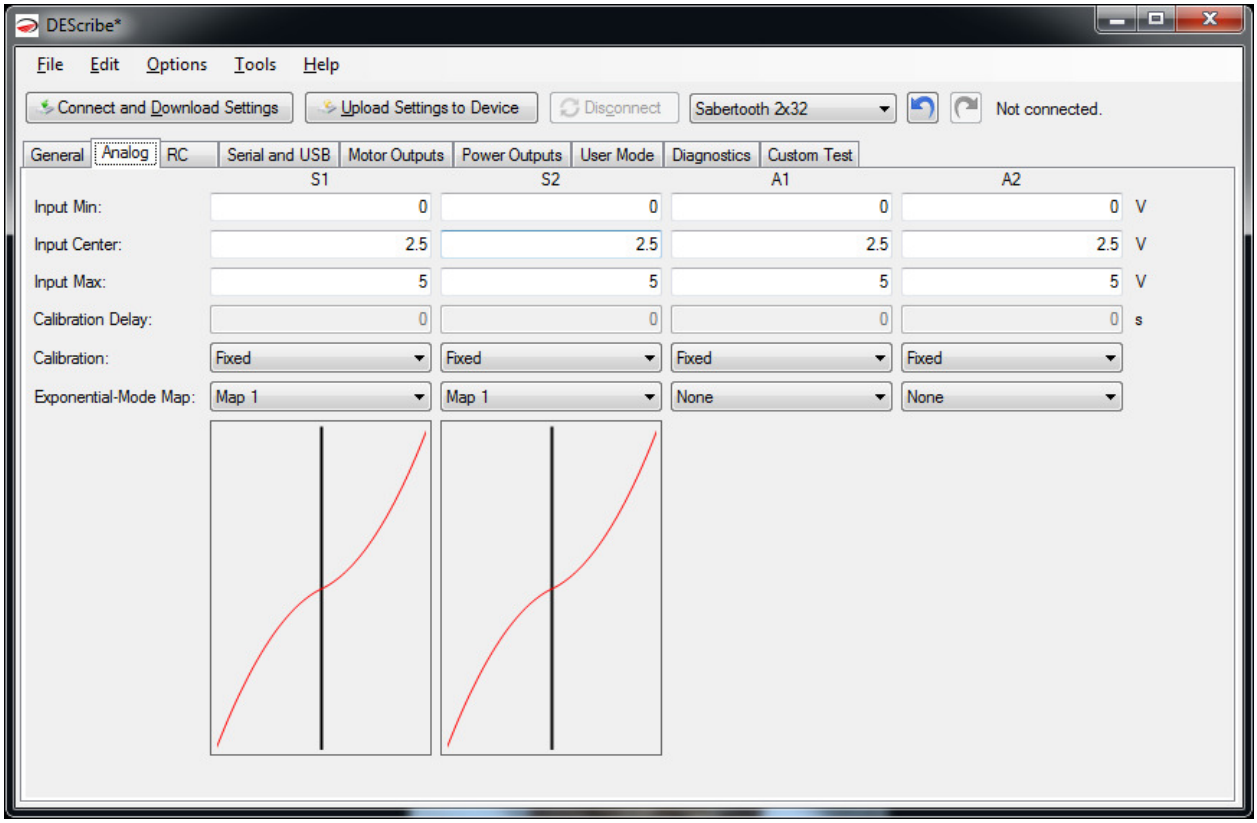
Most of the time, what you want is a certain percentage of exponential and a certain amount of deadband where the motor output is off. These can be handled automatically. Clicking the Exponential button in the bottom right corner of the window will bring up the Exponential Curve window. By dragging the sliders, the currently tabbed exponential map can be modified with little effort.

For more customized maps, you may have to add additional segments. To break a segment into two smaller segments, select the Split tool at the bottom, then click on the curve you wish to divide. The start and end of the curves can be changed by clicking the black vertical divider bars and dragging right or left. To remove a segment, select the Delete tool and click on the segment you wish to remove. To change the curve type, select the segment and change the dropdown from Curve to Linear or Constant. If you have created a custom exponential map, it is often a good idea to save your settings file for re-use later. To save a settings file, click File… Save and create a file name.

There are four maps. By default, Exponential mode uses Map 1 for both channels, so it is usually the only one that needs to be modified. Maps can be modified even if they are not currently being used.

Analog tab

The Analog tab contains settings that are specific to operation in Analog mode. Each of the four analog inputs has the same options. Also, if in a different mode a pin is set for analog input, it is configured here.



Calibration

Depending on what is generating the input signal, different calibration types are most appropriate. Which other settings are active depends on the calibration type selected for the input.

Fixed Calibration

With Fixed calibration, the voltage levels for minimum speed, maximum speed and zero speed are pre-defined and do not change during operation. This would be used with a potentiometer, a DAC or other inputs that are well known and repeatable. Most of the time in Analog mode you will use Fixed calibration. When using Fixed calibration, you can change the Input Min, Input Max and Input Center values. For example, if you are controlling the Sabertooth from a DAC which has a maximum output voltage of 3.0 volts, you would want to change Input Max to 3.0 and Input Center to 1.5.

Automatic Calibration

With Automatic calibration, the voltage levels are re-learned every time the motor driver is powered up. This should only be used with input signals that self-center, such as joysticks. When powered up, after a

delay (the Calibration Delay setting) the Sabertooth will take whatever voltage level is presented at the input and use that as its zero speed setting. It will then constantly look for the largest and smallest voltage that it has ever seen, and use these values as full speed forward and reverse. Automatic calibration is very helpful for signals such as low-cost joysticks that have significant variability in their center and end values. The downside to using Automatic calibration is that if the joystick is not in its center position when the Sabertooth is powered up, or after it loses power for any reason, control will be incorrect.

Joystick Calibration

With Joystick calibration, only the stopped position is re-learned each time the Sabertooth powers up. You can manually change the Input Min and Input Max values. This can prevent slow speed creeping if the joystick does not return exactly to center due to mechanical hysteresis or sticking. Higher quality joysticks will often have a defined output voltage range, usually 0V to 5V, so automatic calibration of the endpoints is unnecessary.

Saved Calibration

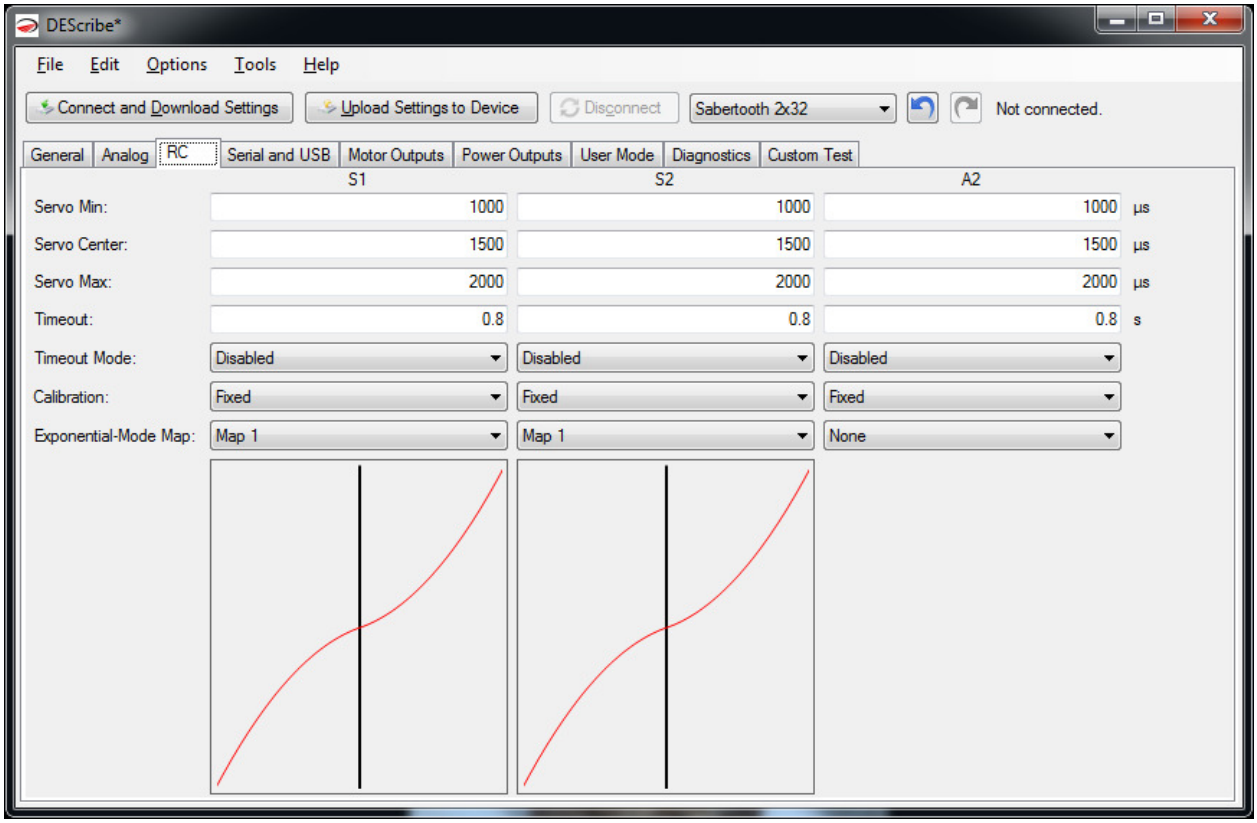
Saved calibration acts exactly like Automatic calibration the first time the driver is powered up after selecting this option, but it then reverts to fixed calibration using the settings it has discovered from that point forward. This is primarily used in production settings, to allow the Sabertooth to learn the specifics of the joystick or control it is mated to, without having the limitations of automatic calibration.

Exponential-Mode Map

To use different maps, select one of the four maps using the drop-down list. A graphical representation of the selected map will appear below the channel. Please note that these maps are only used in Analog mode if DIP switch 5 is set to the OFF position.

RC tab

The R/C tab contains settings that are specific to operation in R/C mode. Because every manufacturer uses slightly different timing for their R/C signals, it is not unusual to have to change the settings to work with a specific transmitter and receiver combination, or a specific microcontroller.



Calibration

Depending on what is generating the input signal, different calibration types are most appropriate. Which other settings are active depends on the calibration type selected for the input. In R/C mode, these settings will only take effect if DIP switch 6 is set to the OFF position. With DIP switch 6 in the ON position, the inputs will always be set for Automatic Calibration.

Fixed Calibration

With Fixed calibration, the R/C pulses for minimum speed, maximum speed and zero speed are pre-defined and do not change during operation. If you are using a microcontroller or other well-defined signal source, you should always use fixed calibration. When using fixed calibration, you can change the Servo Min, Servo Max and Servo Center values. While an Arduino can be programmed to send 1000 to 2000us pulses, a Spektrum DX7 transmitter typically sends pulses ranging from 1200 to 1800us. The Sabertooth 2x32 can be used to read the servo timing of an unknown source by using the Diagnostics tab.

Automatic Calibration

With Automatic calibration, the signal pulses are re-learned every time the motor driver is powered up. This is the normal mode to use with spring-return joystick channels on an R/C transmitter. When powered up, after a delay Sabertooth will take the servo timing and use that as its zero speed setting. It is important to have the joystick centered and the transmitter on when the Sabertooth is powered up. The Sabertooth 2x32 will then constantly look for the longest and shortest signals that it has ever seen, and use these values as full speed forward and reverse. Automatic calibration allows the Sabertooth 2x32 to get acceptable performance from a wide variety of transmitters and receivers. The downside to using Automatic calibration is that if the joystick is not in its center position when the Sabertooth is powered up, or after it loses power for any reason, control will be incorrect.

Joystick Calibration

With Joystick calibration, only the stopped position is re-learned each time the Sabertooth powers up. This can prevent slow speed creeping if the joystick does not return exactly to center due to mechanical hysteresis or sticking. Joystick calibration is rarely used in R/C mode.

Saved Calibration

Saved calibration acts exactly like Automatic calibration the first time the driver is powered up after selecting this option, but it then reverts to Saved calibration using the settings it has discovered from that point forward. This is primarily used in production settings, to allow the Sabertooth to learn the specifics of the transmitter it is mated to, without the limitations of Automatic calibration.

Timeout

This setting controls how long after the receiver stops sending data the Sabertooth 2x32 will shut off the motor outputs.

A note on certain receivers

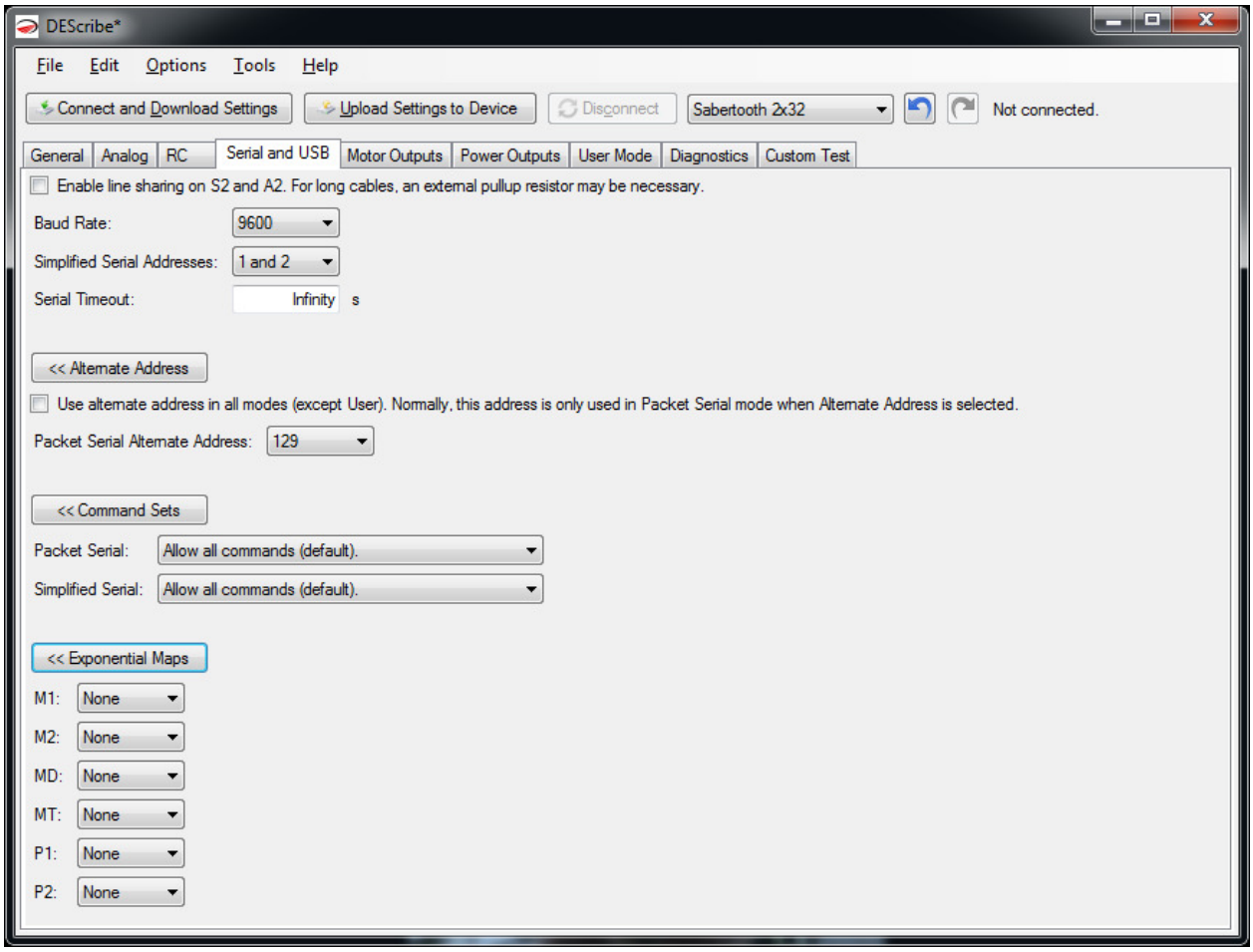
Some receivers will send a “bind” or “failsafe” position before they have gotten any data from the transmitter. The Sabertooth 2x32 will center on the first signal it sees, so if the trim position of the transmitter does not match the bind or failsafe position, your robot may creep forwards or backwards. Typically this can be remedied by re-binding the receiver to the transmitter. Some receivers will continue to put out the last good signal even if communication with the transmitter is lost. This is undesirable with a Sabertooth, because your receiver could continue to send a “drive forward” signal and the rover could run away. Consult your radio’s manual to find out how to disable this “feature.”

Exponential-Mode Map

To use different maps, select one of the four maps using the drop-down list. A graphical representation of the selected map will appear below the channel. Please note that these maps are only used in R/C mode if DIP switch 5 is set to the OFF position.

Serial and USB tab

The Serial and USB tab contains settings that affect both Serial mode and USB Mode. USB and Serial mode use the same communication protocols.



Enable Line Sharing

This setting allows multiple Sabertooth motor drivers to be used with the same serial port. Instead of always being driven, the transmit lines back to the host processor are only driven while data is being sent.

Baud Rate

This changes the baud rate for the serial modes. Options are 2400, 9600, 19200, 38400 and 115200 baud. The default is 9600. The baud rate setting of the Sabertooth must match the setting of the host microcontroller.

Simplified Serial Addresses

Because Plain Text Serial does not have an address byte, this is used to select which motor driver is being addressed. For example, a first Sabertooth could be set to 1 and 2, and a second Sabertooth set to use addresses 3 and 4. In this configuration, a command of M1: 500 will command only the first

Sabertooth's M1 output to turn. A command of M3: 500 would control the M1 output of the second Sabertooth.

Serial Timeout

If the Sabertooth has not gotten a new command within this amount of time, it will assume the host program has locked up and shut down the motors. Sending a new valid command will cause the motors to restart. A setting of **Infinity** means the motors will run at the last commanded speed forever.

Alternate Address

This option is used to set the packet serial address of the Sabertooth. This is used in conjunction with DIP switch 5 in Serial mode. You can also force this address in all modes with the checkbox.

Command sets

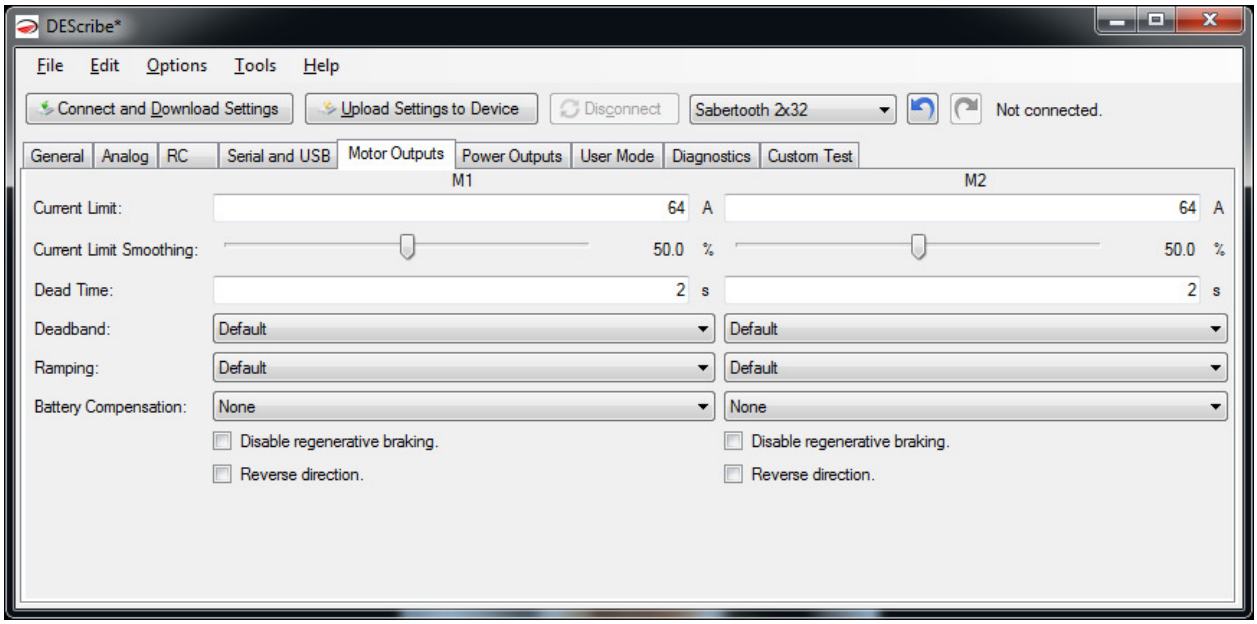
These options tell the Sabertooth to use or ignore certain command sets. These would typically be set for robustness if you are using CRC protected packets.

Exponential

These cause selected serial commands to be processed through the exponential maps. This can reduce the load on the host processor.

Motor Outputs tab

These settings affect the operation of the motor outputs in all modes.



Current Limit

This setting sets the maximum current for the motor output. If the system tries to draw more current than the setting, the output duty cycle will be reduced. For Sabertooth 2x32, each motor output can be limited individually in a range between 0A and 64A. This is a soft current limit, and will slow acceleration but allow motor motion to continue. There is also a fixed hard current limit above approximately 70 amps which protects the Sabertooth from a shorted motor.

Current Limit Smoothing

This setting changes how quickly the current limit responds. More smoothing will cause the Sabertooth to ignore small overcurrent spikes for smoother operation.

Dead time

This is the length of time that the motor will stay in a regenerative braking mode after a stop command is sent. After this delay, if the motor is still being commanded to stop, the motor output will shut down to conserve power. Any command that would move the motors will immediately wake the Sabertooth from this shutdown state.

Deadband

This is the range of input signals that the Sabertooth considers to be a zero speed input. If the setting is left at **Default** the setting varies depending on the operating mode. For example, in Analog mode the default deadband is from 2.375 to 2.625 volts. If the setting is changed to **Custom**, text boxes will appear. Enter your desired deadband in these fields and this deadband setting will be used in all modes.

Ramping

This setting is the amount of time the controller will take to go from full reverse to full forward. If the setting is left as **Default**, then the value depends on the operating mode. The default in Analog and R/C mode is for the ramping to be controlled by an analog signal sent to A1. The default in Serial and USB mode is for the ramp timing to be controlled by a serial command. Changing the selection to **Custom** allows for a pre-defined, fixed ramp speed. The Ramping time is the time it would take to go from a full reverse command to a full forward command.

Battery Compensation

This is a new feature in the third generation Sabertooth motor drivers. As batteries discharge, the voltage they supply declines. A 12v lead acid battery will supply 13.5 volts fully charged, and 9 volts fully discharged. What this means is your system will run faster on a full battery than a depleted one. Sabertooth 2x32 is always measuring the input voltage, so it is able to compensate for this change, resulting in more consistent operation. This is especially helpful with closed loop control.

Battery compensation can only decrease the output, so the Compensated Voltage must be less than or equal to the minimum expected input voltage. Battery compensation is less efficient than using the proper input voltage directly. A Sabertooth with a 24V input compensated to 12v will run hotter than if it was running from a 12V input. The motor will also run less efficiently.

Fixed Battery Compensation

When fixed battery compensation is selected, the duty cycle will be scaled such that full speed forward or reverse puts out the Compensated Voltage, regardless of the input.

Automatic Battery Compensation

Automatic compensation uses the battery cutoff voltage defined in the General tab as the compensated voltage. For example, a 3s lithium polymer battery with a cutoff of 9V will cause the system to respond as if it is always being supplied 9 volts.

Disable Regenerative Braking

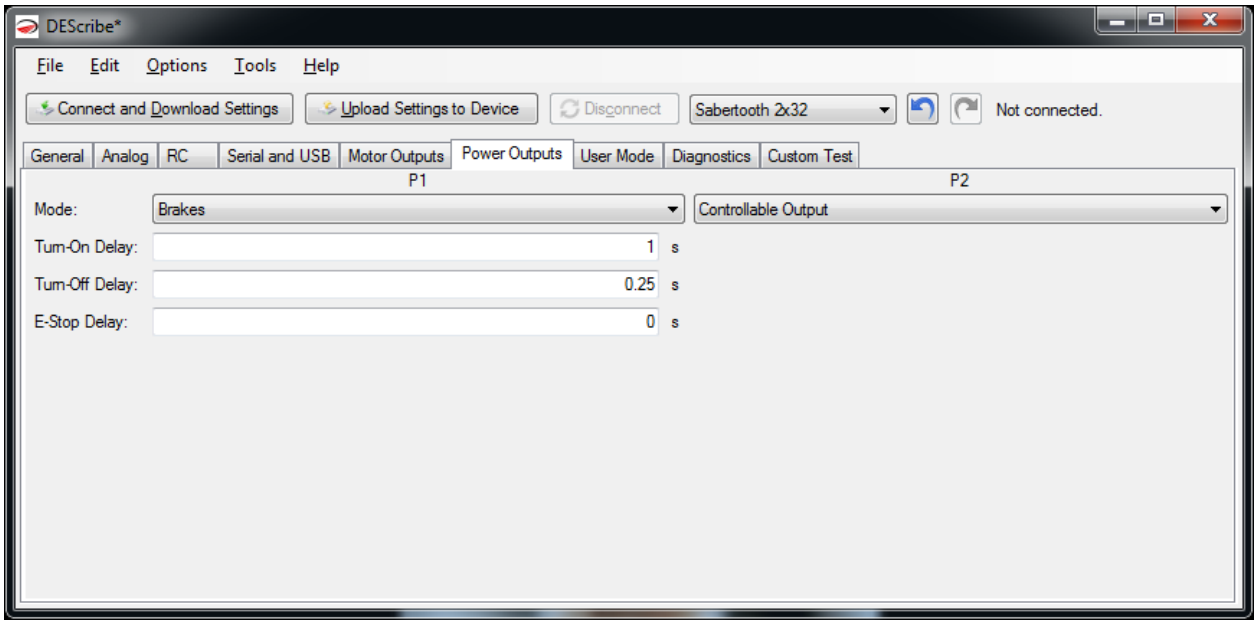
This option disables regenerative braking. This is typically used to drive plating tanks, Peltiers or other loads that are not motors. A motor driven this way may run much less efficiently. Motors driven this way will also stop less rapidly. If you are running motors, but the system runs from a power supply, it is usually better to leave regeneration enabled and use the power outputs P1 or P2 as regenerative voltage clamps.

Reverse Direction

This check box will reverse the motor output for this channel. Now instead of assembling a complicated project only to find one motor is turning backwards and having to change the wiring, you can solve the problem in software.

Power Outputs tab

The power outputs tab controls the behavior of the power outputs P1 and P2.



Mode

The power outputs must be put in the correct mode in order to have the expected function.

Voltage Clamp

In this mode, the power output P1 or P2 will turn on if the input voltage rises. This rise is due to regenerated energy being fed back into a power supply that is not able to accept it. Connecting a power resistor between the power output and the positive supply voltage and using this mode allows for operation from a power supply. **Max Voltage** is the voltage above which the output will turn on. Automatic compares the present input voltage to the average input voltage, and works well in most cases. Moving the slider allows for a custom turn-on voltage.

Brakes

In this mode, the power outputs are used to operate electromagnetic brakes. P1 corresponds to motor 1 and P2 corresponds to motor 2. When the motor stops, the brakes activate after the **Turn-On Delay**. When the motor is commanded to move again, the brakes deactivate immediately, but the motor does not turn until the **Turn-Off Delay** has elapsed. If an emergency stop is commanded, the brakes will activate after the **E-Stop delay** has elapsed.

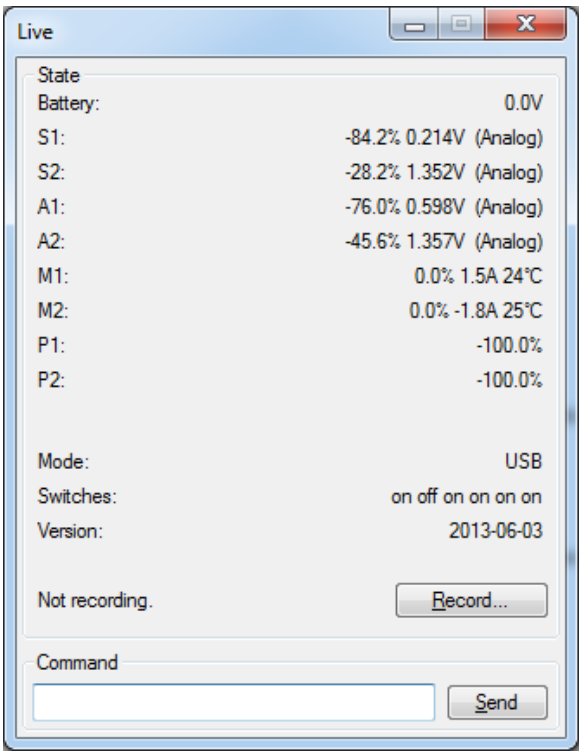
Brakes mode can also be used to drive the field of a shunt or compound wound motor. To do this, connect one side of the field to B+ and the other side of the field to the P1 or P2 power output. Turn all the timings to 0 seconds.

Controllable Output

In this mode the power output is directly controllable. A control setting of 2047 corresponds to full power and a control setting of -2047 corresponds to zero power.

Diagnostics tab

DEScribe can be used to monitor the inputs and outputs of the Sabertooth 2x32 during operation. Click on the **Show Inputs/Outputs** button to bring up the **Live** monitor window.



Battery shows the current input voltage.

The signal inputs **S1**, **S2**, **A1** and **A2** show both the raw input value and the processed percentage that input value corresponds to. This is useful for setting or debugging input calibration.

The motor outputs **M1** and **M2** show the output duty cycle, output current and motor driver temperature.

The power outputs **P1** and **P2** show their current output percentage. Remember that -100% corresponds to an off state, and 100% is full on.

Mode will read the DIP switches and display the current operating mode and options.

Switches reads the DIP switches, which is useful to diagnose if something is set incorrectly.

Version shows which firmware revision is currently loaded onto the Sabertooth 2x32.

Recording will save the monitor values into a .csv file, which can be opened as a spreadsheet. This is useful to catch transient errors or collect long-term data automatically.

If you are in USB mode, **Command** can be used to enter Plain Text Serial commands for debugging purposes. Type the command into the text box, then click the **Send** button to send the command.

Tools menu

The Tools menu at the top of the window contains an assortment of tools to make developing with your Sabertooth 2x32 easier.

Calculator is a basic calculator program.

Serial Terminal is automatically connected to the Sabertooth 2x32. It can be used to send serial commands and read responses.

System Information is used to determine information about your operating system and PC. This is useful if there are compatibility issues with either the Sabertooth 2x32 or DEScribe

Update Firmware is used to update the firmware on your Sabertooth 2x32. This will allow new features to be added or bugs to be fixed without having to send it back to Dimension Engineering.

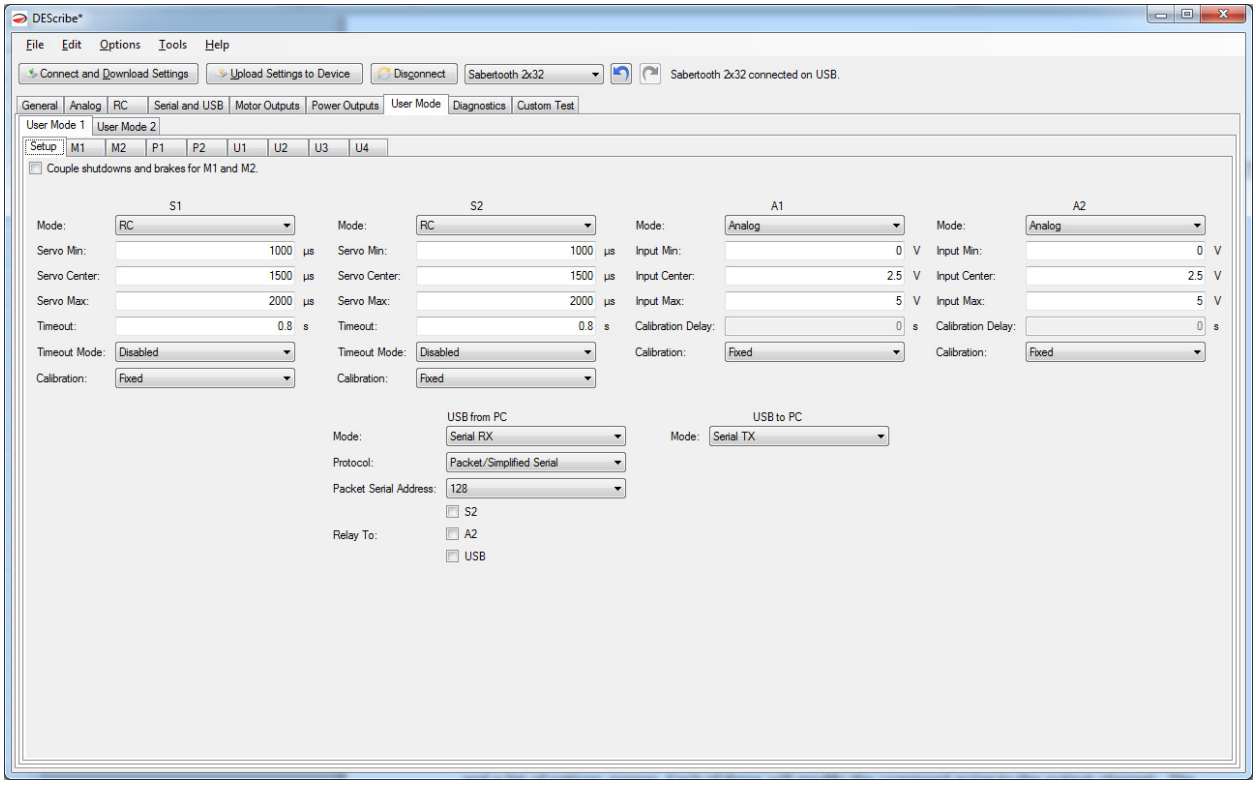
Encoder Calculator is used with quadrature encoders to calculate distance travelled per encoder count. This is most useful with the Kangaroo x2 motion control add-on.

User Mode

A major advance in the third generation Sabertooth motor drivers is the ability to have user-defined operating modes. These are set up using a graphical scripting language in the DEScribe PC software and uploaded over USB. In many cases, this removes the need for an additional microcontroller entirely. For example, you could have main control be from a PC over USB, with a radio control override, or an analog mode that only operates when the A2 input is between 2.2 and 2.5 volts.

Setup tab

To create a custom mode, the first thing that must be done is set up the inputs. On the Setup tab, select how you would like each of the six inputs configured. If you will not be using that input, you may leave it set as the default.



Signal Inputs

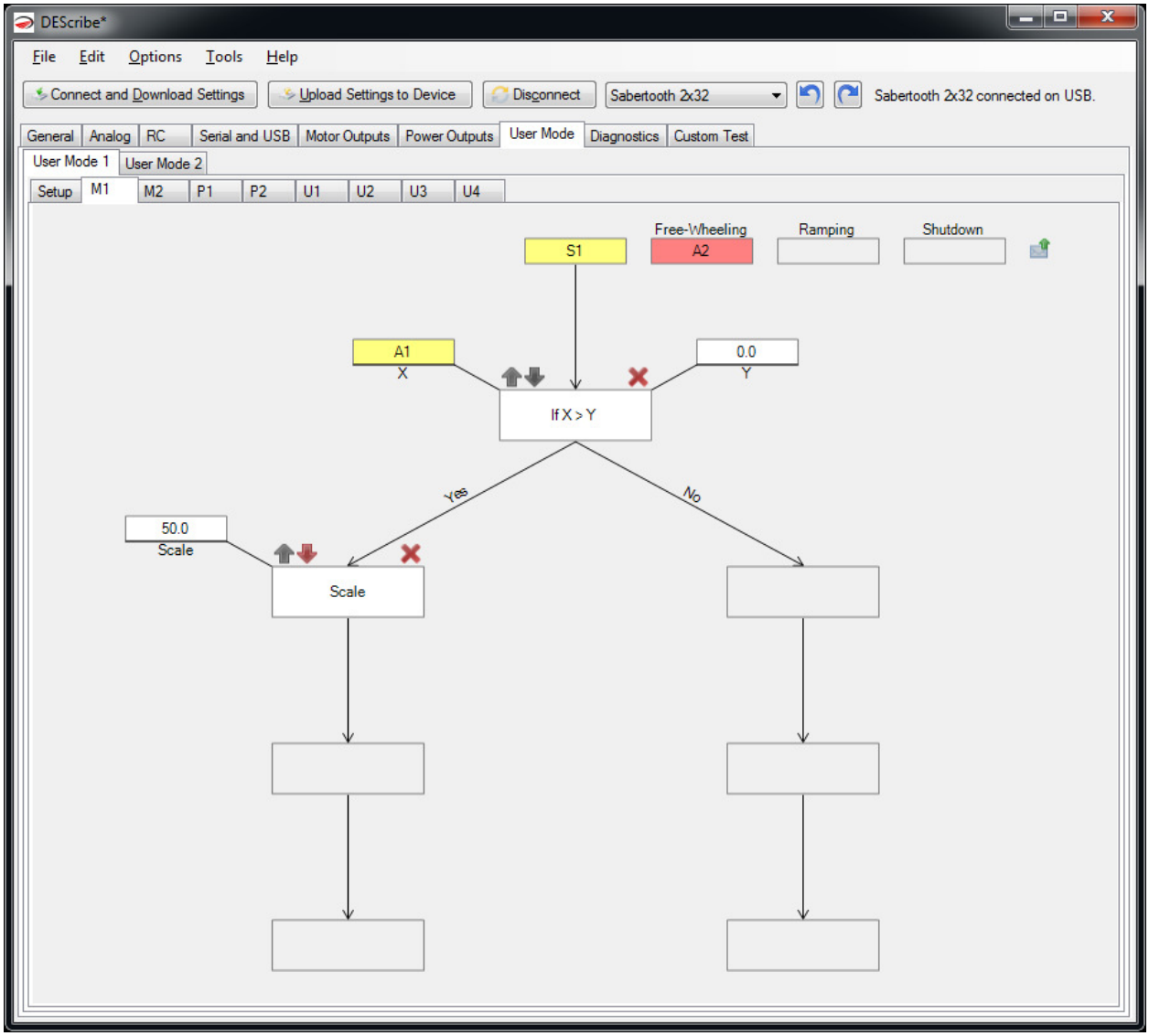
The options for each input will change depending on the Mode selected. These options are the same as for that mode's tab (Analog, R/C or Serial).

Indicator Outputs

Additionally, S2 and A2 can be configured as on/off 20 milliamp logic outputs. This is used to drive external LEDs or signal to a host processor. If the mode is set to Indicator, you must select the source that controls it.

Program tabs

Now that the inputs are set up, a program is created for each output. For example, the M1 tab holds the program that controls the M1 motor output. Each program starts with an initial value read either from an input or from the output of another program. Each program can have up to four commands. Click on the box below the initial input and a list of options appear. Each of these will modify the command going to the output channel. The commands operate in succession, following the arrows. After the last operation, the value is sent to the output.



The program shown would set up an R/C dual rate switch. The program starts with an R/C signal on the S1 input. If the R/C A1 input is positive, then the output is reduced by 50%. If not, then the signal passes through unmodified and directly controls the M1 output.

Program commands

There are ten types of operations available in user mode. Each changes the output or changes program flow.

Each operation takes one of the four block spaces in the program. To add a command, click on a blank box. To remove a block, click the red X above the block. To change or modify a block, click on the block itself and a menu will appear in front of the block. To move a block up or down in the program, click the up or down arrows.

If

The If operation is used to make the Sabertooth do one thing or another, depending on state. For example, you could make it either use USB control or an R/C signal coming into s1, depending on the state of a radio control input into S2.

The If operation takes two additional inputs, which are labeled X and Y. By clicking on the X or Y block, you can select any if the input or internal signals available.

The If operation can perform standard logical comparisons. To change which comparison it is performing, click on the center If block and then select Condition from the list that pops up.

If the comparison is true, the blocks below the Yes arrow will be run. If the comparison is false, the blocks below the No arrow will be run.

Set

The Set operation sets a new value for the output channel, ignoring any values coming into it. This is often used after an If operation. For example, if the temperature is above 10 degrees, operate. Otherwise, **set** the motor outputs to zero.

The Set operation takes one additional input, labeled Value.

Calibrate

The Calibrate operation is similar to the calibration applied to analog and R/C signals. It is used to set a new input to output mapping. Calibrate is computationally intensive and can cause your program to run slowly. For this reason, it is better to calibrate your input signals in the Setup tab and use the Calibrate block only where necessary.

The Calibrate operation takes five additional inputs, labeled Input Min, Input Center, Input Max, Output Min and Output Max.

Flip

The Flip operation reverses the polarity of the signal. For example, if the motor is running forward without the Flip operation, it will run at the same speed backwards with Flip applied.

The Flip operation takes no additional inputs.

In order to save blocks, you can also specify any signal as flipped when it is set. To do this, click on the input box and select Flipped from the menu that pops up.

Map

The Map operation applies one of the custom input-to-output mappings, which are set up in the general tab, to the output channel. This is used for exponential response, or to have a custom range, or to have a user-defined response.

The Map operation takes no additional inputs. To change which of the maps (1 through 4) are being applied, click on the Map block, click Map > at the top, and select the desired map number.

Mix

The Mix operation is used to create differential drive outputs. Mix can also be used to add or subtract signals.

The Mix operation takes two additional inputs. The program flow input (arrow into the top of the block) is the forward/reverse channel and the Mix With input on the left of the block is the right/left channel. The Direction input selects whether this is the left or right wheel.

Scale

The Scale operation is used to scale the motor speed. Typically this would be used for a max speed setting, or reduced sensitivity. It can also be used to multiply one signal by another. You can also make signals bigger by scaling with a fixed value greater than 100%.

The Scale operation takes one additional input, labeled Scale. This is the value by which the original value is scaled. Remember that signals are percentages, so if one signal is scaled by another, the result will end up smaller than the input signals.

Driver Scale

The Driver Scale operation will convert an arbitrary input range into an output of -100% to 100%. This is useful to normalize signals.

The Driver Scale operation takes one input, labeled Source.

Set Bounds

The Set Bounds operation limits a signal to between a user-defined minimum and maximum value. This is different from Scale in that until the bounds are exceeded, the input signal is not modified.

The Set Bounds operation takes two additional inputs, labeled Min and Max.

Set Range

The Set Range operation is used to set a new input or output range for the signal. For example, to make a single direction output, you would use Set Range to change the output range from [-100, 100] to [0, 100].

The Set Range operation takes four additional outputs: Input Min, Input Max, Output Min, and Output Max.

Special considerations

User Scripts U1 through U4

Sometimes while creating a user mode, you will find that you either need more than 4 blocks, or that you need to store an intermediate value that will be used by more than one output. This is what the user scripts U1 through U4 are for. There are a few important points to remember.

Scripts are processed in the following order: U1, U2, U3, U4, M1, M2, P1, P2.

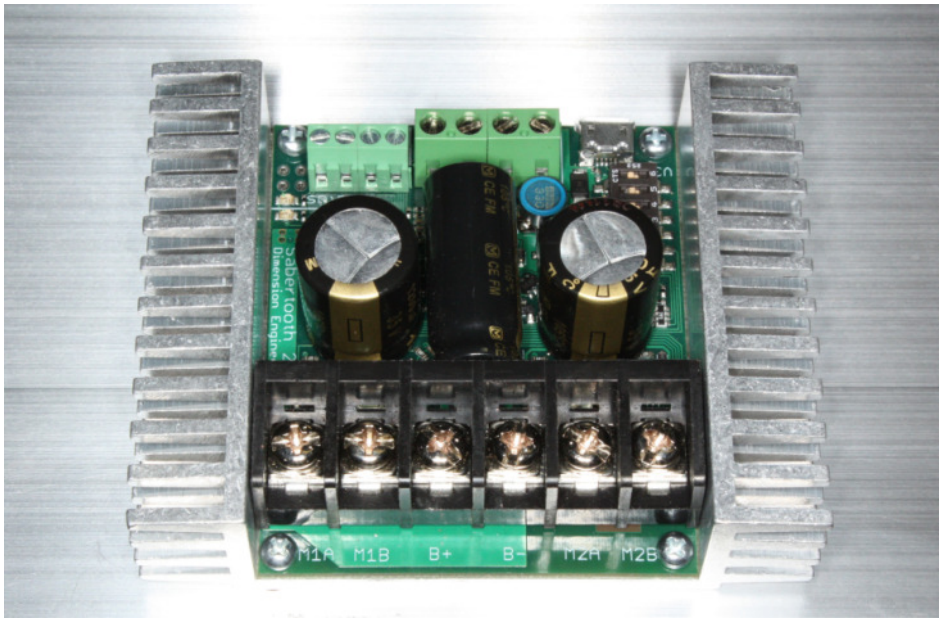
User scripts do not have free-wheeling or shutdown inputs, as those are specific to the motor and/or power outputs. They can have ramp rates.

Output Scripts P1 and P2

It is important to remember that in order to use the P1 and P2 scripts as outputs, the Mode must be set to Controllable Output in the Power Outputs tab of DEScribe. Otherwise the default operating modes such as Voltage Clamp or Brakes take precedence.

Mounting

Sabertooth 2x32 has four mounting holes, which are sized for 4-40 machine screws. 5/8” Phillips machine screws are provided, but other lengths can be used as well. Sabertooth 2x32 has a single-piece machined heat sink. The heat sink features a pin-fin design, so it can be mounted in any orientation with acceptable thermal performance.



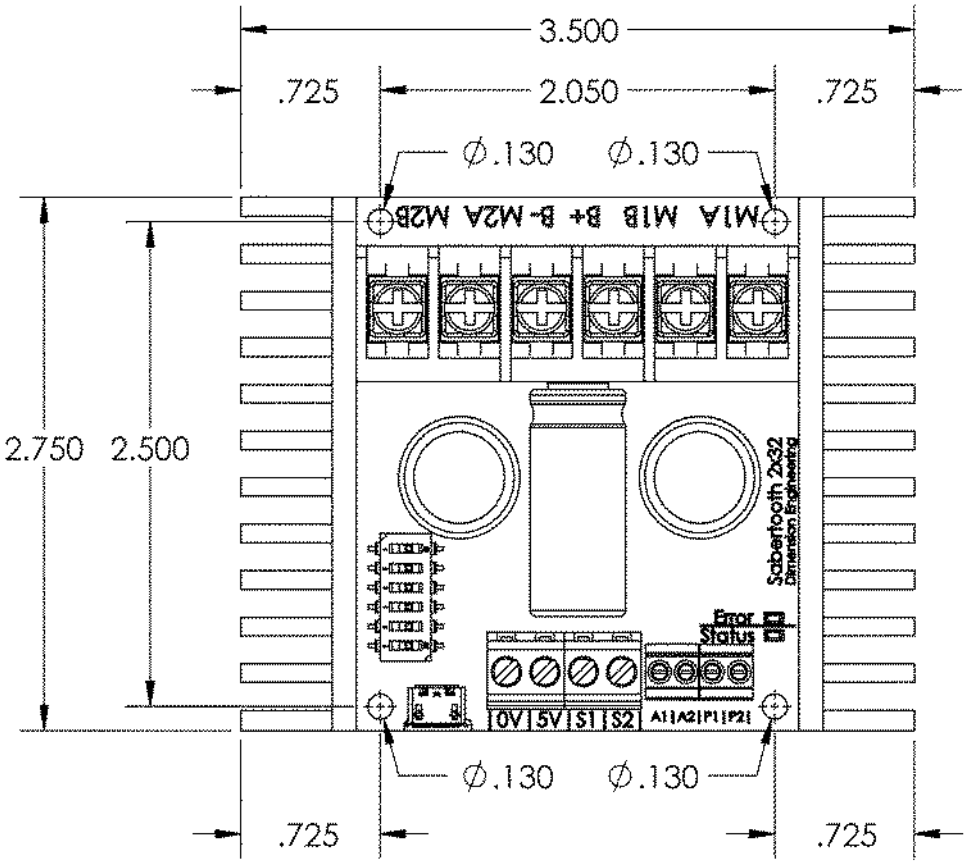
The heat sink is not electrically connected to the electronics of the Sabertooth 2x32. If there is a metal chassis available the best thermal performance will come from mounting the flat bottom of the heat sink directly to the chassis as shown in the photograph above. If the driver is mounted to an insulating material such as wood or plastic, it may be better to use standoffs to allow better cooling.

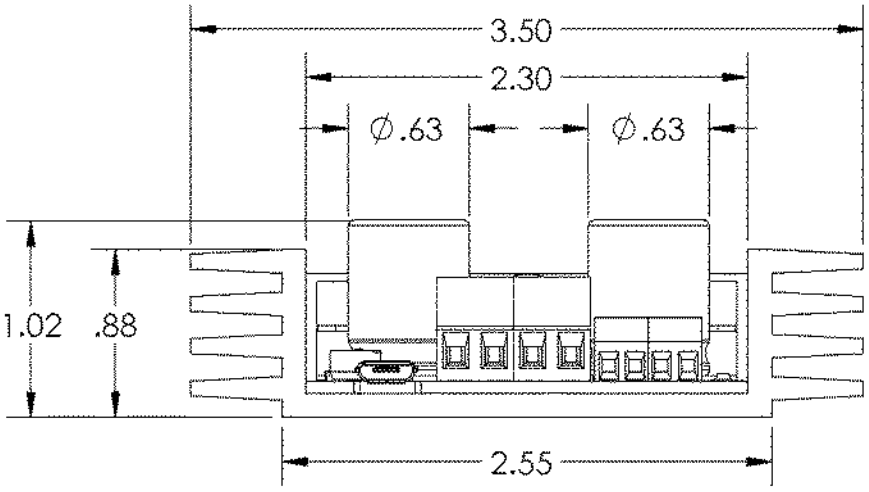
Because Sabertooth 3x32 is a high power component, capable of supplying up to two kilowatts of continuous output power, it will get warm in use. If there is airflow available in the design, having the Sabertooth 2x32 in that airflow will lead to lower board temperatures. At a very minimum, allow some space for air movement between the edges of the heat sink and the enclosure.

There is a printable full size mounting diagram at the end of this manual which can be used to mark the location of the mounting holes. There are also CAD models of the Sabertooth 3x32 available for download on Dimension Engineering’s website.

Drawings

3D models in IGES, STL and Solidworks formats are also available on Dimension Engineering's web site.

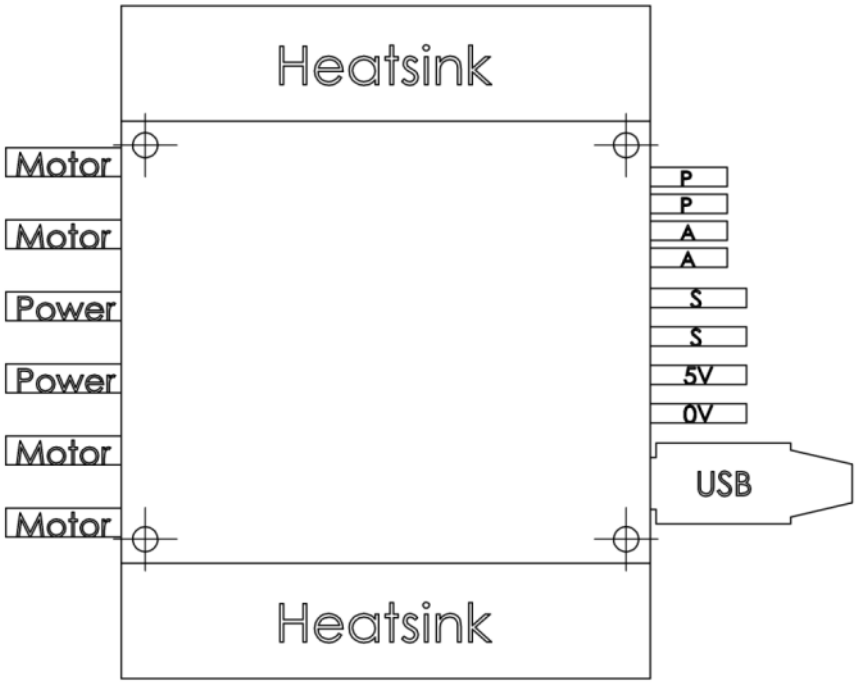




Printable mounting diagram

The following image is the same size as the Sabertooth 2x32. If this page is printed, it can be cut out and used as a template to drill the mounting holes.

Remember when mounting to leave space for the input and output wires, and especially for a micro USB cable, if used. These are included for reference in the mounting diagram.



Universidad de La Laguna

ANEXO III: DATASHEET ENCODERS INCREMENTALES

HEDS-9040/9140
Three Channel Optical Incremental Encoder Modules

Data Sheet

Description

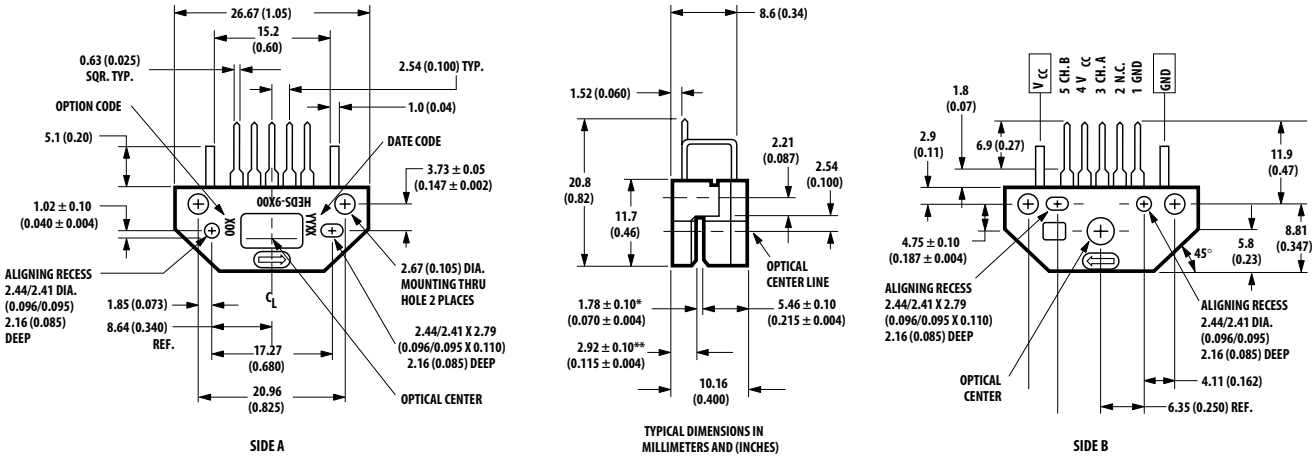
The HEDS-9040 and HEDS-9140 series are three channel optical incremental encoder modules. When used with a codewheel, these low cost modules detect rotary position. Each module consists of a lensed LED source and a detector IC enclosed in a small plastic package. Due to a highly collimated light source and a unique photodetector array, these modules provide the same high performance found in the HEDS-9000/9100 two channel encoder family.

Features

- Two channel quadrature output with index pulse
- Resolution up to 2000 CPR (Counts Per Revolution)
- Low cost
- Easy to mount
- No signal adjustment required
- Small size
- -40°C to 100°C operating temperature
- TTL compatible
- Single 5 V supply

Package Dimensions

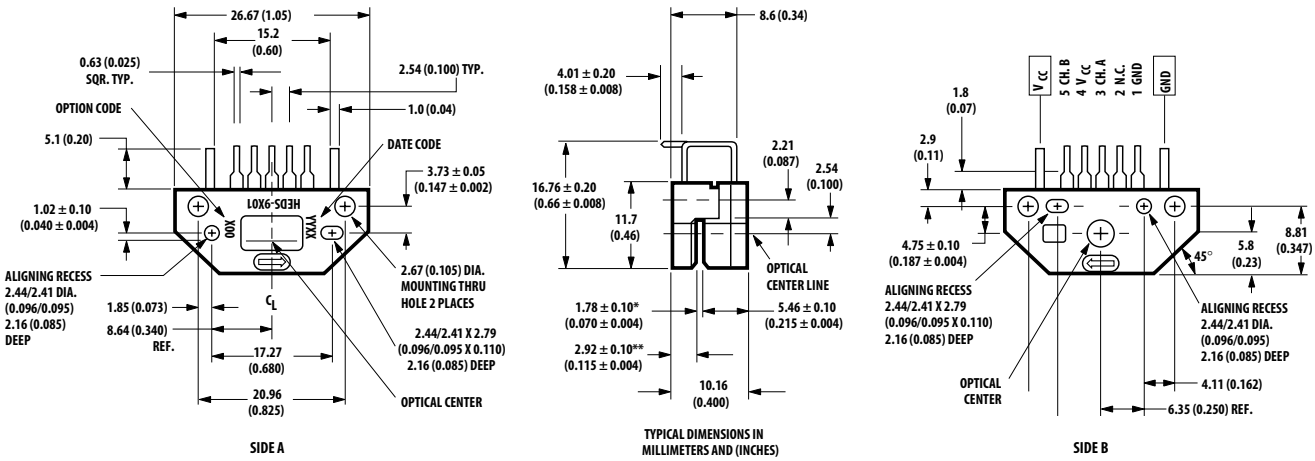
HEDx-9xx0 Option



* GAP MEASUREMENT TO THE SURFACE WINDOW = 1.68 ± 0.10 (0.066 ± 0.004)

** HEIGHT TO THE WINDOW = 3.02 ± 0.10 (0.119 ± 0.004)

HEDx-9xx1 Option



* GAP MEASUREMENT TO THE SURFACE WINDOW = 1.68 ± 0.10 (0.066 ± 0.004)

** HEIGHT TO THE WINDOW = 3.02 ± 0.10 (0.119 ± 0.004)

ESD WARNING: NORMAL HANDLING PRECAUTIONS SHOULD BE TAKEN TO AVOID STATIC DISCHARGE.

The HEDS-9040 and 9140 have two channel quadrature outputs plus a third channel index output. This index output is a 90 electrical degree high true index pulse which is generated once for each full rotation of the codewheel.

The HEDS-9040 is designed for use with a HEDX-614X codewheel which has an optical radius of 23.36 mm (0.920 inch). The HEDS-9140 is designed for use with a HEDx-5x4x codewheel which has an optical radius of 11.00 mm (0.433 inch).

The quadrature signals and the index pulse are accessed through five 0.025 inch square pins located on 0.1 inch centers.

Standard resolutions between 256 and 2000 counts per revolution are available. Consult local Avago sales representatives for other resolutions.

Applications

The HEDS-9040 and 9140 provide sophisticated motion control detection at a low cost, making them ideal for high volume applications. Typical applications include printers, plotters, tape drives, and industrial and factory automation equipment.

Note: Avago Technologies encoders are not recommended for use in safety critical applications. Eg. ABS braking systems, power steering, life support systems and critical care medical equipment. Please contact sales representative if more clarification is needed.

Theory of Operation

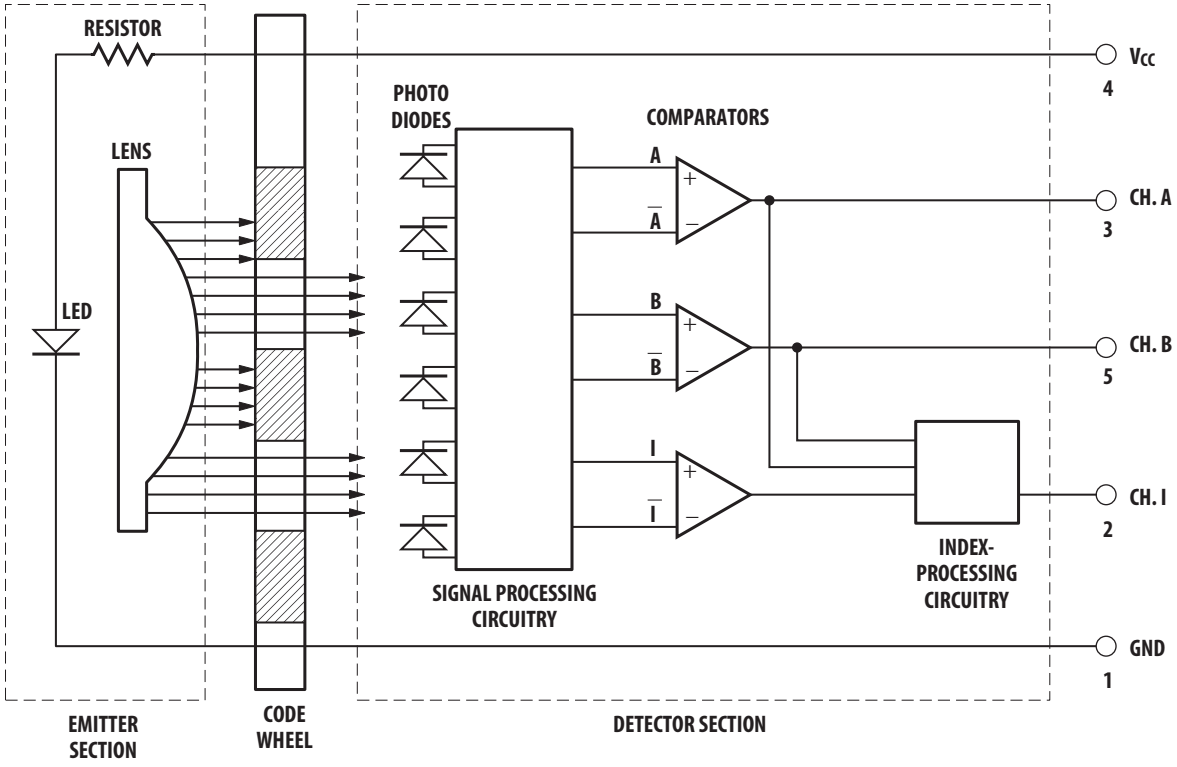
The HEDS-9040 and 9140 are emitter/detector modules. Coupled with a codewheel, these modules translate the rotary motion of a shaft into a three-channel digital output.

As seen in the block diagram, the modules contain a single Light Emitting Diode (LED) as its light source. The light is collimated into a parallel beam by means of a single polycarbonate lens located directly over the LED. Opposite the emitter is the integrated detector circuit. This IC consists of multiple sets of photodetectors and the signal processing circuitry necessary to produce the digital waveforms.

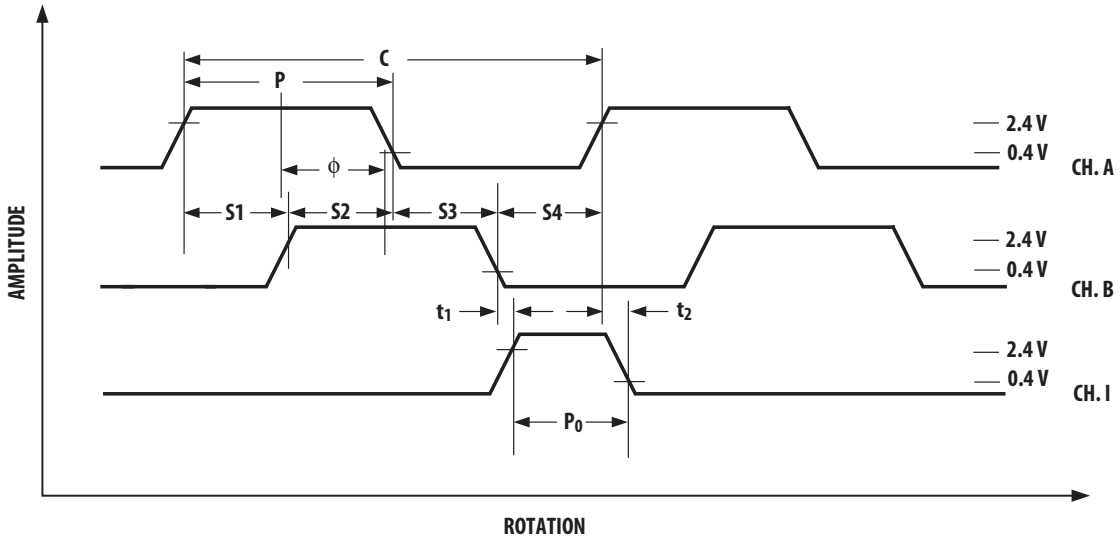
The codewheel rotates between the emitter and detector, causing the light beam to be interrupted by the pattern of spaces and bars on the codewheel. The photodiodes which detect these interruptions are arranged in a pattern that corresponds to the radius and design of the codewheel. These detectors are also spaced such that a light period on one pair of detectors corresponds to a dark period on the adjacent pair of detectors. The photodiode outputs are then fed through the signal processing circuitry resulting in A, $\overline{A}$, B, $\overline{B}$, I and $\overline{I}$. Comparators receive these signals and produce the final outputs for channels A and B. Due to this integrated phasing technique, the digital output of channel A is in quadrature with that of channel B (90 degrees out of phase).

The output of the comparator for I and $\overline{I}$ is sent to the index processing circuitry along with the outputs of channels A and B. The final output of channel I is an index pulse P_O which is generated once for each full rotation of the codewheel. This output P_O is a one state width (nominally 90 electrical degrees), high true index pulse which is coincident with the low states of channels A and B.

Block Diagram



Output Waveforms



Definitions

Count (N): The number of bar and window pairs or counts per revolution (CPR) of the codewheel.

One Cycle (C): 360 electrical degrees (°e), 1 bar and window pair.

One Shaft Rotation: 360 mechanical degrees, N cycles.

Position Error ($\Delta\Theta$): The normalized angular difference between the actual shaft position and the position indicated by the encoder cycle count.

Cycle Error (ΔC): An indication of cycle uniformity. The difference between an observed shaft angle which gives rise to one electrical cycle, and the nominal angular increment of 1/N of a revolution.

Pulse Width (P): The number of electrical degrees that an output is high during 1 cycle. This value is nominally 180°e or 1/2 cycle.

Pulse Width Error (ΔP): The deviation, in electrical degrees, of the pulse width from its ideal value of 180°e.

State Width (S): The number of electrical degrees between a transition in the output of channel A and the neighboring transition in the output of channel B. There are 4 states per cycle, each nominally 90°e.

State Width Error (ΔS): The deviation, in electrical degrees, of each state width from its ideal value of 90°e.

Phase (ϕ): The number of electrical degrees between the center of the high state of channel A and the center of the high state of channel B. This value is nominally 90°e for quadrature output.

Phase Error ($\Delta\phi$): The deviation of the phase from its ideal value of 90°e.

Direction of Rotation: When the codewheel rotates in the direction of the arrow on top of the module, channel A will lead channel B. If the codewheel rotates in the opposite direction, channel B will lead channel A.

Optical Radius (R_{OP}): The distance from the codewheel's center of rotation to the optical center (O.C.) of the encoder module.

Index Pulse Width (P_O): The number of electrical degrees that an index is high during one full shaft rotation. This value is nominally 90°e or 1/4 cycle.

Absolute Maximum Ratings

Storage Temperature, T_S	-40°C to +100°C
Operating Temperature, T_A	-40°C to +100°C
Supply Voltage, V_{CC}	-0.5 V to 7 V
Output Voltage, V_O	-0.5 V to V_{CC}
Output Current per Channel, I_{OUT}	-1.0 mA to 5 mA
Shaft Axial Play	±0.25 mm (±0.010 in.)
Shaft Eccentricity Plus Radial Play	0.1 mm (0.004 in.) TIR
Velocity	30,000 RPM[1]
Acceleration	250,000 rad/sec^2[1]

Note:

1. Absolute maximums for HEDS-5140/6140 codewheels only.

Recommended Operating Conditions

Parameter	Symbol	Min.	Typ.	Max.	Units	Notes
Temperature	T_A	-40		100	°C	
Supply Voltage	V_{CC}	4.5	5.0	5.5	Volts	Ripple < 100 mV_{p-p}
Load Capacitance	C_L			100	pF	2.7 kΩ pull-up
Count Frequency	f			100	kHz	Velocity (rpm) x N/60
Shaft Perpendicularity				±0.25	mm	6.9 mm (0.27 in.) from
Plus Axial Play				(±0.010)	(in.)	mounting surface
Shaft Eccentricity Plus				0.04	mm (in.)	6.9 mm (0.27 in.) from
Radial Play				(0.0015)	TIR	mounting surface

Note: The module performance is guaranteed to 100 kHz but can operate at higher frequencies. For the HEDS-9040 #T00 for operation below 0°C and greater than 50 kHz the maximum Pulse Width and Logic State Width errors are 60°e.

Encoding Characteristics

HEDS-9040 (except #T00), HEDS-9140 (except #B00)

Encoding Characteristics over Recommended Operating Range and Recommended Mounting Tolerances unless otherwise specified. Values are for the worst error over the full rotation of HEDS-5140 and HEDS-6140 codewheels.

Parameter		Symbol	Min.	Typ.[1]	Max.	Units
Cycle Error		ΔC		3	5.5	°e
Pulse Width Error		ΔP		7	30	°e
Logic State Width Error		ΔS		5	30	°e
Phase Error		$\Delta\phi$		2	15	°e
Position Error		$\Delta\Theta$		10	40	min. of arc
Index Pulse Width		P_O	60	90	120	°e
CH. I rise after	-25°C to +100°C	t_1	10	100	250	ns
CH. B or CH. A fall	-40°C to +100°C	t_1	-300	100	250	ns
CH. I fall after	-25°C to +100°C	t_2	70	150	300	ns
CH. A or CH. B rise	-40°C to +100°C	t_2	70	150	1000	ns

Note:

1. Module mounted on tolerance circle of ±0.13 mm (±0.005 in.) radius referenced from module Side A aligning recess centers. 2.7 kΩ pull-up resistors used on all encoder module outputs.

Encoding Characteristics
HEDS-9040 #T00

Encoding Characteristics over Recommended Operating Range and Recommended Mounting Tolerances unless otherwise specified. Values are for the worst error over the full rotation of HEDM-614X Option TXX codewheel.

Parameter		Symbol	Min.	Typ.[1]	Max.	Units
Cycle Error		ΔC		3	7.5	°e
Pulse Width Error		ΔP		7	50	°e
Logic State Width Error		ΔS		5	50	°e
Phase Error		$\Delta\phi$		2	15	°e
Position Error		$\Delta\Theta$		2	20	min. of arc
Index Pulse Width		P_O	40	90	140	°e
CH. I rise after CH. B or CH. A fall	-40°C to +100°C	t_1	10	450	1500	ns
CH. I fall after CH. A or CH. B rise	-40°C to +100°C	t_2	10	250	1500	ns

Note:

1. Module mounted on tolerance circle of ±0.13 mm (±0.005 in.) radius referenced from module Side A aligning recess centers. 2.7 kΩ pull-up resistors used on all encoder module outputs.

Encoding Characteristic
HEDS-9140 #B00

Encoding Characteristics over Recommended Operating Range and Recommended Mounting Tolerances unless otherwise specified. Values are for the worst error over the full rotation of HEDM-504X Option BXX codewheel.

Parameter		Symbol	Min.	Typ. [1]	Max.	Units
Cycle Error		ΔC		6	12	°e
Pulse Width Error		ΔP		10	45	°e
Logic State Width Error		ΔS		10	45	°e
Phase Error		$\Delta\Phi$		2	15	°e
Position Error		$\Delta\Theta$		10	40	min. of arc
Index Pulse Width		P_o	50	90	130	°e
CH. I Rise after CH B or CH A fall	-40°C to +100°	t_1	200	1000	1500	ns
CH. I fall after CH. A or CH.B rise	-40°C to +100°	t_2	0	300	1500	ns

Note:

1. Module mounted on tolerance circle of ±0.13 mm (±0.005 in.) radius referenced from module Side A aligning recess centers. 2.7 kΩ pull-up resistors used on all encoder module outputs.

Electrical Characteristics

Electrical Characteristics over Recommended Operating Range.

Parameter	Symbol	Min.	Typ.[1]	Max.	Units	Notes
Supply Current	I_{CC}	30	57	85	mA	
High Level Output Voltage	V_{OH}	2.4			V	I_{OH} = -200 µA max.
Low Level Output Voltage	V_{OL}			0.4	V	I_{OL} = 3.86 mA
Rise Time	t_r		180[2]		ns	C_L = 25 pF R_L = 2.7 kΩ pull-up
Fall Time	t_f		49[2]		ns	

Notes:

1. Typical values specified at V_{CC} = 5.0 V and 25°C.
2. t_r and t_f 80 nsec for HEDS-9040 #T00.

Electrical Interface

To insure reliable encoding performance, the HEDS-9040 and 9140 three channel encoder modules require 2.7 kΩ (±10%) pull-up resistors on output pins 2, 3, and 5 (Channels I, A and B) as shown in Figure 1. These pull-up resistors should be located as close to the encoder module as possible (within 4 feet). Each of the three encoder module outputs can drive a single TTL load in this configuration.

Mounting Considerations

Figure 2 shows a mounting tolerance *requirement* for proper operation of the HEDS-9040 and HEDS-9140. The Aligning Recess Centers must be located within a tolerance circle of 0.005 in. radius from the nominal locations. This tolerance must be maintained whether the module is mounted with side A as the mounting plane using aligning pins (see Figure 5), or mounted with Side B as the mounting plane using an alignment tool (see Figures 3 and 4).

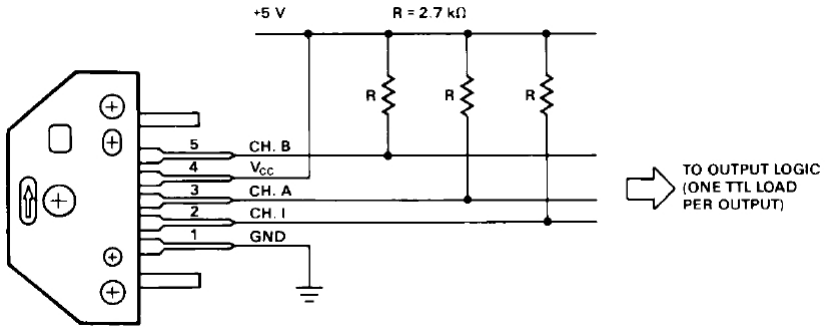


Figure 1. Pull-up Resistors on HEDS-9X40 Encoder Module Outputs.

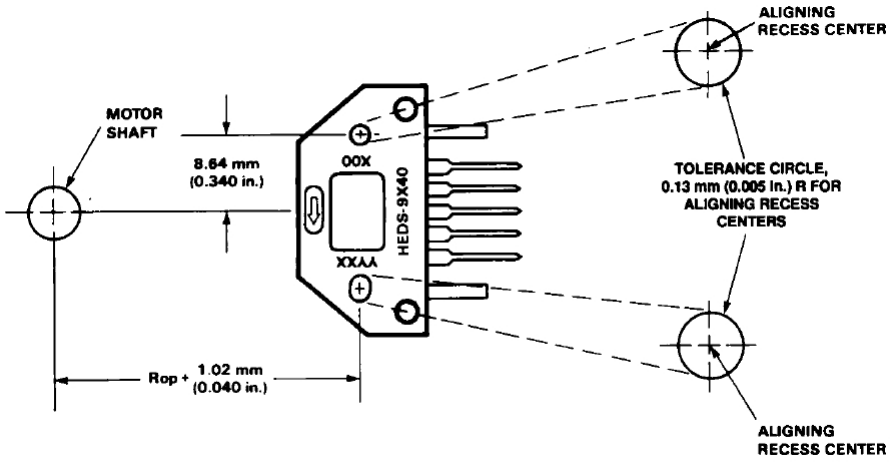


Figure 2. HEDS-9X40 Mounting Tolerance.

Mounting with an Alignment Tool

The HEDS-8905 and HEDS-8906 alignment tools are recommended for mounting the modules with Side B as the mounting plane. The HEDS-8905 is used to mount the HEDS-9140, and the HEDS-8906 is used to mount the HEDS-9040. These tools fix the module position using the codewheel hub as a reference. They will not work if Side A is used as the mounting plane.

The following assembly procedure uses the HEDS-8905/8906 alignment tool to mount a HEDS-9140/9040 module and a HEDS-5140/6140 codewheel:

Instructions:

1. Place codewheel on shaft.
2. Set codewheel height by placing alignment tool on motor base (pins facing up) flush up against the codewheel as shown in Figure 3. Tighten codewheel setscrew and remove alignment tool.
3. Insert mounting screws through module and thread into the motor base. Do not tighten screws.
4. Slide alignment tool over codewheel hub and onto module as shown in Figure 4. The pins of the alignment tool should fit snugly inside the alignment recesses of the module.
5. While holding alignment tool in place, tighten screws down to secure module.
6. Remove alignment tool.

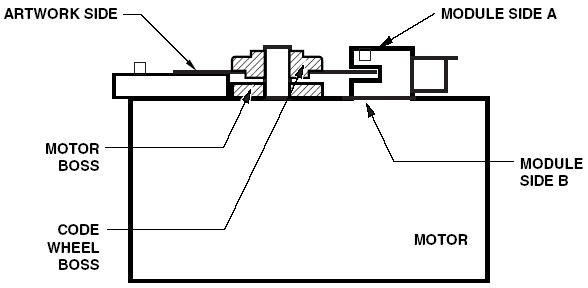


Figure 3. Alignment Tool is Used to Set Height of Codewheel.

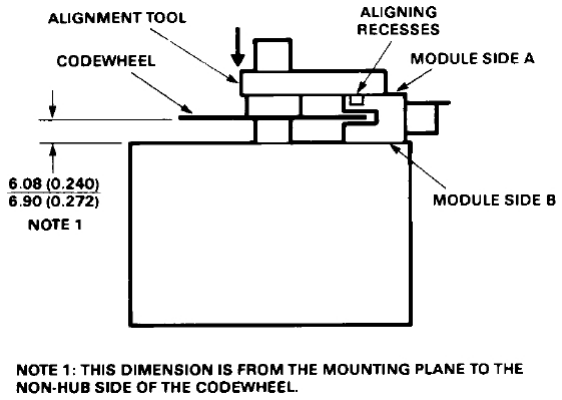


Figure 4. Alignment Tool is Placed over Shaft and onto Codewheel Hub. Alignment Tool Pins Mate with Aligning Recesses on Module.

Mounting with Aligning Pins

The HEDS-9040 and HEDS-9140 can also be mounted using aligning pins on the motor base. (Hewlett-Packard does not provide aligning pins.) For this configuration, Side A must be used as the mounting plane. The aligning recess centers must be located within the 0.005 in. R Tolerance Circle as explained above. Figure 5 shows the necessary dimensions.

Mounting with Aligning Pins

The HEDS-9040 and HEDS-9140 can also be mounted using aligning pins on the motor base. (Avago does not provide aligning pins.) For this configuration, Side A *must* be used as the mounting plane. The aligning recess centers must be located within the 0.005 in. Radius Tolerance Circle as explained in "Mounting Considerations." Figure 5 shows the necessary dimensions.

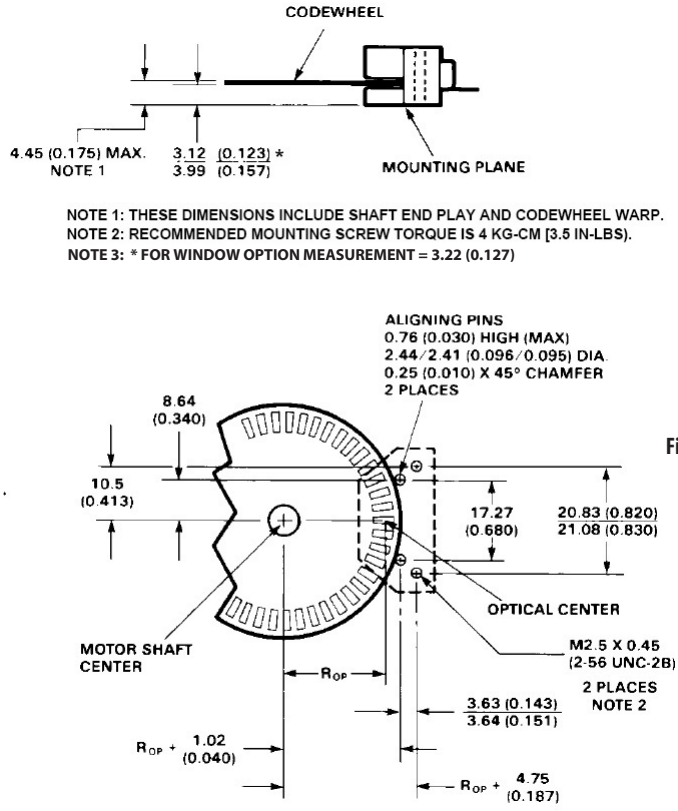


Figure 5. Mounting Plane Side A.

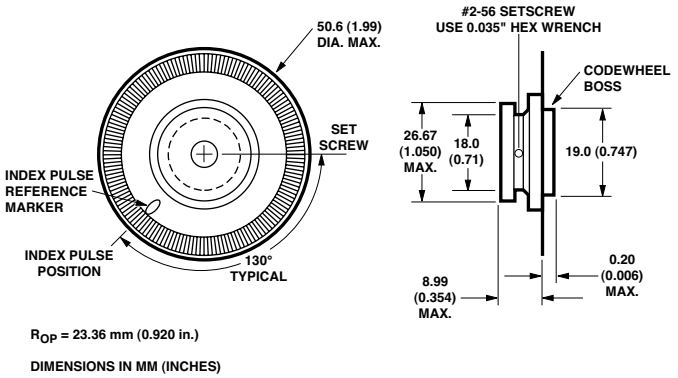


Figure 6a. HEDS-6140 Codewheel Used with HEDS-9040.

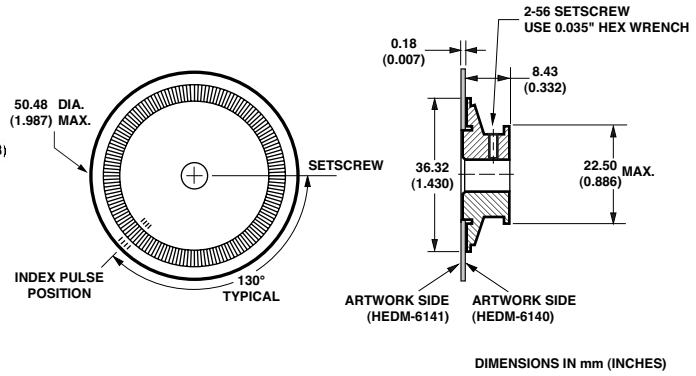


Figure 6b. HEDM-614X Series Codewheel used with HEDS-9040 #T00.

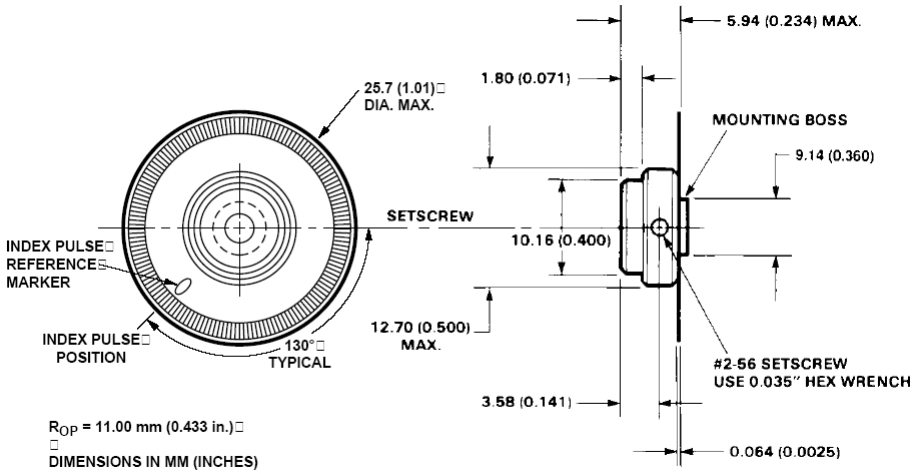
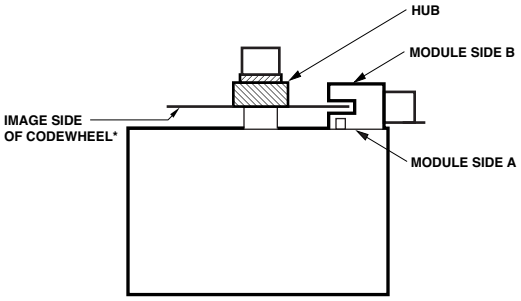


Figure 7. HEDS-5140 Codewheel Used with HEDS-9140.

Orientation of Artwork for HEDS-9040 Option T00 (2000 CPR, 23.36mm Rop) and HEDS-9140 Option B00 (1000CPR, 11.00mm Rop)

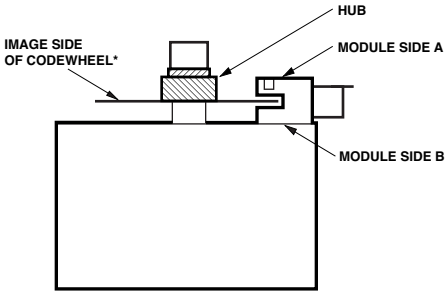
The Index area on the HEDS- 9040 Option T00, 2000 CPR and HEDS-9140 Option B00, 1000 CPR Encoder Module has a nonsymmetrical pattern as does the mating Codewheel. In order for the Index to operate, the "Rightreading" side of the Codewheel disk (the "Artwork Side") must point toward "Side A" of the Module (the side with the connecting pins).

Because the Encoder Module may be used with either "Side A" or with "Side B" toward the Mounting Surface, Avago supplies two versions of Film Codewheels for use with the Option T00 3-channel Module and Option B00 3-Channel Module: Codewheel HEDM-6140 Option TXX and HEDM-5040 Option Bxx has the Artwork Side on the "Hub Side" of the Codewheel/hub assembly and works with "Side B" of the Module on the user's mounting surface. Codewheel HEDM-6141 Option TXX and HEDM-5041-Bxx has the Artwork Side opposite the "Hub Side" and works with "Side A" of the Module on the mounting surface. For the Index to operate, these parts must be oriented as shown in Figure 7a and 7b.



* USE HEDM-6141#Txx or HEDM-5041#Bxx

Figure 7a.



* USE HEDM-6140#Txx or HEDM-5040#Bxx

Figure 7b.

*Please note that the image side of the codewheel must always be facing the module Side A.

Connectors

Manufacturer	Part Number
AMP	103686-4
	640442-5
Avago	HEDS-8902 (2 ch.) with 4-wire leads
	HEDS-8903 (3 ch.) with 5-wire leads
Molex	2695 series with 2759 series term.

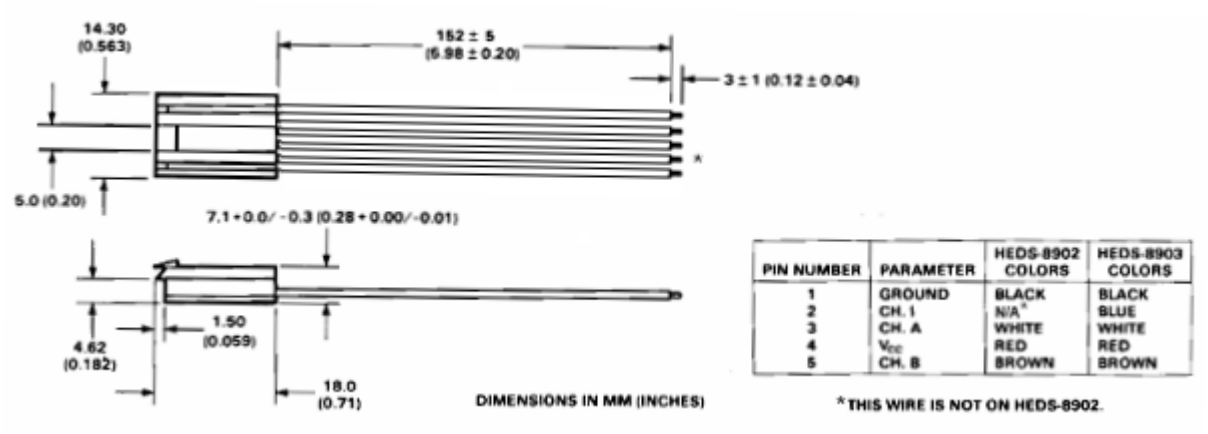
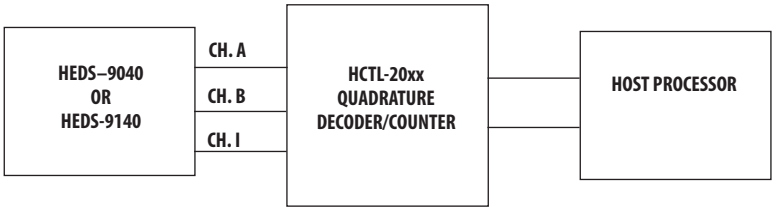
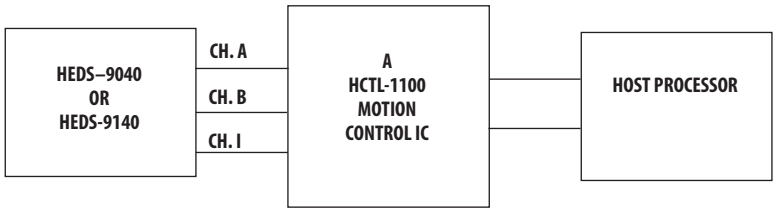


Figure 8. HEDS-8903 Connector.

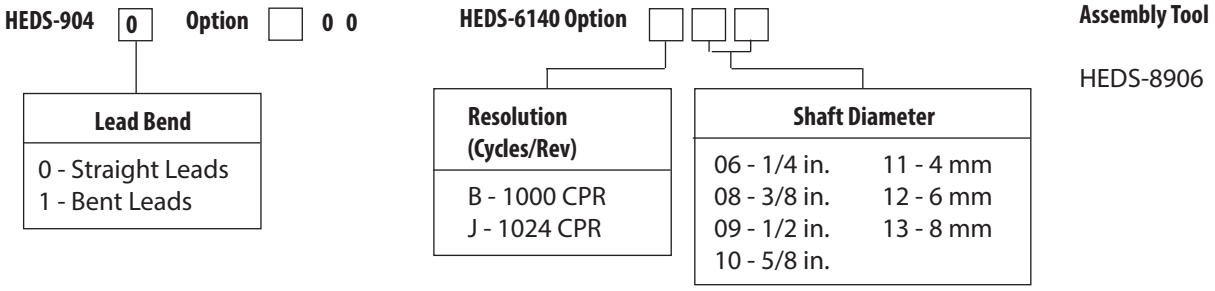
Typical Interfaces





Ordering Information

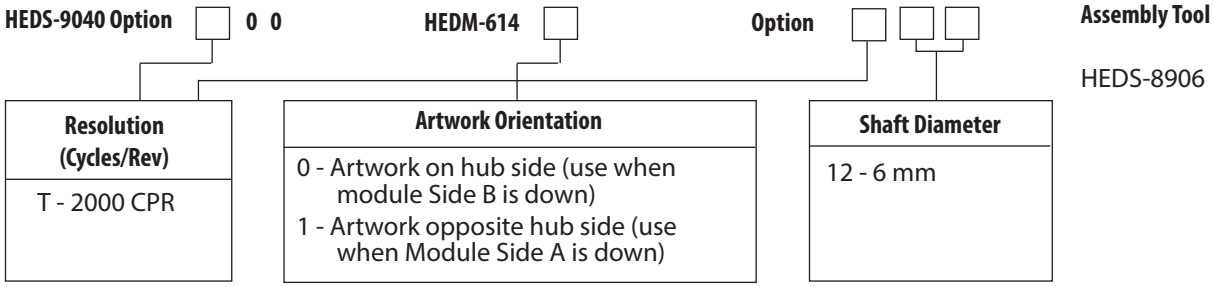
Three Channel Encoder Modules and Codewheels, 23.36 mm Optical Radius.



Assembly Tool

HEDS-8906

Three Channel Encoder Modules and Codewheels, 23.36 mm Optical Radius



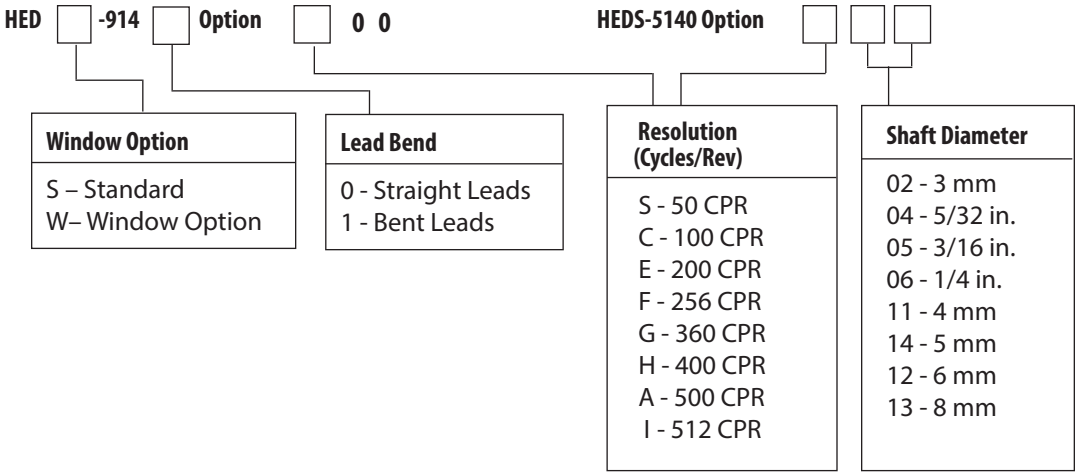
Assembly Tool

HEDS-8906

	A	B	C	D	E	F	G	H	I	J	K	S	T	U
HEDS-9040	*									*			*	
HEDS-9041	*													

	01	02	03	04	05	06	08	09	10	11	12	13	14
HEDS-6140	B						*	*	*	*	*	*	*
	J						*		*			*	*
HEDM-6140	T											*	

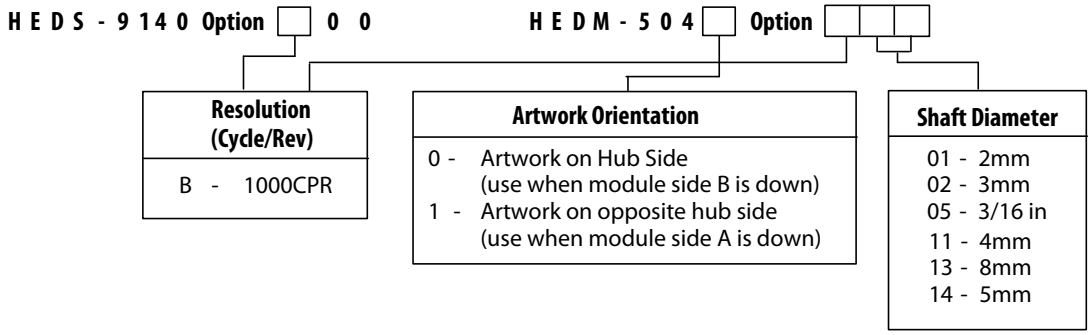
Three Channel Encoder Modules and Codewheels, 11.00 mm Optical Radius



Assembly Tool

HEDS-8905

Three Channel Encoder Modules and Codewheels, 11.000 Optical Radius



Assembly Tool

HEDS-8905

	A	B	C	D	E	F	G	H	I	J	K	S	T	U
HEDS-9140	*	*	*		*	*	*	*	*		*			
HEDS-9141	*				*	*	*							
HEDW-9140	*								*					

		01	02	03	04	05	06	08	09	10	11	12	13	14
HEDS-5140	A		*		*	*	*				*	*	*	*
	C				*		*					*	*	
	E						*				*	*		*
	F				*							*		*
	G						*					*		*
	I		*		*		*				*	*	*	*
HEDM-5040	B	*	*			*					*		*	*
HEDM-5041	B	*	*			*					*		*	*

For product information and a complete list of distributors, please go to our website: **www.avagotech.com**

AV02-1132EN - March 17, 2014