FLOTADOR DE CALADO AUTORREGULABLE/CONSTANTE: Electrónica

Julián Bermúdez Ortega

25 de julio de 2013

Índice general

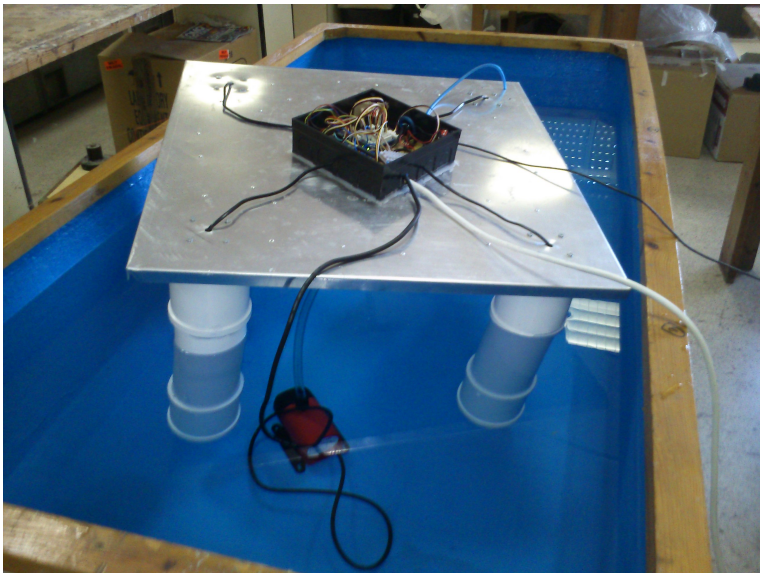
Capítulo 1

Preámbulo

El siguiente proyecto final de carrera, es una de las dos partes que componen en conjunto el verdadero objetivo del mismo: La creación de una plataforma flotante capaz de adrizarse automáticamente y recuperar su calado inicial cuando colocamos un peso sobre ella.

En esta parte, se ha desarrollado un panel electrónico capaz de activar y desactivar las diferentes bombas que permiten controlar el adrizado y calado. La primera parte, la parte estructural, será presentada por Alejandro Macanás Vidal.

La división del proyecto en dos partes se ha debido a motivos ajenos a nuestra voluntad, y no representa con exactitud el proceso de creación del mismo, ya que ambos hemos participado en las dos partes.



Capítulo 2

Introducción

Diciembre de 1947, Laboratorios Bell, John Bardeen, Walter Houser Brattain y William Bradford Shockley estudian y desarrollan el efecto transistor, lo que años más tarde les valdría para ganar el premio Nobel.

El transistor es sin duda, uno de los inventos más importantes del siglo XX. Sin el transistor, la era de las telecomunicaciones en la que vivimos, sería totalmente imposible. Ordenadores, teléfonos móviles, radios, calculadoras, televisiones, memorias de almacenamiento masivo, equipos de música, todos ellos tienen un denominador común, y no es otro que el de requerir transistores para su funcionamiento y existencia.

Nuestro proyecto tratará de aunar electrónica y arquitectura naval en su forma más primitiva, campos entre los que hay una gran distancia temporal, pero que gracias a la naturaleza adaptativa de la electrónica, tienen un camino común de desarrollo. Esta fusión la realizaremos mediante software abierto y hardware modificable.

Capítulo 3

Objetivos

El objetivo de nuestro proyecto de fin de carrera es el cálculo y construcción de una plataforma flotante que sea capaz de regular su calado y/o escora cuando se le aplican fuerzas externas.

Para ello, la plataforma contará con un medidor diferencial de presión por el que sabremos con exactitud el calado del artefacto flotante.

Al colocar un peso sobre nuestra plataforma, el calado aumentará, y esto será percibido por el sensor, que enviará una señal de apertura y expulsión del agua contenida en los tanques de lastre, de modo que al tener menos desplazamiento el calado quedará equilibrado.

Todo el proceso está controlado por una plataforma controladora Arduino TM, y contaremos con una interfaz gráfica para PC a fin de mantener un control sobre el proceso, así como poder elegir si el proceso se realiza automáticamente, ya que existe la posibilidad de que para algún caso concreto no nos interese que el proceso sea automático como por ejemplo en prácticas educacionales.

Capítulo 4

Electricidad

4.1. Bombas periféricas

Son bombas de acuario sumergibles que consiguen una altura de agua de 0.5m -factor por el cual fueron elegida- y una capacidad de entre 150 hasta 300l/h según la especificación del fabricante. Están situadas en el fondo de cada uno de los cilindros y realizan el papel de achique o vaciado de los mismos.

Figura 4.1: Bomba periférica

Figura 4.2: Vista interior cilindro

4.2. Bomba central

Es una bomba de jardín sumergible con una capacidad de 1000l/h. Ésta únicamente tiene como función el llenado de los cilíndro cuando el calado es mayor que el programado.

Figura 4.3: Bomba central

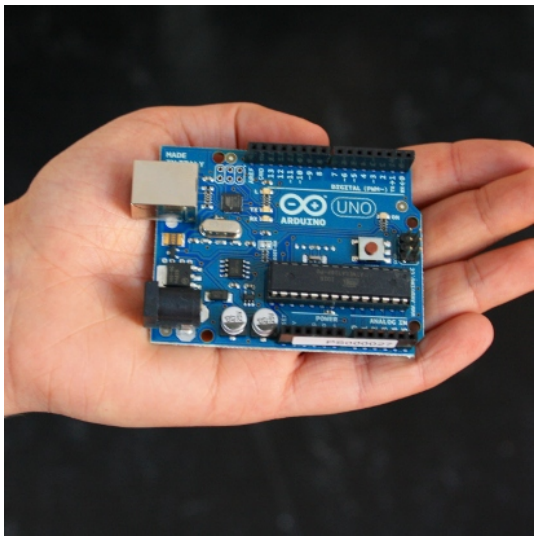
Capítulo 5

Electrónica

5.1. Hardware

5.1.1. Arduino

Arduino es una plataforma de electrónica abierta para la creación de prototipos basada en software y hardware flexibles y fáciles de usar.



Arduino puede tomar información del entorno a través de sus pines de entrada de toda una gama de sensores y puede afectar aquello que le rodea controlando todo tipo de periféricos. El microcontrolador en la placa de Arduino se programa mediante el lenguaje de programación Arduino (basado en Wiring) y el entorno de desarrollo Arduino (basado en Processing). Los proyectos hechos con Arduino puede ejecutarse sin necesisdad de conectar

a un ordenador, si bien tienen la posibilidad de hacerlo y comunicar con diferentes tipos de software (p.ej. Flash, Processing, MaxMSP).

El modelo concreto utilizado es el Arduino Uno el cual cuenta con 14 pines digitales (de los cuales 6 pueden ser utilizados como salidas PWN), 6 analógicos (de los cuales el pin 4 y 5 pueden ser utilizados como SDA y SCL, respectivamente). Ver Figura 5.1.

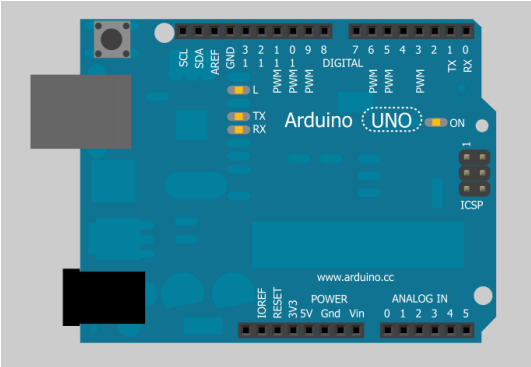


Figura 5.1: Arduino

5.1.2. Acelerómetro

Como su propio nombre indica, un acelerómetro es un instrumento capaz de medir aceleraciones. Actualmente los acelerómetros son un instrumento de moda ya que han abierto un nuevo modo de interactuación de los consumidores con los denominados smartphone y en algunas plataformas de juego.

El objetivo del acelerómetro en este proyecto es medir la inclinación de nuestra plataforma para posteriormente actuar sobre las diferentes bombas.

Debido a algunos inconveniente que iremos detallando a lo largo de este documento, nos hemos visto obligados a utilizar dos acelerómetros direfentes:

- Acelerómetro de 3 ejes modelo SEN21853P basado en MMA7660FC con salida digital I2C.

- Acelerómetro analógico de tres ejes modelo ADXL335.

A continuación representamos un esquema de las conexiones del acelerómetro I2C con Arduino. Ver Figura 5.4.

Figura 5.2: Acelerómetro I2C

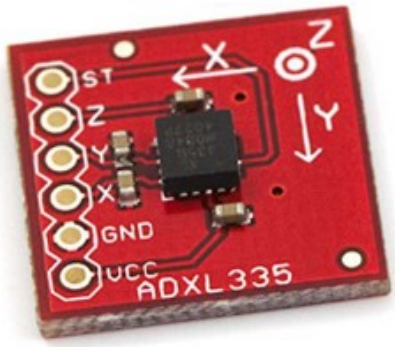


Figura 5.3: Acelerómetro analógico

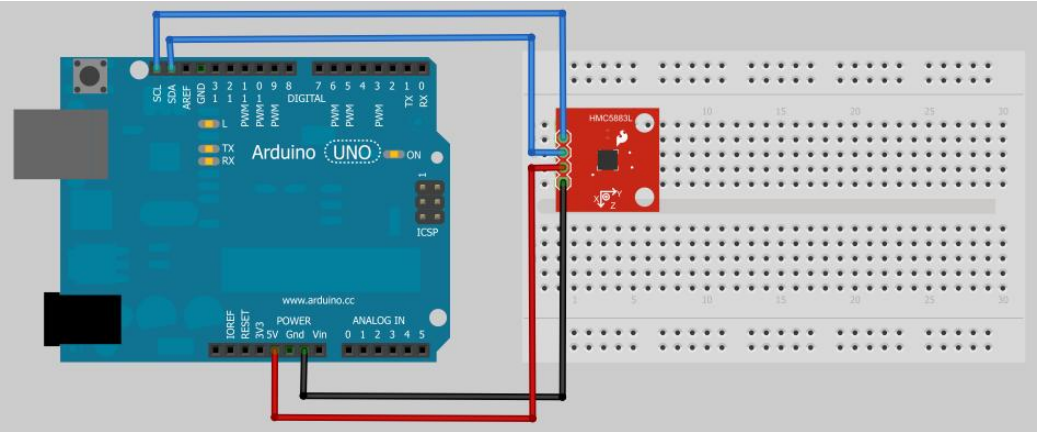


Figura 5.4: Conexión I2C

5.1.3. Relés

Un relé o relevador es un dispositivo electromagnético que funciona como interruptor controlado por un circuito eléctrico en el que, por medio de una bobina y un electroimán se accionan los diferentes contactos.

Figura 5.5: Relé

En nuestro caso, los relés serán los encargados de activar o desactivar las bombas. Por otra parte, el encargado de controlar el relé será Arduino mediante sus salidas digitales, que proporcionan el voltaje necesario para accionar el electroimán. En la figura 5.6 podemos ver un esquema de las conexiones utilizadas para conectar los relés a Arduino.

5.1.4. Bluetooth

El módulo bluetooth nos proporciona una conexión inalámbrica de alta velocidad y sobre todo con mucha flexibilidad de manejo. En nuestro caso

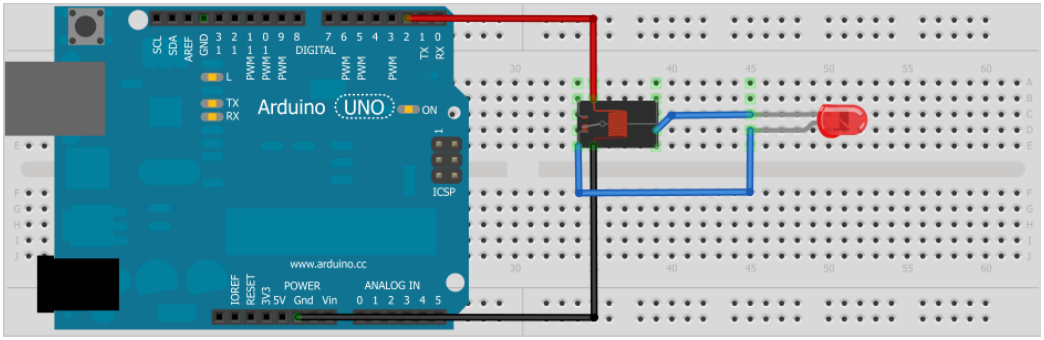


Figura 5.6: Conexión relé

el módulo utilizado es el Grove-Serial WLS31746P compatible con el Shield existente también de Grove, que además puede funcionar como esclavo o maestro para conectar con otros módulos. El puerto del módulo es 2.0+EDR (Enhanced Data Rate) y proporciona hasta 3mbps modulados a 2.4 GHz.

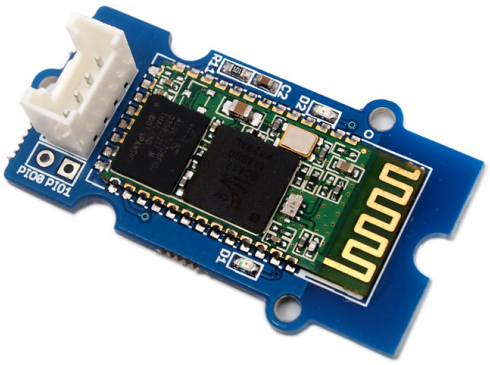


Figura 5.7: Bluetooth

Cuenta con auto-conexión al último dispositivo siempre y cuando éste se encuentre encendido. Esta característica es muy valorable dado el hecho de que el módulo bluetooth tan solo está conectado con la plataforma, lo que nos ahorra tiempo y posible problemas de desconexión.

En su conexión con Arduino, según las especificaciones del fabricante, lo más eficiente es un uso a 5 voltios transfiriendo a 2mbps y sensibilidad de -80 dbm. Dado que la comunicación del bluetooth es del tipo *Serial* la conexión con Arduino es muy sencilla, conectado simplemete las salidas RxT y TxD del módulo bluetooth con dos pines digitales de Arduino, además de la conexión requerida para alimetar el dispositivo. La figura 5.8 muestra un esquema de dicha conexión.

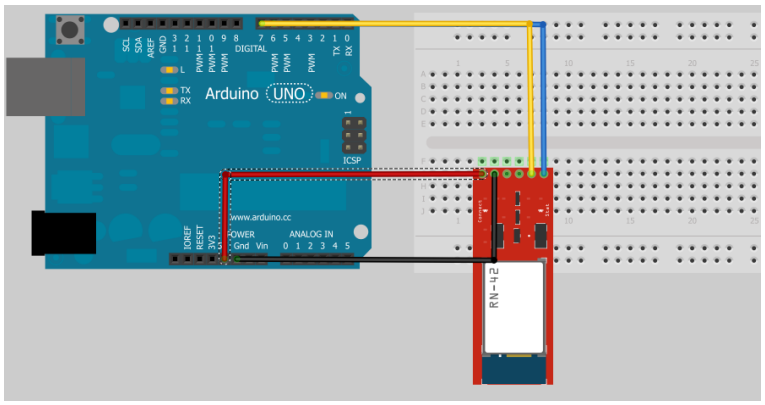


Figura 5.8: Conexión bluetooth

5.1.5. Sensor de presión

El MPX2010DP es un sensor integrado de silicio de alta precisión que mide la presión del aire. Provee una salida de voltaje analógico lineal que varía proporcinalmente con la presión aplicada. Cabe destacar que es un sensor diferencial, es decir, tiene una entrada de referencia de manera que podemos medir la presión con relación a ésta.

Figura 5.9: Sensor de presión

El rango de presión va desde 0 hasta 10kPa, produciendo un voltaje máximo de salida de 25mV lo que nos proporciona una sensibilidad de 2.5mV. Debido a los voltajes que obtenemos a la salida del sensor, nos vemos obligados

a amplificar dicha señal, dado que la sensibilidad de los pines analógicos de Arduino es alrededor de 5mV. Con el objetivo de ampliar la señal y proporcionar al sensor de presión el voltaje necesario para su funcionamiento, que según fabricante es de unos 10V, proyectamos el siguiente circuito electrónico.

5.1.6. Amplificador operacional

Construcción

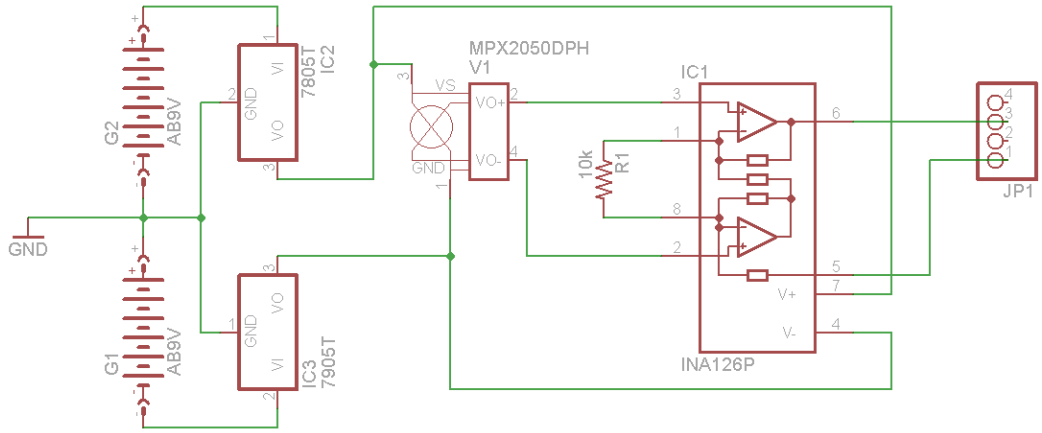


Figura 5.10: Esquema amplificador

En la figura 5.10 podemos ver el esquema del circuito electrónico. Para la realización del diseño hemos utilizado el programa Eagle, el cual nos permite realizar de una forma muy sencilla, tanto el esquema eléctrico como al PCB que posteriormente mostraremos. Como podemos apreciar el circuito consta de los siguientes elementos:

- Un amplificador operacional INA126P.
- Un sensor de presión MPX2010DP.
- Un regulador de tensión LM7805.
- Un regulador de tensión LM7905.
- Dos baterias de 9V.
- Una resistencia de 1kΩ.

A continuación hacemos una breve exposición del funcionamiento de este circuito.

1. Los reguladores de tensión, como su propio nombre indica, regulan el voltaje salida de la baterias de 9V produciendo un voltaje de +5V en el caso del LM7805 y -5V en el LM7905. Con esto obtenemos una tensión simétrica de ±5V que se mantiene constante siempre que el voltaje de las baterías supere el voltaje mínimo de funcionamiento de los reguladores, según fabricande 7V.

2. Una vez obtenido un voltaje constante de 10V, lo utilizamos para alimentar al propio sensor de presión y al amplificador operacional.

3. El voltaje de salida que obtenemos del sensor de presión lo ampliamos gracias al amplificador operacional INA126P. La ganancia que ofrece el INA126P viene determinada, según el fabricante, por la siguiente expresión:
$$G = 5 + \frac{80k\Omega}{R_G} \tag{5.1}$$
donde G es la ganancia y R_G es la resistencia que debemos seleccionar que en nuestro caso es de 1k Ω, con lo que obtenemos una ganacia de $G = 85$.

4. Finalmente conectamos la salida del amplificador operacional a uno de los pines digitales de Arduino.

Una vez que tenemos el esquema eléctrico, podemos obtener, de forma casi directa, el diseño de la PCB con Eagle. Sin embargo es importante cerciorarnos que la anchura de la líneas es la adecuada, además de hacer algunas modificaciones en el trazado de éstas, ya que de lo contrario podríamos tener problemas de conexión posteriormente. En la figura 5.11 podemos ver el diseño definitivo.

La fabricación de PCBs de forma artesanal se puede realizar por diferente métodos (insolacion, plancha, fresado, etc), siendo el de la plancha el elegido debido a que no require de aparatos eléctricos o electrónicos espefícicos, además de obtener unos resultados bastante satisfactorios. Los materiales necesario para este método son:

- Una impresora laser.
- Una plancha.
- Una placa virgen de fibra de vidrio.

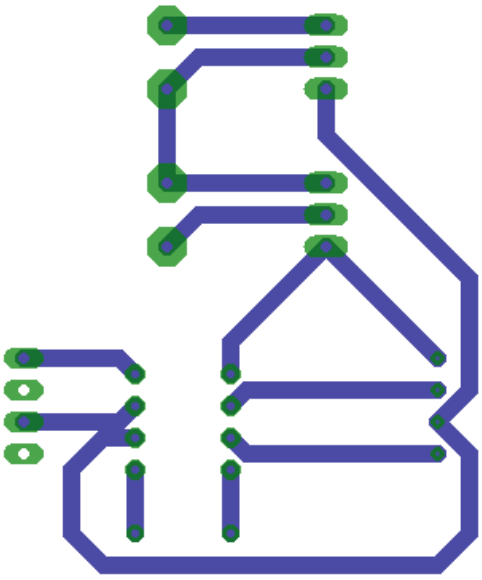


Figura 5.11: Diseño PCB

- Ácido clorhídrico.
- Peróxido de hidrógeno.
- Papel fotográfico.
- Un recipiente de suficiente capacidad para albergar la placa en su interior.
- Un taladro o dremel.
- Una broca de 1mm.
- Un soldador electrónico.
- Estaño.

Como podemos comprobar todos los materiales son fáciles de encotrar ya sea en casa o en la tienda de electrónica más cercana, sin que ello suponga una gran inversión.

El procedimiento seguido es el siguiente:

1. Imprimimos en esquema del PCB en papel fotográfico, es importante que la impresión se haga en una impresora laser.

2. Con ayuda de celo o cinta aislante lo pegamos a la placa virgen, con la cara impresa pegada a la placa.

3. Colocamos un trapo encima del papel fotográfico y lo calentamos durante unos 5min con ayuda de la plancha. La plancha debe estar a una temperatura de unos 200ºC, ya que es aproximadamente la temperatura de fusión de la tinta.

4. Una vez planchado lo sumergimos durante 10min en el recipiente que contendrá, en porciones iguales, ácido clohídrico y peróxido de hidrógeno. Esta mezcla atacará la placa virgen dejando únicamente la tinta que anteriormente hemos adherido con la planta.

5. Limpiamos la tinta dejando al descubierto el cobre.

6. Taladramos los agujero donde posteriormente irán alojados los diferentes componentes electrónicos.

7. Colocamos los componentes y soldamos.

Calibración

Para obtener la relación voltaje-profundidad hemos realizado un sencillo experimento que consiste en sumergir el tubo, que posteriormente utilizaremos en la plataforma, en un recipiente que contiene agua, anotando tanto voltaje como profundidad para un determinado número de ensayos. En la figura 5.3 podemos ver un esquema de dicho experimento.

Según el fabricante la relación entre voltaje-presión sigue una regresión lineal de manera que si obvianos los efectos de compresión que se darán en el interior del tubo, el ajuste voltaje-profundidad seguirá una tendencia lineal.

En la figura 5.13 podemos ver el resultado del ajuste lineal. Dicho ajuste sigue la ecuación descrita a continuación:

$$P = 17{,}77V - 27{,}64 \tag{5.2}$$

donde P es la profundidad en cm y V es el voltaje en V.

5.2. Software

5.2.1. LabVIEW

NI LabVIEW es un lenguaje de programación gráfico diseñado para ingenieros y científicos para desarrollar aplicaciones de pruebas, control y medidas. La naturaleza intuitiva de la programación gráfica de LabVIEW lo hace

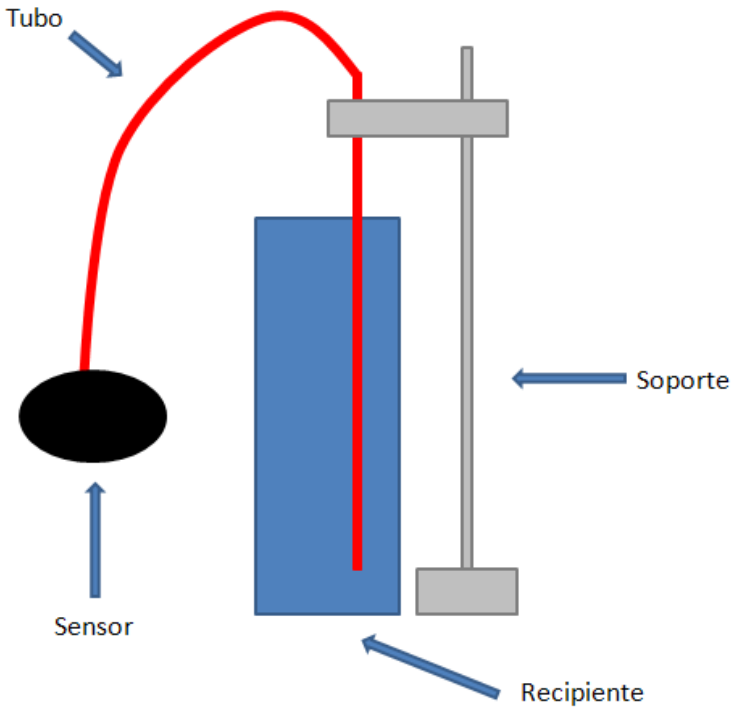


Figura 5.12: Esquema del experimento

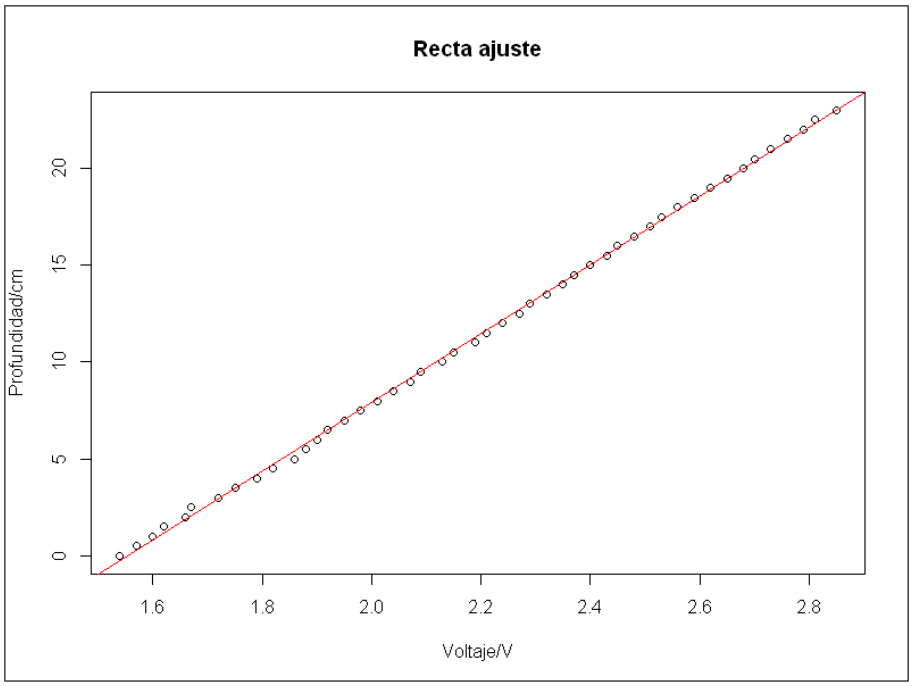


Figura 5.13: Ajuste

fácil de usar por educadores e investigadores para incorporar el software a varios cursos y aplicaciones. El lenguaje de programación que usa se llama lenguaje G, donde la G simboliza que es lenguaje Gráfico.

Este programa fue creado por National Instruments en 1976 para funcionar sobre máquina MAC, aunque en la actualidad está disponible para el resto de plataformas.

Los programas desarrollados con LabVIEW se llama Instrumentos Virtuales, o VIs y su origen provenía del control de instrumentos, aunque hoy en día se ha expandido ampliamente no sólo al control de todo tipo de electrónica sino también a su programación embebida, comunicaciones, matemáticas, etc.

Su principal característica es la facilidad de uso, válido para programadores profesionales como para personas con pocos conocimientos en programación pueden hacer programas relativamente complejos, imposibles para ellos de hacer con lenguajes tradicionales.

Como se ha dicho es una herramienta gráfica de programación, esto significa que los programas no se escriben, sino que se dibujan, facilitando su compresión. Al tener ya pre-diseñados una gran cantidad de bloques, se le facilita al usuario la creación del proyecto, con lo cual en vez de estar una gran cantidad de tiempo en programar un dispositivo/bloque, se le permite

invertir mucho menos tiempo y dedicarse un poco más en la interfaz gráfica y la interacción con el usuario final. Cada VI consta de dos partes difereciadas:

- *Panel Frontal*: es la interfaz con la que interacciona el usuario cuando el programa se esta ejecutando. Los usuario podrán observar los datos del programa actualizados en tiempo real. En esta interfaz se definen los *controles*, usados como entradas, e *indicadores*, usados como salidas.

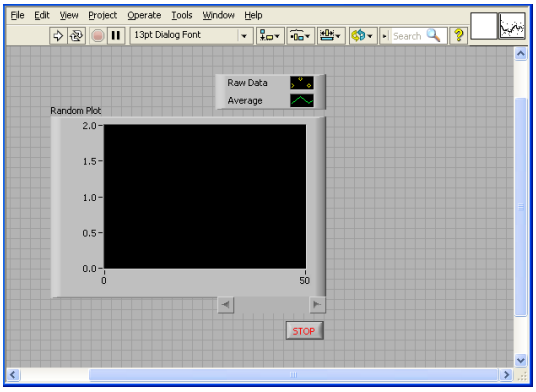


Figura 5.14: Panel Frontal

- *Diagrama de Bloques*: es el programa propiamente dicho, donde se define su funcionalidad, aquí se colocan iconos que realizan una determinada función y se interconecta (el código que controla el programa).

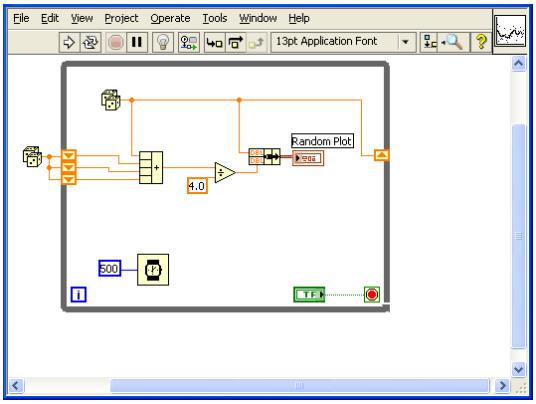


Figura 5.15: Diagrama de bloques

En el panel frontal, encontraremos todo tipos de controles o indicadores, donde cada uno de estos elementos tiene asignado en el diagrama de bloques una terminal, es decir el usuario podrá diseñar un proyecto en el panel frontal

con controles e indicadores, donde estos elementos serán las entradas y salidas que interectuarán con la terminal del VI. Podemos observar en el diagrama de bloques, todos los valores de los controles e idicadores, como van fluyendo entre ellos cuando se está ejecutando un programa VI.

5.2.2. IDE Arduino

El IDE (Integrated Development Environment) es un programa especial que permite escribir *sketches* para la placa Arduino en un lenguaje simple modelado previamente el lenguaje de Processing. El código que escribimos en el sketch es traducido a lenguaje C (el cual es generalmente complicado para usuarios principiantes), y es pasado a el compilador avr-gcc, una importante pieza de código libre que realiza la traducción final en el lenguaje entendible por el microcontrolador.

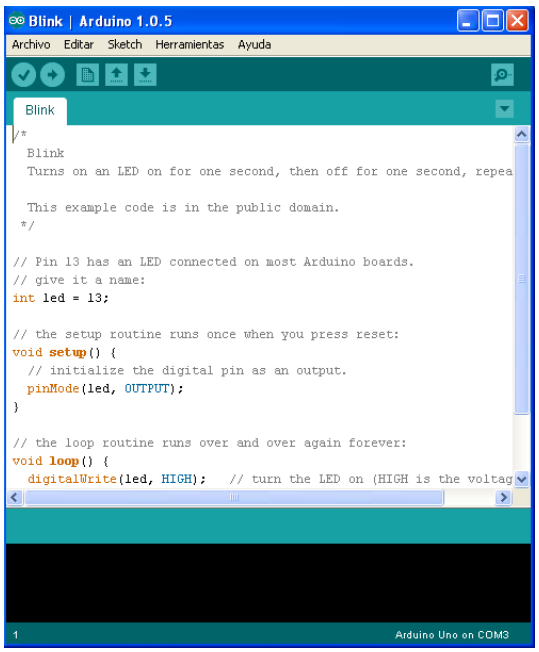


Figura 5.16: IDE Arduino

El ciclo de programación de Arduino es basicamente el siguiente:

1. Conectamos la placa al puerto USB de nuestro ordenador.
2. Escribimos el programa que posteriormente será ejecutado en Arduino.
3. Cargamos el programa en la placa a través del puerto USB.
4. La placa ejecuta el sketch que hemos escrito.

5.2.3. Processing

Processing es un lenguaje de programación y entorno de desarrollo integrado de código abierto basado en Java, de fácil utilización, y que sirve como medio para la enseñanza y producción de proyectos multimedia e interactivos de diseño digital. Fue iniciado por Ben Fry y Casey Reas a partir de reflexiones en el Aesthetics an Computation Group del MIT Media Lab dirigido por John Maeda.

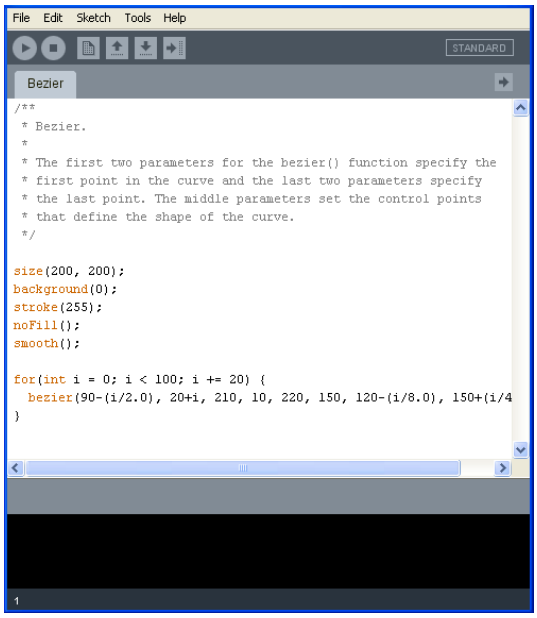


Figura 5.17: Processing

Las principales características de Processing podemos destacar las siguientes:

- Es libre su descarga y además es de código abierno.
- Permite crear programas interactivos en 2D, 3D y PDF.
- Integra la interfaz OpenGL para modelos 3D.
- Está disponible en Linux, Mac y Windows.
- Ofrece más de 100 librerias.
- La documentación disponible en la red es amplia y accesible.

5.2.4. Eagle

El nombre EAGLE es el acrónimo de Easily Applicable Graphical Layout Editor. Éste es un programa de diseño de diagramas y PCBs con autoenrutador. Famoso alrededor del mundo de los proyectos electrónicos DiY, debido a que muchas versiones de este programa tienen una licencia Freeware y gran cantidad de bibliotecas de componentes alrededor de la red.

Este software dispone de un editor de diagramas electrónicos. Los componentes pueden ser colocados en el diagrama con un solo click y fácilmente enrutables con otros componentes a base de "cables"o etiquetas.

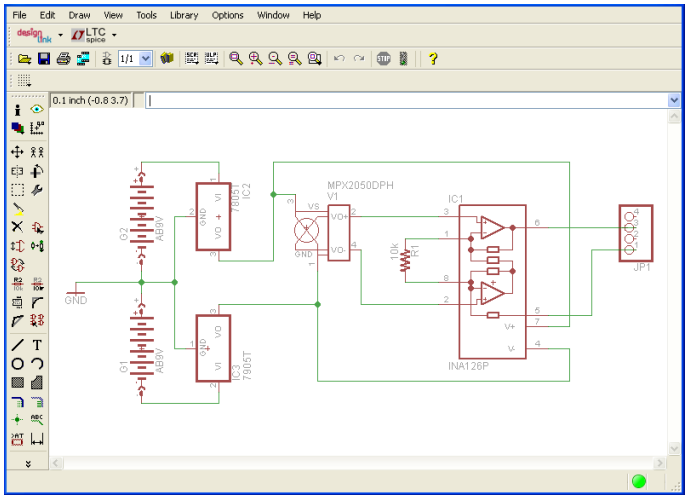


Figura 5.18: Editor de diagramas

Eagle también dispone de un editor de PCBs con un autoenrutados bastante eficiente, capaz de reproducir archivos GERBER y demás, que son utilizados en el momento de la producción.

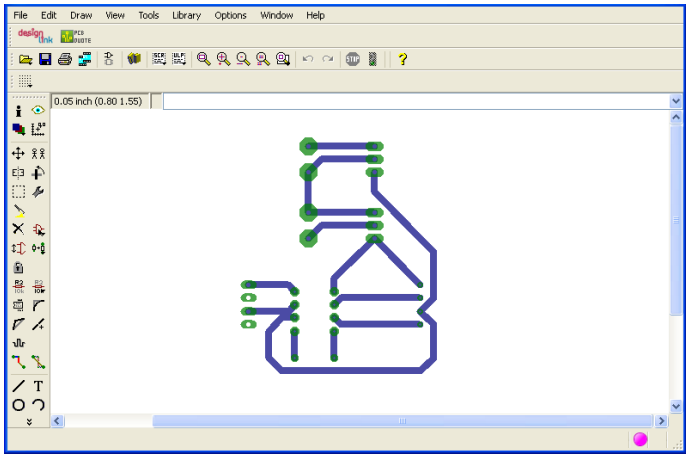


Figura 5.19: Editor de PCBs

Capítulo 6

Programación LabVIEW

6.1. Comunicación con Arduino

Lo primero que tenemos que comprobar es si nuestro Arduino es capaz de comunicarse con Labview, para ello realizamos un simple programa como en que se muestra en la siguiente figura 6.1.

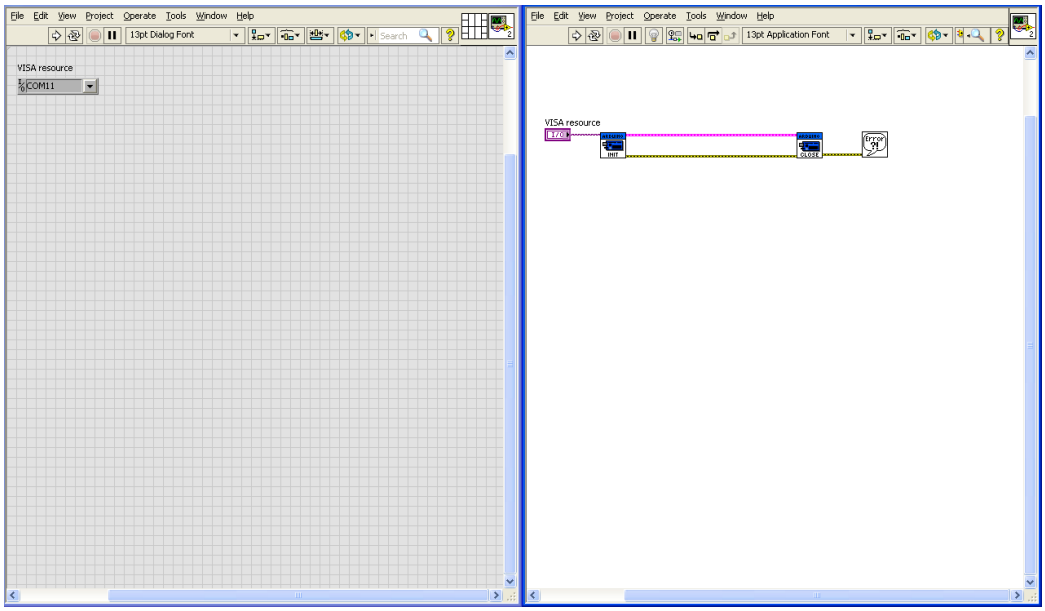


Figura 6.1:

Como nota importante del funcionamiento de LabVIEW podemos señalar que los programas realizados en el diagrama de bloques tienen dirección, es decir, se ejecuta de izquierda a derecha un ejemplo de ello es el programa

de la Figura 6.1, en él se aprecia una función de la Interface de Arduino llamada *Init*, esta es la encargada de arrancar la comunicación con Arduino, que como hemos señalado anteriormente debemos colocarla a la izquierda para que se ejecute en primer lugar, junto a esta función hay un controlador que nos va a permitir elegir el puerto COM utilizado para comuncarnos con Arduino. En la parte derecha se encuentra la función *Close* encargada de cerrar la comunicación con Arduino, además dicha función va acompañada de un indicador de error muy utilizado y de suma importancia.

6.2. Procesamiento de sañal I2C

La señal del acelerómetro es del tipo I2C, de manera que debemos iniciar una comunicación con Arduino utilizando el bus I2C como el siguiente.

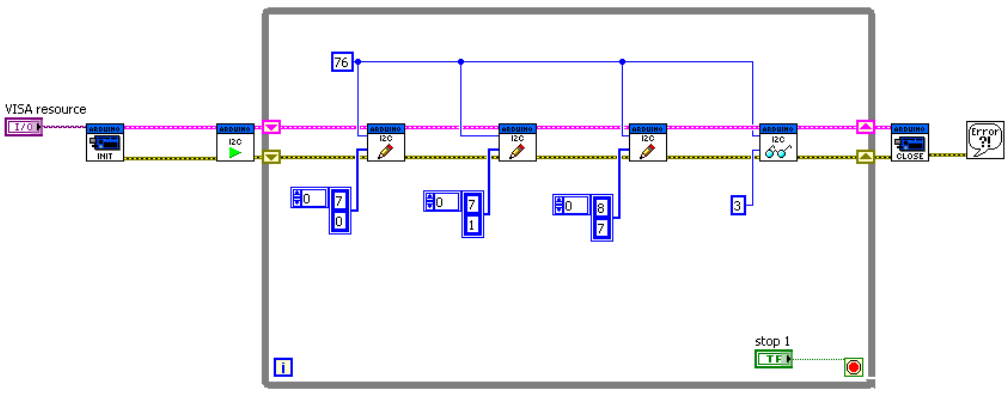


Figura 6.2:

Comenzamos la comunicación I2C con la función llamada *I2C Init* que corresponde al icono con el triángulo verde. Para programar el acelerómetro hemos utilizado la tabla de registro suministrada por el fabricante.

Como indica la tabla anterior debemos poner en modo Standby nuestro acelerómetro, para ello utilizamos la función *I2C Write* que corresponde al icono con el lápiz. A continuación seleccionamos el número de muestras por segundo y activamos el modo Auto-Sleep esta operación corresponde a la segunda función *I2C Write*, para este proyecto hemos seleccionado una muestra por segundo. Finalmente seleccionamos el modo activo en el acelerómetro lo que corresponde a la tercera y última función *I2C Write*. Los datos del dispositivo I2C los leemos con la función *I2C Read* que corresponde con el icono

Address	Name	Definition	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$00	XOUT	6-bit output value X	-	Alert	XOUT[5]	XOUT[4]	XOUT[3]	XOUT[2]	XOUT[1]	XOUT[0]
$01	YOUT	6-bit output value Y	-	Alert	YOUT[5]	YOUT[4]	YOUT[3]	YOUT[2]	YOUT[1]	YOUT[0]
$02	ZOUT	6-bit output value Z	-	Alert	ZOUT[5]	ZOUT[4]	ZOUT[3]	ZOUT[2]	ZOUT[1]	ZOUT[0]
$03	TILT	Tilt Status	Shake	Alert	Tap	PoLa[2]	PoLa[1]	PoLa[0]	BaFro[1]	BaFro[0]
$04	SRST	Sampling Rate Status	0	0	0	0	0	0	AWSRS	AMSRS
$05	SPCNT	Sleep Count	SC[7]	SC[6]	SC[5]	SC[4]	SC[3]	SC[2]	SC[1]	SC[0]
$06	INTSU	Interrupt Setup	SHINTX	SHINTY	SHINTZ	GINT	ASINT	PDINT	PLINT	FBINT
$07	MODE	Mode	IAH	IPP	SCPS	ASE	AWE	TON	-	MODE
$08	SR	Auto-Wake/Sleep and Portrait/Landscape samples per seconds and Debounce Filter	FILT[2]	FILT[1]	FILT[0]	AWSR[1]	AWSR[0]	AMSR[2]	AMSR[1]	AMSR[0]
$09	PDET	Tap Detection	ZDA	YDA	XDA	PDTH[4]	PDTH[3]	PDTH[2]	PDTH[1]	PDTH[0]
$0A	PD	Tap Debounce Count	PD[7]	PD[6]	PD[5]	PD[4]	PD[3]	PD[2]	PD[1]	PD[0]
$0B-$1F	Factory	Reserved	-	-	-	-	-	-	-	-

NOTE: To write to the registers the MODE bit in the MODE (0x07) register must be set to 0, placing the device in Standby Mode.

Figura 6.3:

de las gafas, para esta función hemos indicado que lea los tres primeros bytes correspondientes a las aceleraciones en los ejes x, y, z.

6.3. Lectura del acelerómetro

Dado que el acelerómetro envía valores discretos de 0 hasta 63, es decir, tiene una resolución de 6 bits, para ganar precisión en la lectura realizamos el promedio de un determinado número de nuestras del acelerómetro, este proceso lo podemos apreciar en la imagen 6.4.

Para realizar el promedio de las medidas hemos añadido la función *Loop For* o ciclo for que está representado por un rectángulo en forma de cascada. Esta función nos permite realizar una tarea un determinado número de veces que previamente programamos, para nuestro caso en concreto toma 50 medidas del acelerómetro sumando los valores de cada una de ellas, lo que nos permite posteriormente realizar el promedio. Para realizar el sumatorio de los valores recurrimos a un *Shift Register* que nos guarda en cada ciclos el valor asignado en el anterior. El *Shift Register* viene representado por dos iconos situados a ambos extremos del ciclo for. Para realizar el sumatorio utilizamos la función *Add* representada por un triángulo en cuyo interior aparece un símbolo de suma, en cada ciclo se suma el valor de lectura del acelerómetro con el valor previo del *Shift Register*.

Para el siguiente paso hemos realizado un Sub-VI, es decir, un VI que funciona dentro del VI principal a modo de sub-proyecto. Este Sub-VI nos permite simplificar la tarea de programación al mismo tiempo que evita la acumulación de cableado en el VI principal. En la siguiente imagen podemos

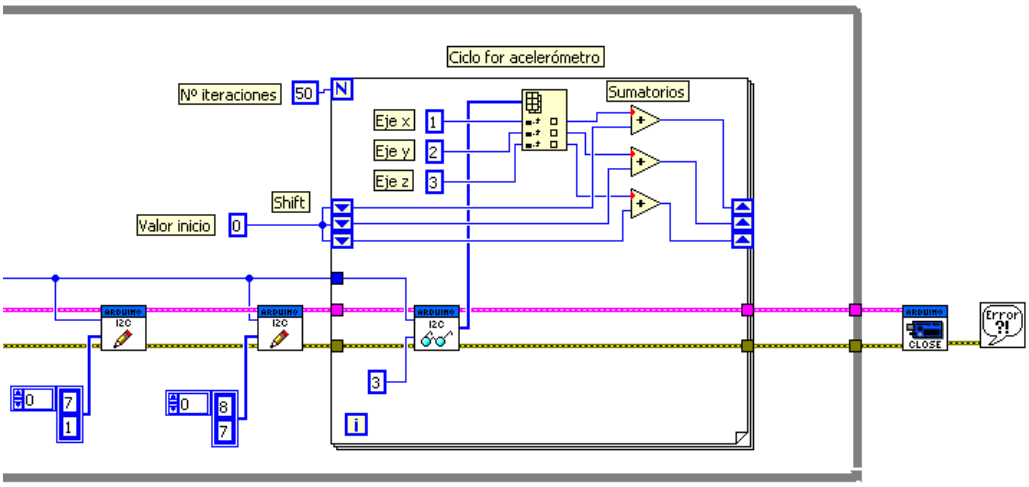


Figura 6.4:

ver el Sub-VI creado para la obtener los ángulos que experimenta nuestro acelerómetro.

En la parte izquierda encontramos los sumatorios que realizamos de la aceleración en los tres ejes correspondientes, utilizando estos sumatorios podemos obtener los ángulos experimentados por la plataforma, con la división dos a dos de las aceleraciones correspondientes a cada eje utilizando la función *Divide*, representado con un triángulo en cuyo interior aparece un símbolo de división. Cuando hacemos la división estamos calculando las tangentes, de manera que es necesario utilizar la función arcotangente, denominada *Atan* en LabVIEW, para obtener el ángulo. La función *Atan* devuelve en radianes, para pasarlo a grados utilizamos la función *Wrap Angle* que nos permite cambiar la unidades de los ángulos de forma muy sencilla.

Una vez que hemos realizado todas las operaciones anteriores debemos definir el Sub-VI, para ello nos dirigimos al panel frontal. En la esquina superior derecha aparece un diagrama (ver figura 6.6), en éste se definen las entradas y salidas que tendrá el Sub-VI, para ello seleccionamos una de las casilla y posteriormente seleccionamos el elemento que hará de salida o entrada. Es importante tener en cuenta que las entradas estarán situadas en el lado izquierdo, mientras que las salidas en el derecho, esto permite que el Sub-VI pueda seguir correctamente el sentido de ejecución en el diagrama de bloques. También podemos editar el icono que aparecerá en el diagrama de bloques lo que nos puede servir de gran ayuda.

Finalmente insertamos el Sub-VI en el VI principal haciendo click en

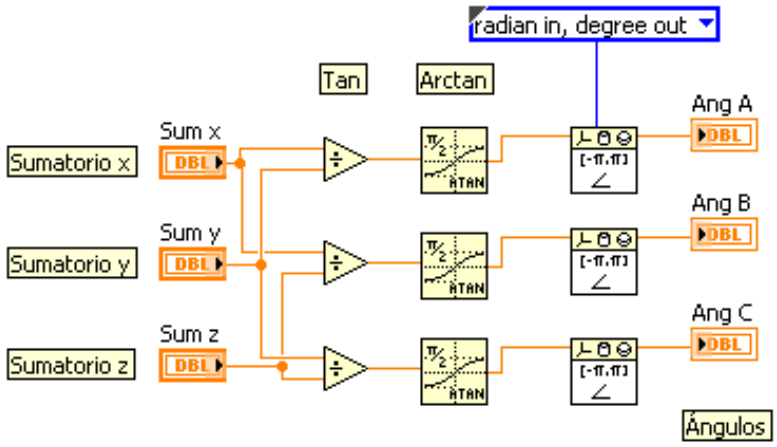


Figura 6.5:

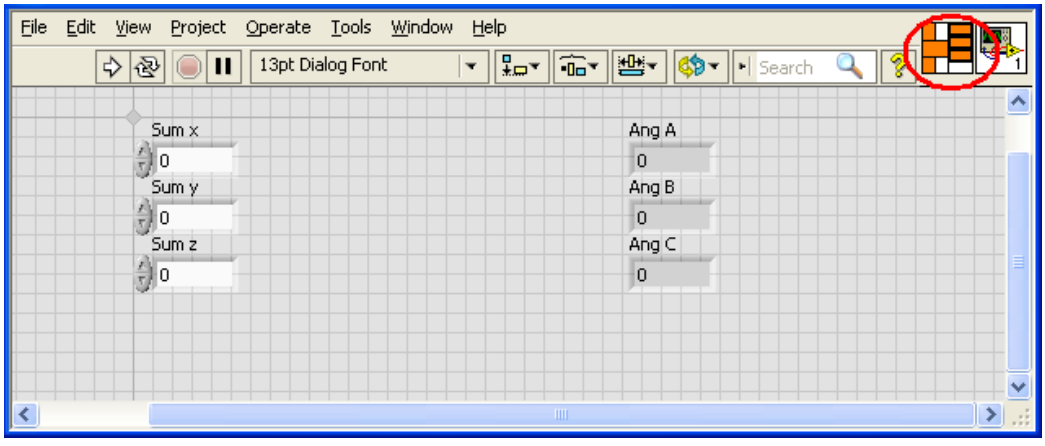


Figura 6.6:

el botón derecho y seleccionando la opción Select VI, además de hacer las conexiones oportunas, como muestra la figura 6.7.

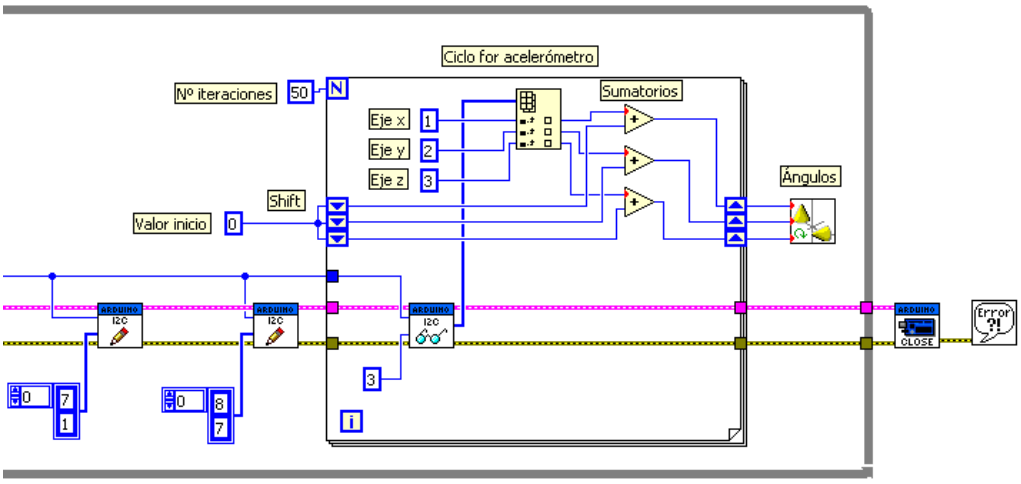


Figura 6.7:

6.4. Lectura sensor de presión

Con el objetivo de simplificar el diagrama de bloques del VI principal, realizamos otro Sub-VI destinado a la lectura del sensor de presión.

Por un lado, como vimos en el apartado de electrónica, el sensor de presión devuelve un voltaje que posteriormente es ampliado gracias al amplificador operacional. Por otro lado, Arduino es capaz de hacer lecturas de voltajes gracias a sus pines analógicos, de forma que el primer elemento del Sub-VI será la función *Analog Read Pin*. Como ya hicimos en la lectura del acelerómetro, optamos por la función *For Loop* con el objetivo de ganar precisión.

Para poder conectar posteriormente el Sub-VI al VI principal, necesitamos crear dos controles y los indicadores, representados por un rectángulo rosa para el control e indicador del Arduino y por un rectángulo color mostaza para los correspondientes a los errores que pudieran aparece al ejecutar el programa. En la figura 6.8 podemos apreciar dichos controles e indicadores.

El siguiente paso es definir el pin analógico que será el encargado de hacer la lectura, para ello creamos una constante donde se indique el número del

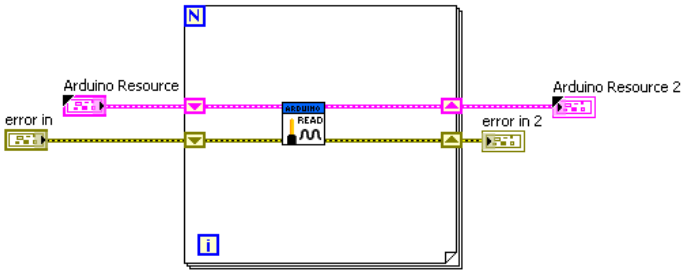


Figura 6.8:

pin (en nuestro caso el pin A2). Además de la constante anterior, debemos crear dos constantes más, una para indicar el número de ciclos for y otra para el valor inicial del sumatorio, es decir, 0. Una vez hecho esto, introducimos la función *Suma*, en el interior del ciclo for, y sumamos la constante inicial junto con la lectura del pin analógico. Para que se produzca el sumario de las lecturas es necesario un *Shift Register* como ya hicimos en el caso del acelerómetro. Este proceso lo podemos ver en la figura 6.9

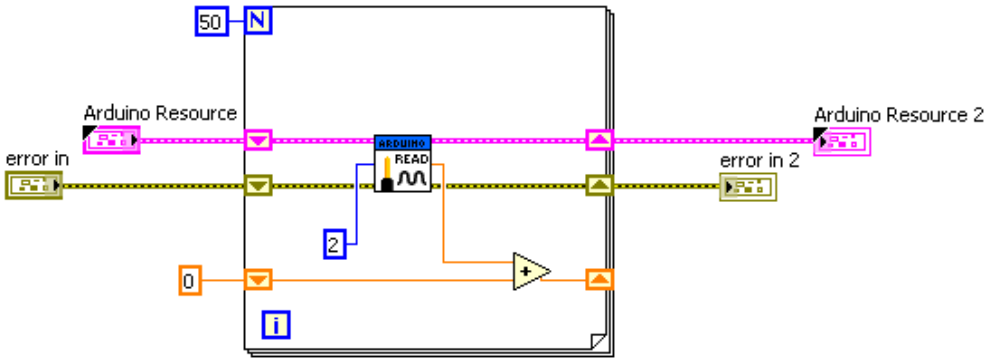


Figura 6.9:

Para finalizar debemos hacer la conversión de voltaje-presión, la cual es lineal como vimos en el apartado del amplificador operacional, por tanto será suficiente con introducir: una función *Divide* para realizar la media; una función *Multiply* que multiplicará la pendiente; y una función *Add* que sumará la ordenada en el origen. Además es necesario crear un indicador que aparecerá posteriormente en el panel frontal.

El resultado final lo podemos ver en la figura 6.10.

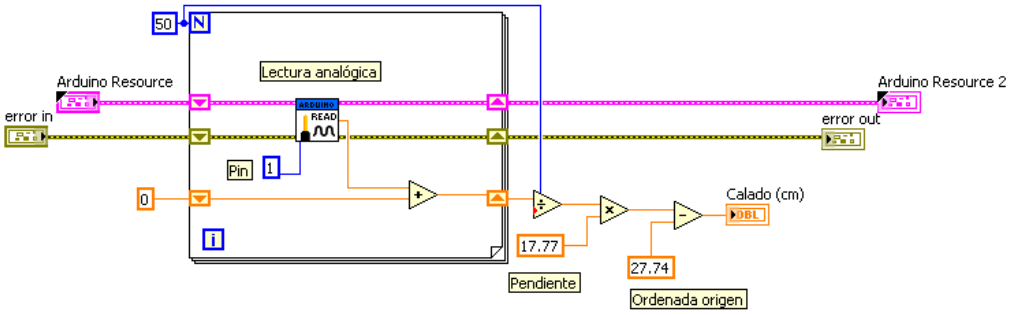


Figura 6.10:

Una vez introducido el Sub-VI en el VI principal conectamos las líneas de ejecución y error de Arduino, quedando una salida de calado disponible para su uso. En la figura 6.11 podemos ver la progresión en el VI principal.

6.5. Control de relés

El control de los relés, parte esencial del proyecto, por ello la hemos dividido en dos subsecciones: control de adrizado y control de calado.

Dicho control se realiza a través de los pines digitales de la placa Arduino. El procedimiento seguido es muy similar al caso del sensor de presión, con la diferencia de que los pines que debemos considerar son digitales. El primer paso que debemos realizar en el Sub-VI, será insertar los cinco pines digitales por medio de la función *Digital Write Pin*, a cada uno de estos pines irán conectadas la bomba correspondiente.

Las bombas se activarán bajo una condición determinada, para regular dicha condición introducimos la función *Case Structure* representada por un rectángulo con un signo de interrogación en la parte derecha. Esta función tiene dos condiciones, true y false, bajo las cuales el programa realizará la operación establecida para cada caso. En nuestro caso particular, cuando la condición sea true, devolverá un 1, es decir activara el pin digital, mientras que cuando sea false, devolverá 0 desactivando el pin. En la figura 6.13, podemos ver el progreso.

6.5.1. Control de adrizado

En cuanto a al control de calado la condición true depende de los ángulos que experimente la plataforma, además de un margen que debemos considerar para evitar problemas de reiteración, dada la imposibilidad de que la

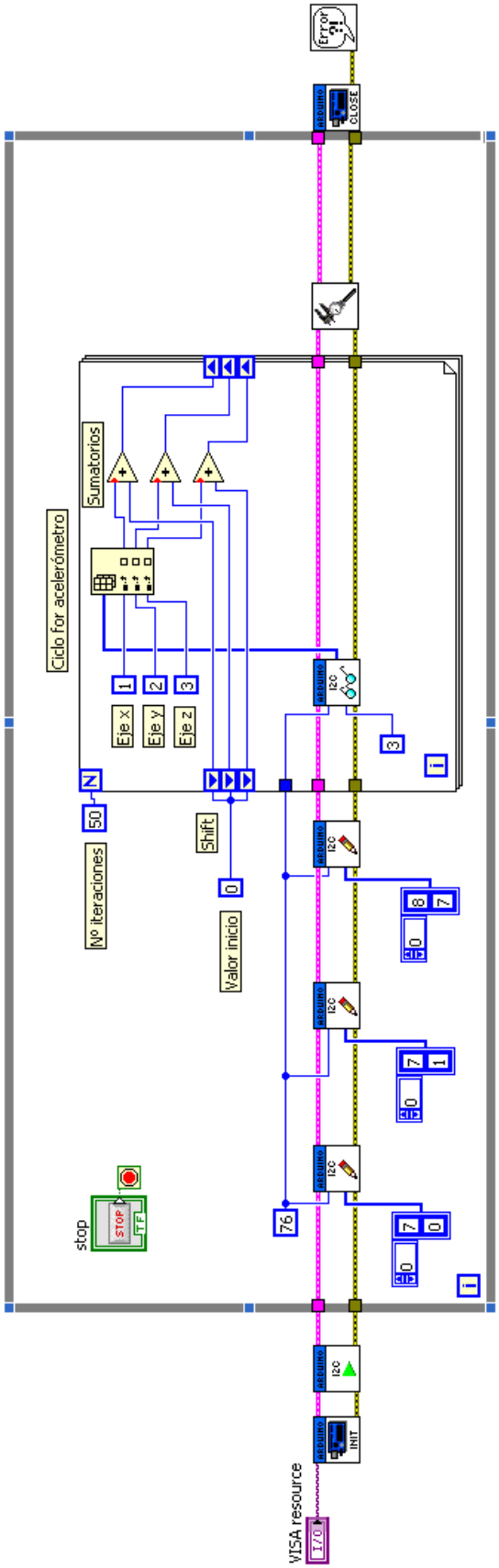

Error
ARDUINO
CLOSE
Sumatorios
Ciclo for acelerómetro
1
2
3
Eje x
Eje y
Eje z
I2C
N
50
Nº iteraciones
Shift
0
Valor inicio
stop
STOP
TF
76
INIT
VISA resource
I/O

Figura 6.11:

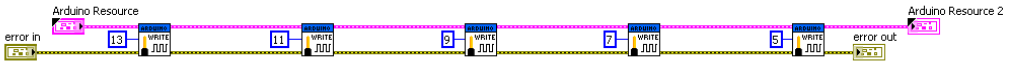


Figura 6.12:

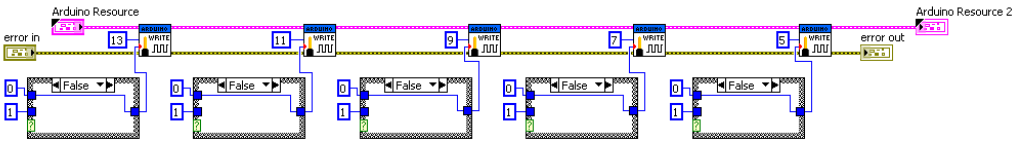


Figura 6.13:

plataforma quede totalmente horizontal. El siguiente paso será por tanto, insertar tres controles correspondientes a: ángulo X, ángulo Y y margen. Junto con estas variables es necesario recurrir a las funciones condicionales: *Greater or Equal*, *Less or Equal*, *And*, *Or*, *Absolute Value* y *Negate*.

Sin entrar en detalle las bombas se activarán individualmente cuando la mesa se incline en la dirección de la pata donde está colocada dicha bomba, o en conjuntos de dos cuando se incline lateralmente. En la figura 6.14 se muestra la programación necesario para realizar dicha acción.

6.5.2. Control de Calado

Para el control del calado interviene otras variables como son: el calado, el calado programado y el margen en el calado. Al igual que en el caso del control de adrizado creamos los controles de dichas variable, que junto con las operaciones lógicas configuran el control completo de los relés.

En este caso la secuencia lógica de activación de las bombas es más simple, conectando las cuatro bombas periféricas cuando el calado es más alto que el programado más el margen, y la central cuando el calado es más bajo que el programado menos el margen.

Una vez terminado el control de los relés, el VI principal queda configurado como muestra la figura 6.16.

6.6. Modelo 3D

Para crear la simulación a tiempo real de la plataforma, hacemos un modelo de la plataforma con un paralelepípedo utilizando las funciones *Create Object* y *Create Box* en el prinpio VI principal. Una vez que hemos creado

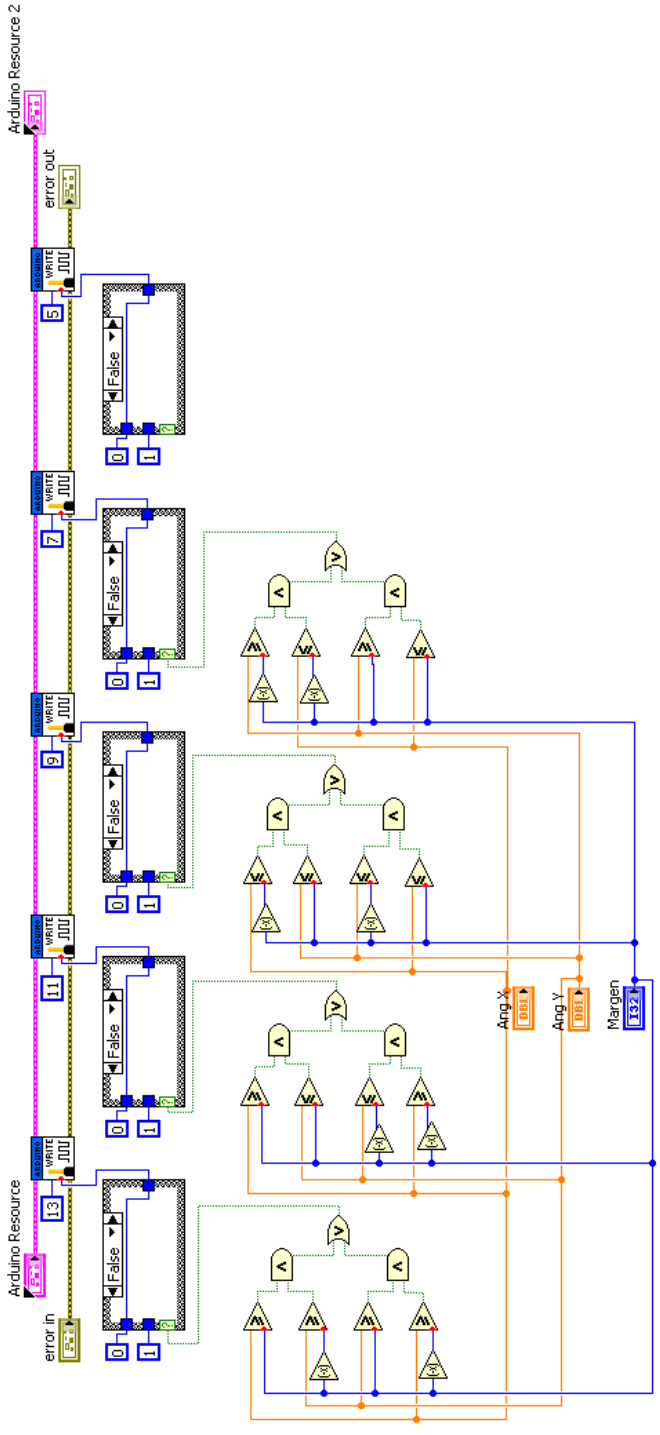

Arduino Resource 2
error out
Arduino Resource
error in
False
Ang x
Ang y
Margen

Figura 6.14:

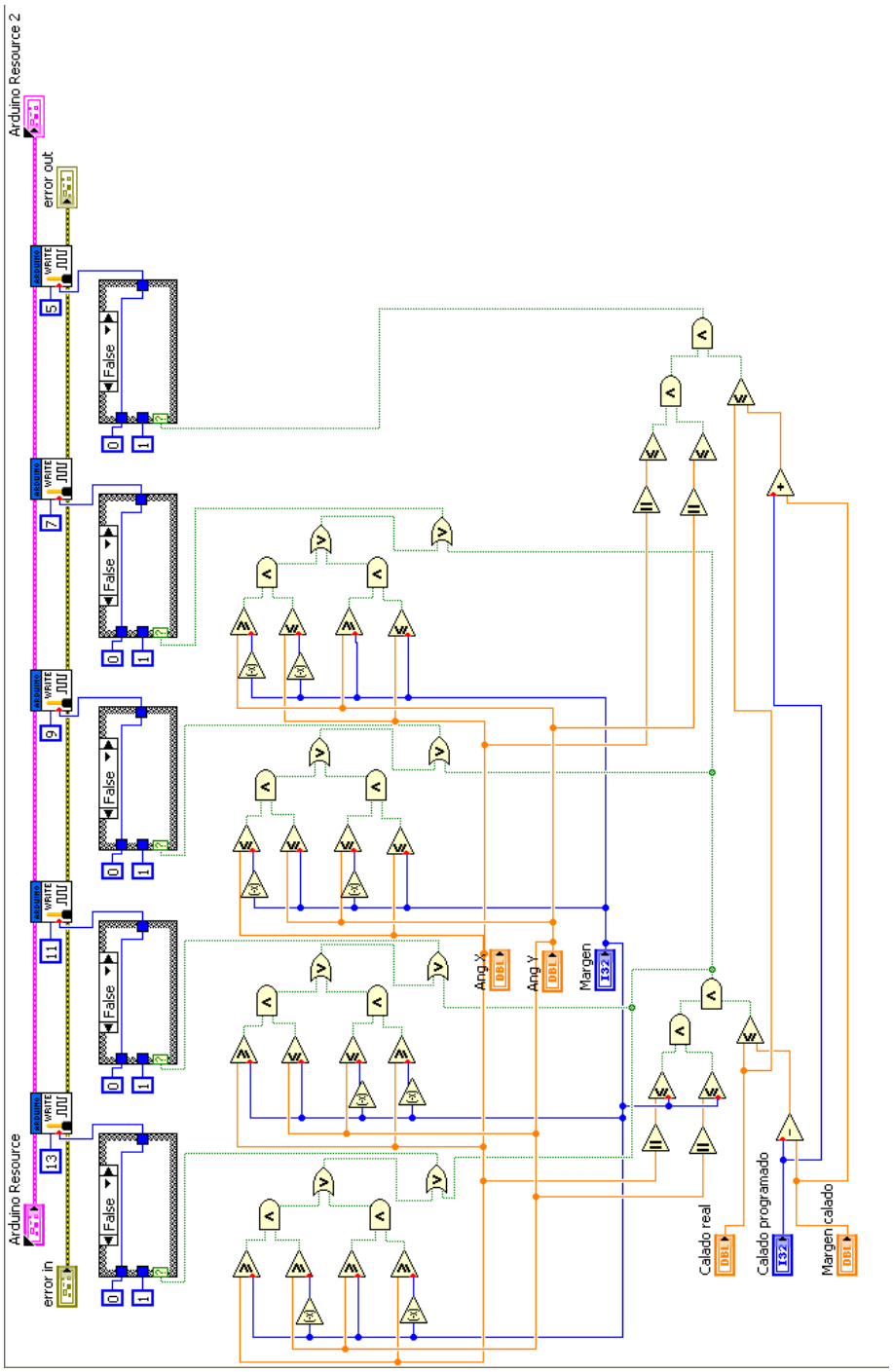

Arduino Resource 2
error out
Arduino Resource
error in
False
Ang X
Ang Y
Margen
Calado real
Calado programado
Margen calado

Figura 6.15:

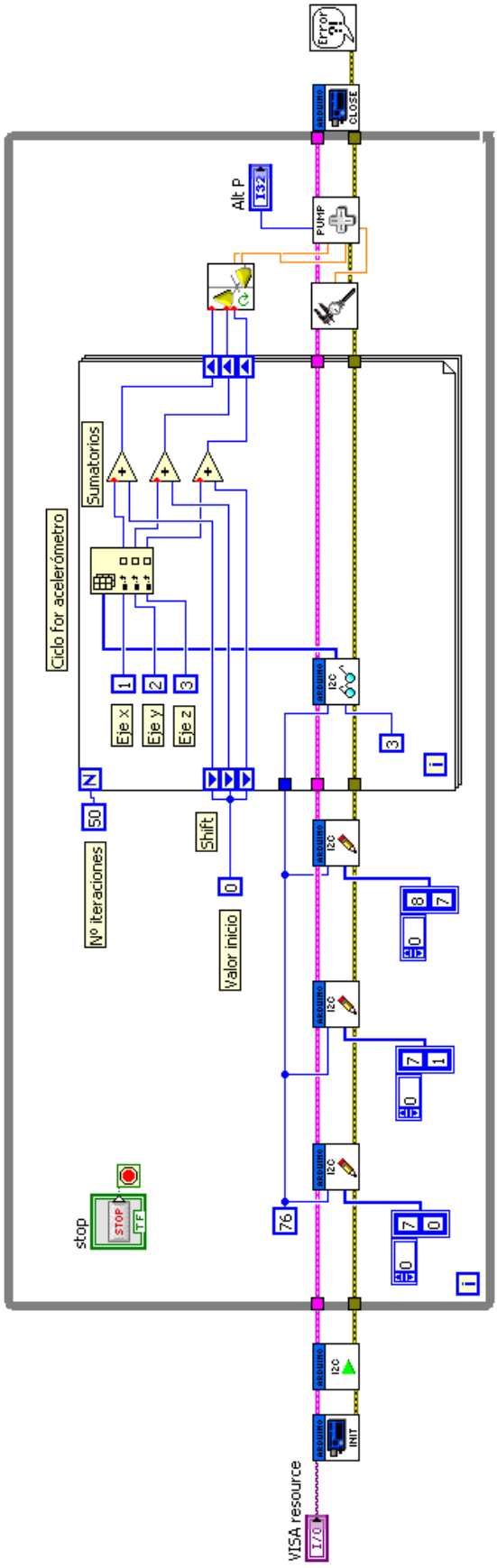

Error
CLOSE
Alt P
PUMP
Sumatorios
Ciclo for acelerómetro
Eje x
Eje y
Eje z
Nº iteraciones
Shift
Valor inicio
stop
STOP
TF
INIT
VISA resource

Figura 6.16:

dicho objeto debemos crear el fondo que se moverá con ayuda de los valores del acelerómetro. En la parte inferior del diagrama de bloques principal podemos apreciar la programación del modelo 3D.

6.7. Resultado final

Una vez que hemos introducido el modelo 3D de la plataforma la programación en LabVIEW queda concluida, obteniendo como resultados finales el diagrama de bloques y el panel de control que se muestran en las figuras 6.17 y 6.18 respectivamente.

El hecho de utilizar LabVIEW vino supeditado por la facilidad de programación, la disposición gráfica en el control de la plataforma, y otras ventajas, además del consejo del profesor titular de este proyecto. Estas ventajas fueron facilitando el intenso camino de la programación hasta el momento en el que probamos el funcionamiento en la plataforma. Inmensa fue nuestra sorpresa cuando, tras miles de pruebas llevadas a cabo en nuestro taller, decidimos evaluar el funcionamiento del programa de LabVIEW en la piscina de la Universidad. En un primer instante todo parecía que funcionaba correctamente, el acelerómetro nos daba valores correctos, el modelo 3D giraba sincronizado con el movimiento real de la plataforma, los relés saltaban en su debido momento, etc. Todo cambió al conectar la plataforma a la tensión de 220V necesaria para activas las bombas, y es que la conexión Serial Port fallaba reiteradamente sin ningún arreglos posibles, esto propició que nuestro proyecto se encaminara a su estado inicial, lo que suponía empezar desde cero la programación utilizando esta vez el lenguaje de Arduino.

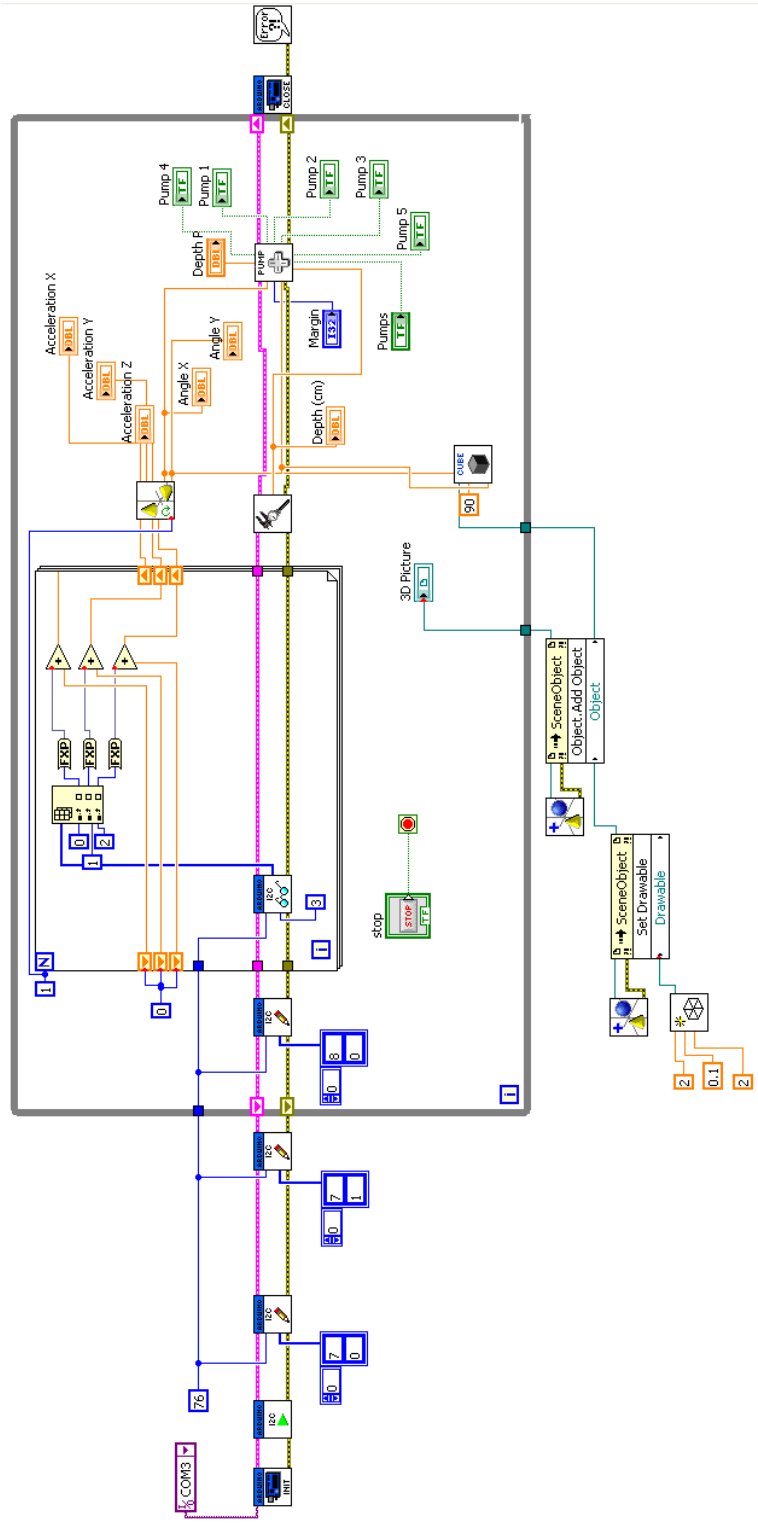


Figura 6.17: Diagrama de bloques final

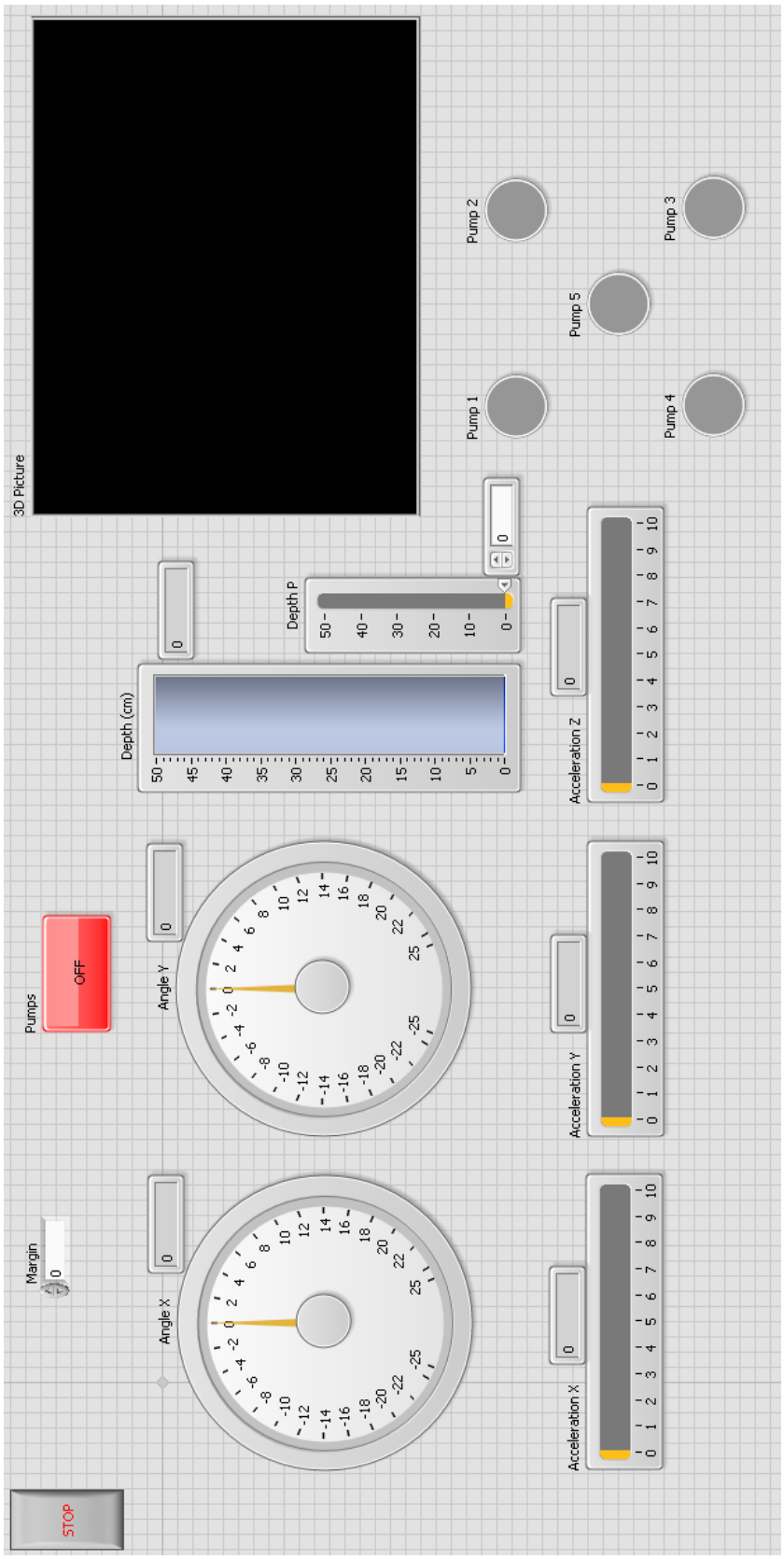


Figura 6.18: Panel de control final

Capítulo 7

Programación Arduino

En el apéndice A encontraremos el código de programación de Arduino utilizado con el acelerómetro analógico, mientras que en el apéndice B el correspondiente código utilizado con el acelerómetro I2C. Para evitar ser repetitivos realizaremos únicamente la exposición del primero de ellos.

7.1. Librerías/Variables

Introducimos las librerías, es decir sub-programas, y variables que posteriormente utilizaremos.

```
#include <SoftwareSerial.h>
#include <math.h>

#define RxD 2
#define TxD 3

SoftwareSerial blueTS(RxD, TxD);

const int n = 50;

int rx[n], ry[n], rz[n], rh[n];
int tx = 0, ty = 0, tz = 0;
int ax = 0, ay = 0, az = 0;
int i = 0, j = 0;
byte m[5] = {0,0,0,0,0}, q[4] = {0,0,0,0}, pump[5] = {0,0,0,0,0}, var;
float angx, angy, h, th = 0, ah = 0;
int mang = 2, mh = 2, hp = 35, turn = 0, mode = 0, tactive = 4;
int input[11];
```

```
const int xpin = A3;
const int ypin = A2;
const int zpin = A1;
const int hpin = A4;
```

Funciones utilizadas

- **#include:** es utilizado para incluir librerías externar, en nuestro caso las librerías *SoftwareSerial* y *math*, encargadas de realizar la comunicación Serial y funciones matemáticas respectivamente.

- **#define:** define una variable dándole un valor constante antes de que compile el programa. No aumentan el tamaño que el programa ocupa en el chip.

- **const:** define una variable de forma que su valor solo puede ser utilizado como lectura pero nunca modificando su valor.

- **byte:** variable capaz de almacenar un número sin signo de 8-bit, desde o hasta 255.

- **int:** variable utilizada para almacenar números enteros (integrer), que guardan valores de 2 bytes. Esto produce un rango entre -2^{15} hasta $2^{15} - 1$.

- **float:** variable para los números en coma flotante. Ocupan 4bytes o lo que es lo mismo 32bits, por tanto pueden alcanzar un valor máximo de $2^{31} - 1$ y un valor mínimo de -2^{31}.

- **array:** las variables que van seguidas de corchetes son arrays, es decir, una colección de variables.

7.2. Setup

La función *setup()* se ejecuta al arrancar el programa. Normalmente empleamos esta función para iniciar variables, establecer el modo de los pines, iniciar librerías, etc. En nuestro caso la hemos utilizado para iniciar la comunicación Serial y Bluetooth Serial además de indicar los pones correspondientes a los relés son de salida.

```
void setup(){
  Serial.begin(115200);
  pinMode(RxD, INPUT);
  pinMode(TxD, OUTPUT);
  setupblueTS();
  delay(15000);
  pinMode(13, OUTPUT);
  pinMode(11, OUTPUT);
  pinMode(9, OUTPUT);
  pinMode(7, OUTPUT);
  pinMode(5, OUTPUT);
  for (int k = 0; k > n; k++){
    rx[k] = 0;
    ry[k] = 0;
    rz[k] = 0;
    rh[k] = 0;
  }
}
```

Funciones utilizadas

- **void:** se una sólo en la declaración de funciones cuando queremos que la función que llamamos no aporte información al programa principal.
- **Serial.begin():** inicia la comunicación Serial estableciento los datos en bits por segundo (baudios).
- **pinMode():** configura el pin especificando si ha de comportarse como una entrada o una salida.
- **delay():** establece un período de pausa del programa (en milisegundos).
- **for():** bucle comúnmente utilizado para trabajar con arrays. Este bucle está formado por tres partes: iniciación, condición e incremento. La iniciación se produce sólo cuando se inicia el ciclo for. Por otro lado el bucle se repetirá si la condición es cierta, mientras que si la condición es false, el bucle se termina.

7.3. Loop

La función *loop()* es la parte más importante del programa puesto que se ejecuta continuamente permitiéndolo al programa variar y responder.

Dentro de la función *loop()* podemos encontrar la llamada de otras funciones que posteriormente trataremos, pero la función principal consiste en recoger lecturas tanto del acelerómetro como del sensor de presión. Utilizando dichas lecturas realizamos lo que se denomina un smoothing o media móvil, esto se utiliza para ganar precisión en las lecturas de los pines digitales puesto que hace la media de un número de lecturas determinado.

```
void loop(){
  tx = tx - rx[i];
  ty = ty - ry[i];
  tz = tz - rz[i];
  th = th - rh[i];

  rx[i] = analogRead(xpin);
  ry[i] = analogRead(ypin);
  rz[i] = analogRead(zpin);
  rh[i] = analogRead(hpin);

  tx = tx + rx[i];
  ty = ty + ry[i];
  tz = tz + rz[i];
  th = th + rh[i];

  ax = map(tx/n,406,271,100,-100);
  ay = map(ty/n,399,264,100,-100);
  az = map(tz/n,412,280,100,-100);
  ah = (th/n*17.774*5/1024-27.64);

  angx = atan2(ax,az)*180/PI;
  angy = atan2(-ay,az)*180/PI;

  blueTS.print(angx); blueTS.print(",");
  blueTS.print(angy); blueTS.print(",");
  blueTS.print(ah); blueTS.print(",");
  for (int i = 0; i < 5; i++){
    blueTS.print(m[i]); blueTS.print(",");
  }
  blueTS.println();

  i = i + 1;
```

```
  if (i >= n){
    i = 0;
    j = j + 1;
  }

  getProcessing();

  if (mode == 1){
    automode();
  }else{
    manualmode();
  }

  delay(10);
}
```

Funciones utilizadas

- **analogRead():** realiza una lectura del pin analógico con una resolución de 10bits. Los voltajes soportados van desde 0 hasta 5V.

- **map():** realiza un cambio de escala de un número a otro.

- **print():** función propia de la comunicación Serial, imprime un valor en el puesto serie como texto ASCII.

- **println():** como la anterior función imprime en el puerto serie en texto ASCII seguido de un retorno de carro.

- **if():** comprueba si la condición impuesta sea cierta, en caso afirmativo ejecuta el código de su interior. Dentro de esta función se utilizan **operadores comparativos** como: ==(igual), ! =(distinto), <(menor que), etc. También se utilizan los denominados **operadores booleanos**: &&(y), ||(o), y !(negación)

7.4. Auto mode

La función *automode()*, es una función creada para controlar los relés de forma automática, es decir, mediante las lecturas del acelerómetro y el sensor de presión. En el diagrama de flujo podemos ver con más detalle el funcionamiento de esta función.

```
void automode(){
 if (j == 4){
   if (angx > mang && angy < mang || angy < -mang && angx > -mang){
     digitalWrite(13, HIGH);
     m[1] = 1; q[1] = q[1] + 1;
   }else{
     m[1] = 0; q[1] = 0;
   }
   if (angx > mang && angy > -mang || angy > mang && angx > -mang){
     digitalWrite(11, HIGH);
     m[2] = 1; q[2] = q[2] + 1;
   }else{
     m[2] = 0; q[2] = 0;
   }
   if (angx < -mang && angy > -mang || angy > mang && angx < mang){
     digitalWrite(9, HIGH);
     m[3] = 1; q[3] = q[3] + 1;
   }else{
     m[3] = 0; q[3] = 0;
   }
   if (angx < -mang && angy < mang || angy < -mang && angx < mang){
     digitalWrite(7, HIGH);
     m[4] = 1; q[4] = q[4] + 1;
   }else{
     m[4] = 0; q[4] = 0;
   }
   if (q[1] > 8 || q[2] > 8 || q[3] > 8 || q[4] > 8){
     digitalWrite(5, HIGH);
     m[0] = 1;
   }else{
     m[0] = 0;
   }

   if (abs(angx) < mang && abs(angy) < mang){
     if (ah > (hp + mh)){
       digitalWrite(13, HIGH);
       digitalWrite(11, HIGH);
       digitalWrite(9, HIGH);
       digitalWrite(7, HIGH);
       m[1] = m[2] = m[3] = m[4] = 1;
     }
```

```
      if (ah < (hp - mh)){
        digitalWrite(5, HIGH);
        m[0] = 1;
      }else{
        m[0] = 0;
      }
    }

    blueTS.print(angx); blueTS.print(",");
    blueTS.print(angy); blueTS.print(",");
    blueTS.print(ah); blueTS.print(",");
    for (int i = 0; i < 5; i++){
      blueTS.print(m[i]); blueTS.print(",");
    }
    blueTS.println();

    delay(tactive*1000);
    digitalWrite(13, LOW);
    digitalWrite(11, LOW);
    digitalWrite(9, LOW);
    digitalWrite(7, LOW);
    digitalWrite(5, LOW);

    j = 0;
  }
}
```

Funciones utilizadas

- **digitalWrite():** escribe un valor HIGH o LOW hacia un pin digital.
- **abs():** calcula el valor absoluto de un número.

7.5. Manual mode

La función *manualmode*, está diseñada para controlar las bomba de forma manual gracias a la interacción con Processing.

```
void manualmode(){
  if (pump[1] == 1){
    digitalWrite(13, HIGH);
```

```
    m[1] = 1;
  }else{
    digitalWrite(13, LOW);
    m[1] = 0;
  }
  if (pump[2] == 1){
    digitalWrite(11, HIGH);
    m[2] = 1;
  }else{
    digitalWrite(11, LOW);
    m[2] = 0;
  }
  if (pump[3] == 1){
    digitalWrite(9, HIGH);
    m[3] = 1;
  }else{
    digitalWrite(9, LOW);
    m[3] = 0;
  }
  if (pump[4] == 1){
    digitalWrite(7, HIGH);
    m[4] = 1;
  }else{
    digitalWrite(7, LOW);
    m[4] = 0;
  }
  if (pump[0] == 1){
    digitalWrite(5, HIGH);
    m[0] = 1;
  }else{
    digitalWrite(5, LOW);
    m[0] = 0;
  }

  j = 0;
}
```

7.6. Processing communication

La función *getProcessing()* está diseñada recoger la información proporcionada por Processing a través de la comunicación Serial.

```
int getProcessing(){
  while (blueTS.available()){
    var = blueTS.read();
    switch(var){
      case 'A': var = blueTS.read();
        if (var < 50) turn = var;
      case 'B': var = blueTS.read();
        if (var < 50) mode = var;
      case 'C': var = blueTS.read();
        if (var < 50) pump[1] = var;
      case 'D': var = blueTS.read();
        if (var < 50) pump[2] = var;
      case 'E': var = blueTS.read();
        if (var < 50) pump[3] = var;
      case 'F': var = blueTS.read();
        if (var < 50) pump[4] = var;
      case 'G': var = blueTS.read();
        if (var < 50) pump[0] = var;
      case 'H': var = blueTS.read();
        if (var < 50) hp = var;
      case 'I': var = blueTS.read();
        if (var < 50) mang = var;
        Serial.println(mang);
      case 'J': var = blueTS.read();
        if (var < 50) mh = var;
      case 'K': var = blueTS.read();
        if (var < 50) tactive = var;
    }
  }
}
```

Funciones utilizadas

- **while():** bucle similar al for con la diferencia de que se repite mientras la condición impuesta sea cierta.

- **switch():** función similar a if capaz de ejecutar diferentes. Junto con

esta función actúa el comando **case**, de forma que la sentencia switch compara el valor de una variable con el valor especificado en las sentencias case.

7.7. Setup Bluetooth

Utilizamos esta función para iniciar la comunicación Serial Bluetooth con Arduino siguiendo las instrucciones del fabricante.

```
void setupblueTS(){
  blueTS.begin(115200);
  delay(1000);
  blueTS.print("\r\n+STWMOD=0\r\n");
  blueTS.print("\r\n+STNA=SeeedBTSlave\r\n");
  blueTS.print("\r\n+DLPIN\r\n");
  blueTS.print("\r\n+STOAUT=1\r\n");
  blueTS.print("\r\n+STAUTO=0\r\n");
  delay(2000);
  blueTS.print("\r\n+INQ=1\r\n");
  delay(2000);
  blueTS.flush();
}
```

7.8. Diagramas de flujo

En las figuras 7.1 y 7.2, podemos ver los diagramas de flujo de la programación en general realizada con Arduino.

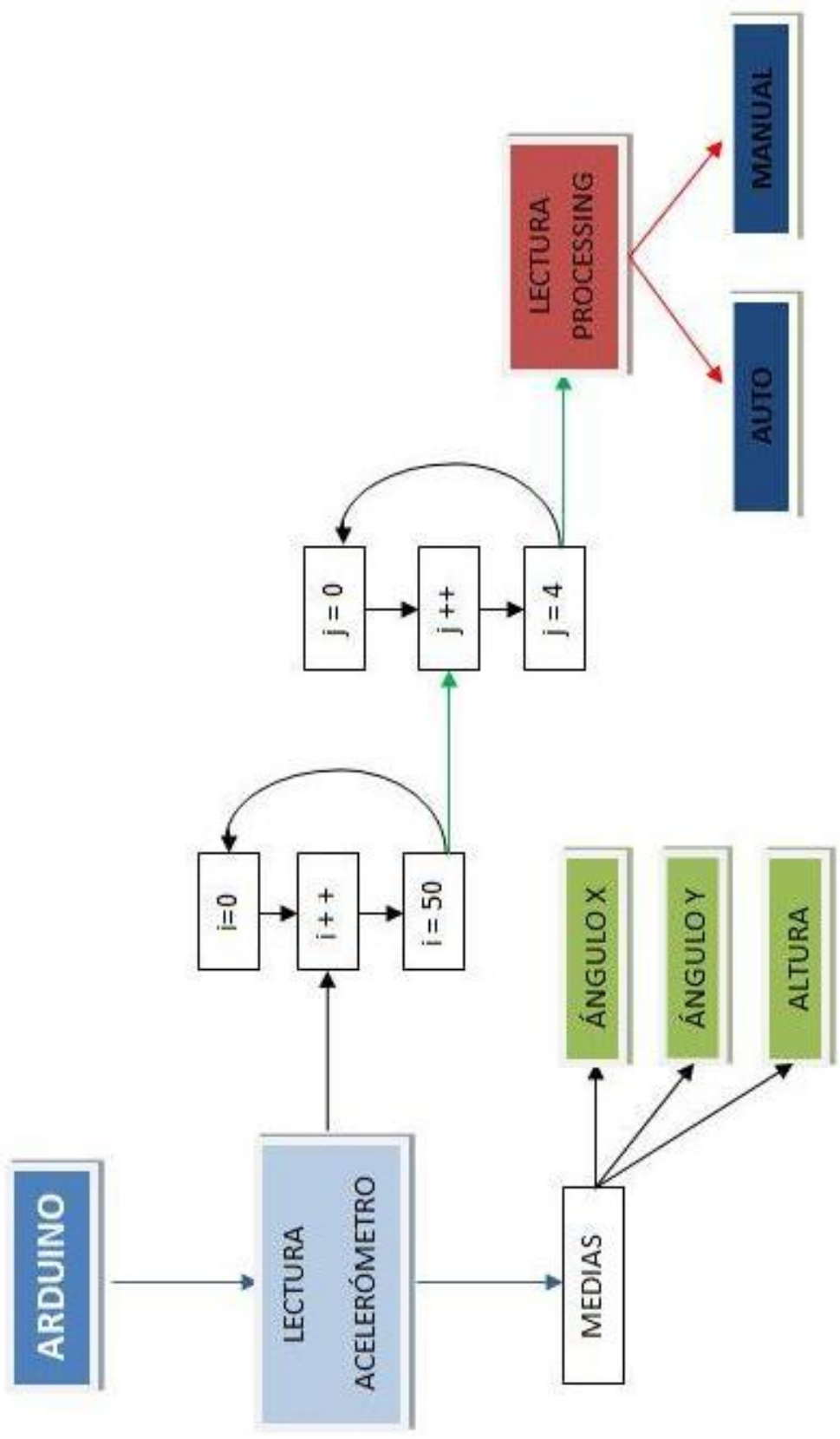


Figura 7.1: Diagrama de flujo 1

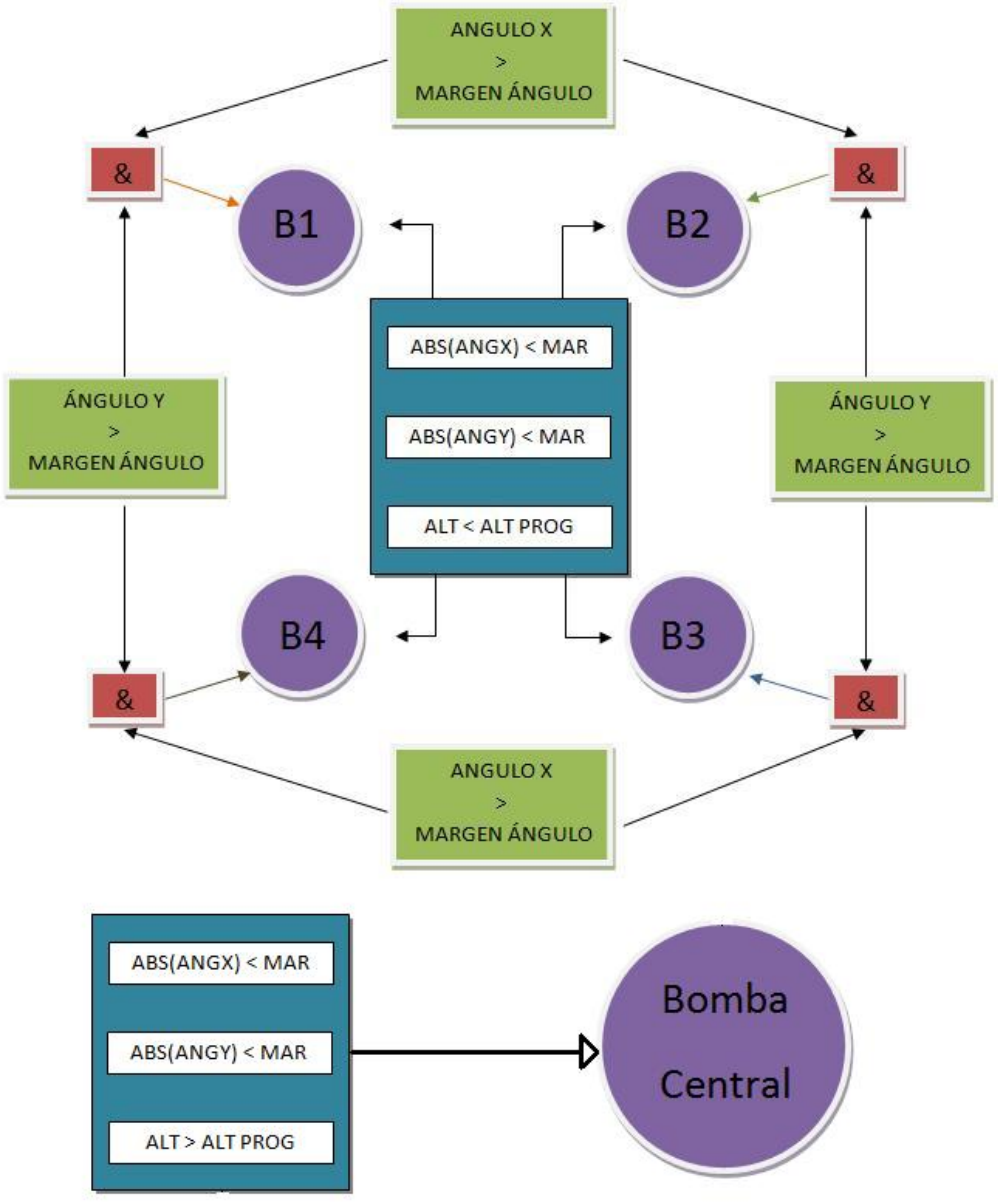


Figura 7.2: Diagrama de flujo 2

Capítulo 8

Programación Processing

En esta sección abordaremos, la programación realizada en Processing, de una manera muy similar a como lo hicimos con la programación de Arduino. Dado que el lenguaje de Arduino está basado en Processing, las variables serán idénticas con una pequeña diferencia a la hora de establecerlas.

En el apéndice C podemos encontrar el código completo utilizado en Processing.

8.1. Librerías/Variables

Las librerías utilizadas son las siguientes:

- **controlP5:** esta librería nos permite crear controles e indicadores (botones, barras, forma circular, ect), de forma práctica y sencilla.
- **serial:** nos permita la comunicación Serial con Arduino u otro componente.
- **opengl:** librería gráfica utilizada para la recreación del modelo en 3D.

```
import controlP5.*;
import processing.serial.*;
import processing.opengl.*;

Serial myPort;

int baudRate = 115200;

ControlP5 cp5;
```

```
Knob myKnobX;
Knob myKnobY;
Slider mySliderP;
Slider mySliderR;
Slider mySliderA;
Slider mySliderB;
Slider mySliderC;
Toggle myButton1;
Toggle myButton2;
Toggle myButton3;
Toggle myButton4;
Toggle myButton5;
Toggle myButton6;
Toggle myButton7;

float angx, angy, h, led0, led1, led2, led3, led4, p;
int draftP, mang, mh, tactive, index = 0, k = 0;
boolean turn = false, mode = false;
boolean pump1 = false, pump2 = false, pump3 = false, pump4 = false,
byte out[] = new byte[11];
```

Funciones utilizadas

- **Knob, Slider, Toggle:** variables propias de la librería *controlP5*, correspondientes a diferentes indicadores.
- **boolean:** variables booleana, es decir, con sólo dos posibles valores *true* o **false**.

8.2. Setup

Como en el caso de Arduino, la función *setup()* es llamada una sola vez cuando arranca el programa y es utilizada para definir las variables como el tamaño de la pantalla, el color de fondo, etc.

```
void setup(){
  size (800,800,OPENGL);
  smooth();
  noStroke();
```

```
  myPort = new Serial(this, "COM1", baudRate);

  cp5 = new ControlP5(this);

  Group g1 = cp5.addGroup("g1")
                .setPosition(25,25)
                .setBackgroundHeight(350)
                .setWidth(200)
                .setBackgroundColor(color(255,50))
                .setLabel("ANGLE")
                ;
  myKnobX = cp5.addKnob("Angle X")
               .setRange(-15,15)
               .setPosition(40,20)
               .setRadius(60)
               .setNumberOfTickMarks(6)
               .setTickMarkLength(5)
               .setViewStyle(2)
               .setGroup(g1)
               ;
  myKnobY = cp5.addKnob("Angle Y")
               .setRange(-15,15)
               .setPosition(40,180)
               .setRadius(60)
               .setNumberOfTickMarks(6)
               .setTickMarkLength(5)
               .setViewStyle(2)
               .setGroup(g1)
               ;

  Group g2 = cp5.addGroup("g2")
                .setPosition(575,25)
                .setBackgroundHeight(350)
                .setWidth(200)
                .setBackgroundColor(color(255,50))
                .setLabel("DRAFT")
                ;
  mySliderP = cp5.addSlider("draftP")
                 .setPosition(25,25)
                 .setSize(40,300)
                 .setRange(20,40)
```

```
                    .setNumberOfTickMarks(11)
                    .setGroup(g2)
                    .setLabel("DRAFT PROGRAMM")
                    ;
  mySliderR = cp5.addSlider("draftR")
                    .setPosition(125,25)
                    .setSize(40,300)
                    .setRange(0,50)
                    .setValue(36.35)
                    .setGroup(g2)
                    .setLabel("DRAFT REAL")
                    ;

  Group g3 = cp5.addGroup("g3")
                  .setPosition(250,25)
                  .setBackgroundHeight(350)
                  .setWidth(300)
                  .setBackgroundColor(color(255,50))
                  .setLabel("PUMPS")
                  ;
  myButton1 = cp5.addToggle("turn")
                  .setPosition(50,25)
                  .setSize(50,25)
                  .setGroup(g3)
                  .setMode(ControlP5.SWITCH)
                  .setLabel("OFF/ON")
                  ;
  myButton2 = cp5.addToggle("mode")
                    .setPosition(175,25)
                    .setSize(50,25)
                    .setGroup(g3)
                    .setMode(ControlP5.SWITCH)
                    .setLabel("AUTO/MANUAL")
                    ;
  myButton3 = cp5.addToggle("pump1")
                       .setPosition(45,100)
                       .setSize(25,25)
                       .setGroup(g3)
                       .setLabel("PUMP 1")
                       ;
  myButton4 = cp5.addToggle("pump2")
```

```
                    .setPosition(85,100)
                    .setSize(25,25)
                    .setGroup(g3)
                    .setLabel("PUMP 2")
                    ;
  myButton5 = cp5.addToggle("pump3")
                    .setPosition(125,100)
                    .setSize(25,25)
                    .setGroup(g3)
                    .setLabel("PUMP 3")
                    ;
  myButton6 = cp5.addToggle("pump4")
                    .setPosition(165,100)
                    .setSize(25,25)
                    .setGroup(g3)
                    .setLabel("PUMP 4")
                    ;
  myButton7 = cp5.addToggle("pump0")
                    .setPosition(205,100)
                    .setSize(25,25)
                    .setGroup(g3)
                    .setLabel("PUMP 0")
                    ;
  mySliderA = cp5.addSlider("mang")
                .setPosition(25,150)
                .setSize(200,10)
                .setRange(0,8)
                .setValue(2)
                .setNumberOfTickMarks(9)
                .setGroup(g3)
                .setSliderMode(Slider.FLEXIBLE)
                .setLabel("MARGIN ANGLE")
                ;
  mySliderB = cp5.addSlider("mh")
                .setPosition(25,200)
                .setSize(200,10)
                .setRange(0,8)
                .setValue(2)
                .setNumberOfTickMarks(9)
                .setGroup(g3)
                .setSliderMode(Slider.FLEXIBLE)
```

```
                    .setLabel("MARGIN DRAFT")
                    ;
  mySliderC = cp5.addSlider("tactive")
                    .setPosition(25,250)
                    .setSize(200,10)
                    .setRange(0,10)
                    .setValue(4)
                    .setNumberOfTickMarks(11)
                    .setGroup(g3)
                    .setSliderMode(Slider.FLEXIBLE)
                    .setLabel("TIME ACTIVE")
                    ;
}
```

Funciones utilizadas

- **size():** define la dimensión de la ventana gráfica en unidades de pixeles.
- **smooth():** mejora la calidad de la imagen evitando el pixelado de las formas.
- **noStroke():** desactiva la vista de las aristas.

8.3. Draw

Similar a la función *loop* en Arduino, la función *draw* en Processing se ejecuta continuamente reproducción las diferentes forma y movimiento que vemos en nuestra pantalla.

```
void draw(){
  background(0);
  lights();

  if (turn == true){
    myButton1.setColorActive(color(255,0,0));
  }else{
    myButton1.setColorActive(color(0,255,0));
  }

  led0 = int(led0*100);
  led1 = int(led1*255);
```

```
led2 = int(led2*255);
led3 = int(led3*255);
led4 = int(led4*255);

myKnobX.setValue(angx);
myKnobY.setValue(angy);
mySliderR.setValue(h);

out[0] = byte(turn);
out[1] = byte(mode);
out[2] = byte(pump1);
out[3] = byte(pump2);
out[4] = byte(pump3);
out[5] = byte(pump4);
out[6] = byte(pump0);
out[7] = byte(draftP);
out[8] = byte(mang);
out[9] = byte(mh);
out[10] = byte(tactive);

if (myPort.available() > 0){
  for (int index = 0; index < 11; index++){
    myPort.write(char(65+index));
    myPort.write(out[index]);
  }
 }

 delay(25);

pushMatrix();
//Plancha
translate(400,450,0);
rotateX(radians(angx));
rotateZ(radians(angy));
fill(100+led0,100-led0,100-led0);
box(210,6,210);

//Cilindro 1
translate(75,78,75);
fill(255-led3,255-led3,255);
cylinder(16.5,150,100);
```

```
  //Cilindro 2
  translate(-150,0,0);
  fill(255-led4,255-led4,255);
  cylinder(16.5,150,100);

  //Cilindro 3
  translate(0,0,-150);
  fill(255-led1,255-led1,255);
  cylinder(16.5,150,100);

  //Cilindro 4
  translate(150,0,0);
  fill(255-led2,255-led2,255);
  cylinder(16.5,150,100);
  popMatrix();

}
```

Funciones utilizadas

- **background():** selecciona el color de fondo.
- **lights():** añade focos, proporcionando al modelo 3D una imagen más realista.
- **Serial.available():** nos proporciona el número de bits que hay almacenados en el puerto serie.
- **Serial.write():** envía datos a través del puerto serie.
- **pushMatrix()/popMatrix():** estas funciones permiten realizar una serie de transformaciones geométricas a uno o varios objetos sin que afecten al resto de figuras. Se debe insertar la función *pushMatrix()* al inicio de las transformaciones, mientras que *popMatrix()* irá ubicada el final.
- **translate():** traslada el origen de coordenadas a las coordenadas establecidas. Es importante destacar que tanto esta función como el resto de transformaciones que vamos a ver quedan grabadas en Processing, es decir, las siguientes transformaciones se realizarán a partir de las anteriores. En este sentido la funciones *pushMatrix()/popMatrix()* son un recurso muy utilizado.

- **rotateX/Y/Z():** rota la imagen según el eje considerado.

- **fill():** establece el color del objeto u objetos de los que vaya precedido.

8.4. Arduino communication

Para la comunicación con Arduino hemos utilizado la función *serialEvent*. Ésta se ejecuta una vez por cada vez que lo hace la función *draw*.

```
void serialEvent(Serial myPort){
  String inStr = myPort.readStringUntil('\n');

  if(inStr!=null){
    inStr = trim(inStr);
    float[] values = float(split(inStr,","));
    if(values.length >= 9){
      angx = values[0];
      angy = values[1];
      h = values[2];
      led0 = values[3];
      led1 = values[4];
      led2 = values[5];
      led3 = values[6];
      led4 = values[7];
    }
  }
}
```

Funciones utilizadas

- **Serial.readStringUntil():** leer la cadena de caracteres enviada por Arduino hasta un valor establecido.

- **trim():** elimina los espacios en blanco enviados por Arduino a través del puerto serie.

- **split():** separa la cadena de caracteres por medio de la designación del carácter que los separa.

8.5. Cylinder draw

La útima función que nos queda por mostrar es *cylinder()*. Aunque no esté recogida en las librerías internas de processing en su página web.

```
void cylinder(float r, float h, int sides){
  float angle;
  float[] x = new float[sides + 1];
  float[] z = new float[sides + 1];

  for(int i=0; i < x.length; i++){
    angle = TWO_PI/(sides)*i;
    x[i] = sin(angle)*r;
    z[i] = cos(angle)*r;
  }
  beginShape(TRIANGLE_FAN);
  vertex(0, -h/2, 0);
  for(int i=0; i < x.length; i++){
    vertex(x[i], -h/2, z[i]);
  }
  endShape();

  beginShape(QUAD_STRIP);
  for(int i=0; i < x.length; i++){
    vertex(x[i], -h/2, z[i]);
    vertex(x[i], h/2, z[i]);
  }
  endShape();

  beginShape(TRIANGLE_FAN);
  vertex(0, h/2, 0);
  for(int i=0; i < x.length; i++){
    vertex(x[i], h/2, z[i]);
  }
  endShape();
}
```

Capítulo 9

Futuras Aplicaciones/Mejoras

En esta sección se tratarán tanto posibles usos de la plataforma, como mejoras que pueden ser implantadas en ella:

9.1. Docencia

El método manual de la plataforma-quad permite que pueda ser utilizada con motivos educacionales, por ejemplo como prácticas de Hidrostática.

En ellas, podrían calcularse de manera experimental distintos parámetros que intervienen en la estabilidad de un flotador, como por ejemplo la altura metacéntrica.

9.2. Mesa para piscina

Nuestra plataforma al fin y al cabo es una mesa que si contase con la posibilidad de achicar y rellenar rápidamente sería virtualmente involcable.

Por motivos industriales no parece cercano el desarrollo de este producto, ya que fabricar plataformas como la nuestra pero con mejores dispositivos y electrónica, elevaría el precio de la misma a un nivel fuera de mercado.

No obstante, no nos parece una idea descabellada si en el proceso industrial se optimizasen suficiente los gastos.

9.3. Plataformas petrolíferas

Pese a no ser un campo que dominemos en gran medida, se intuye que es posible que tenga alguna aplicación en las mismas.

9.4. Wi-Fi

La integración de un módulo Wi-Fi en lugar del módulo Bluetooth permitiría comunicaciones más rápidas y a mayor distancia.

9.5. Telecontrol

En la actualidad, desde nuestros dispositivos móviles(tanto IOS como ANDROID OS) podemos realizar las lecturas del acelerómetro y del diferencial de presión, pero no enviar ordenes a la plataforma-quad.

La creación de una interfaz cómoda y práctica para sistema operativo Android, Bada o IOS, junto con la instalación de un receptor en el flotador, nos permitiría que cuando la plataforma esté en modo manual en vez de automático, podamos enviarle los valores de los parámetros deseados, de modo que controlaríamos la plataforma desde nuestros dispositivos móviles.

9.6. Autonomía

Aunque la plataforma-quad es autoadrizante, requiere de tomas de corriente, dado que no cuenta con ninguna batería ni fuente de energía propia.

La instalación de una batería permitiría que la plataforma no solo tuviese la opción automática, si no que se convirtiría en un flotador autónomo.

Capítulo 10

Bibliografía

10.1. Libros

- Teoría de las Estructuras - *Timoshenko* - 1965.

- Teoría del buque. Flotabilidad y estabilidad - *Joan Olivella Puig* - Edicions UPC 1994

- Mecánica de Fluidos General - *Manuel M. Sánchez Nieto* - Universidad Politécnica de Cartagena 2007

- Electrínica - *Hambley Allan R.* - Thomson 2005

10.2. Artículos

- Flotador de Calado Autoregulable/Constante: Estructura - *Alejandro Macanás Vidal* 2013

10.3. Webs

- http://www.arduino.cc/

- http://www.processing.org/

- http://www.seeedstudio.com/wiki/Grove_-_Relay

- http://www.seeedstudio.com/wiki/Grove_-_Serial_Bluetooth

- http://www.seeedstudio.com/wiki/images/2/25/Bluetooth_Software_Instruction.pdf

- http://www.ni.com/academic/why_labview/esa/
- http://www.cadsoftusa.com/?language=en
- http://www.cursomicros.com/avr/bus-i2c/protocolo-bus-i2c.html
- http://wiki.processing.org/w/Cylinder
- http://es.wikipedia.org/wiki/Relé
- http://es.wikipedia.org/wiki/LabVIEW
- http://es.wikipedia.org/wiki/Processing
- http://es.wikipedia.org/wiki/EAGLE

10.4. Software

- Autocad.
- Rhinoceros.
- SolidWorks.
- Processing.
- Ansys.
- Excell.
- R.
- Excell.
- Latex.
- Worktex.
- TeXstudio.
- Bluetooth SPP.
- Eagle.
- Dropbox.
- Catch.

- CamStudio.
- VideoPad.

Apéndice A

Código Arduino I

```
#include <SoftwareSerial.h>
#include <math.h>

#define RxD 2
#define TxD 3

SoftwareSerial blueTS(RxD, TxD);

const int n = 50;

int rx[n], ry[n], rz[n], rh[n];
int tx = 0, ty = 0, tz = 0;
int ax = 0, ay = 0, az = 0;
int i = 0, j = 0;
byte m[5] = {0,0,0,0,0}, q[4] = {0,0,0,0}, pump[5] = {0,0,0,0,0}, var;
float angx, angy, h, th = 0, ah = 0;
int mang = 2, mh = 2, hp = 35, turn = 0, mode = 0, tactive = 4;
int input[11];

const int xpin = A3;
const int ypin = A2;
const int zpin = A1;
const int hpin = A4;

void setup(){
  Serial.begin(115200);
  pinMode(RxD, INPUT);
```

```
  pinMode(TxD, OUTPUT);
  setupblueTS();
  delay(15000);
  pinMode(13, OUTPUT);
  pinMode(11, OUTPUT);
  pinMode(9, OUTPUT);
  pinMode(7, OUTPUT);
  pinMode(5, OUTPUT);
  for (int k = 0; k > n; k++){
    rx[k] = 0;
    ry[k] = 0;
    rz[k] = 0;
    rh[k] = 0;
  }
}

void loop(){
  tx = tx - rx[i];
  ty = ty - ry[i];
  tz = tz - rz[i];
  th = th - rh[i];

  rx[i] = analogRead(xpin);
  ry[i] = analogRead(ypin);
  rz[i] = analogRead(zpin);
  rh[i] = analogRead(hpin);

  tx = tx + rx[i];
  ty = ty + ry[i];
  tz = tz + rz[i];
  th = th + rh[i];

  ax = map(tx/n,406,271,100,-100);
  ay = map(ty/n,399,264,100,-100);
  az = map(tz/n,412,280,100,-100);
  ah = (th/n*17.774*5/1024-27.64);

  angx = atan2(ax,az)*180/PI;
  angy = atan2(-ay,az)*180/PI;

  blueTS.print(angx); blueTS.print(",");
```

```
  blueTS.print(angy); blueTS.print(",");
  blueTS.print(ah); blueTS.print(",");
  for (int i = 0; i < 5; i++){
    blueTS.print(m[i]); blueTS.print(",");
  }
  blueTS.println();

  i = i + 1;

  if (i >= n){
    i = 0;
    j = j + 1;
  }

  getProcessing();

  if (mode == 1){
    automode();
  }else{
    manualmode();
  }

  delay(10);
}

void automode(){
 if (j == 4){
   if (angx > mang && angy < mang || angy < -mang && angx > -mang){
     digitalWrite(13, HIGH);
     m[1] = 1; q[1] = q[1] + 1;
   }else{
     m[1] = 0; q[1] = 0;
   }
   if (angx > mang && angy > -mang || angy > mang && angx > -mang){
     digitalWrite(11, HIGH);
     m[2] = 1; q[2] = q[2] + 1;
   }else{
     m[2] = 0; q[2] = 0;
   }
   if (angx < -mang && angy > -mang || angy > mang && angx < mang){
     digitalWrite(9, HIGH);
```

```
  m[3] = 1; q[3] = q[3] + 1;
}else{
  m[3] = 0; q[3] = 0;
}
if (angx < -mang && angy < mang || angy < -mang && angx < mang){
  digitalWrite(7, HIGH);
  m[4] = 1; q[4] = q[4] + 1;
}else{
  m[4] = 0; q[4] = 0;
}
if (q[1] > 8 || q[2] > 8 || q[3] > 8 || q[4] > 8){
  digitalWrite(5, HIGH);
  m[0] = 1;
}else{
  m[0] = 0;
}

if (abs(angx) < mang && abs(angy) < mang){
  if (ah > (hp + mh)){
    digitalWrite(13, HIGH);
    digitalWrite(11, HIGH);
    digitalWrite(9, HIGH);
    digitalWrite(7, HIGH);
    m[1] = m[2] = m[3] = m[4] = 1;
  }
  if (ah < (hp - mh)){
    digitalWrite(5, HIGH);
    m[0] = 1;
  }else{
    m[0] = 0;
  }
}

blueTS.print(angx); blueTS.print(",");
blueTS.print(angy); blueTS.print(",");
blueTS.print(ah); blueTS.print(",");
for (int i = 0; i < 5; i++){
  blueTS.print(m[i]); blueTS.print(",");
}
blueTS.println();
```

```
    delay(tactive*1000);
    digitalWrite(13, LOW);
    digitalWrite(11, LOW);
    digitalWrite(9, LOW);
    digitalWrite(7, LOW);
    digitalWrite(5, LOW);

    j = 0;
  }
}

void manualmode(){
  if (pump[1] == 1){
    digitalWrite(13, HIGH);
    m[1] = 1;
  }else{
    digitalWrite(13, LOW);
    m[1] = 0;
  }
  if (pump[2] == 1){
    digitalWrite(11, HIGH);
    m[2] = 1;
  }else{
    digitalWrite(11, LOW);
    m[2] = 0;
  }
  if (pump[3] == 1){
    digitalWrite(9, HIGH);
    m[3] = 1;
  }else{
    digitalWrite(9, LOW);
    m[3] = 0;
  }
  if (pump[4] == 1){
    digitalWrite(7, HIGH);
    m[4] = 1;
  }else{
    digitalWrite(7, LOW);
    m[4] = 0;
  }
  if (pump[0] == 1){
```

```
    digitalWrite(5, HIGH);
    m[0] = 1;
  }else{
    digitalWrite(5, LOW);
    m[0] = 0;
  }

  j = 0;
}

int getProcessing(){
  while (blueTS.available()){
    var = blueTS.read();
    switch(var){
      case 'A': var = blueTS.read();
        if (var < 50) turn = var;
      case 'B': var = blueTS.read();
        if (var < 50) mode = var;
      case 'C': var = blueTS.read();
        if (var < 50) pump[1] = var;
      case 'D': var = blueTS.read();
        if (var < 50) pump[2] = var;
      case 'E': var = blueTS.read();
        if (var < 50) pump[3] = var;
      case 'F': var = blueTS.read();
        if (var < 50) pump[4] = var;
      case 'G': var = blueTS.read();
        if (var < 50) pump[0] = var;
      case 'H': var = blueTS.read();
        if (var < 50) hp = var;
      case 'I': var = blueTS.read();
        if (var < 50) mang = var;
        Serial.println(mang);
      case 'J': var = blueTS.read();
        if (var < 50) mh = var;
      case 'K': var = blueTS.read();
        if (var < 50) tactive = var;
    }
  }
}
```

```
void setupblueTS(){
  blueTS.begin(115200);
  delay(1000);
  blueTS.print("\r\n+STWMOD=0\r\n");
  blueTS.print("\r\n+STNA=SeeedBTSlave\r\n");
  blueTS.print("\r\n+DLPIN\r\n");
  blueTS.print("\r\n+STOAUT=1\r\n");
  blueTS.print("\r\n+STAUTO=0\r\n");
  delay(2000);
  blueTS.print("\r\n+INQ=1\r\n");
  delay(2000);
  blueTS.flush();
}
```

Apéndice B

Código Arduino II

```
#include <SoftwareSerial.h>
#include <Wire.h>
#include <math.h>
#include "MMA7660.h"
MMA7660 accelemeter;

#define RxD 2
#define TxD 3

SoftwareSerial blueTS(RxD, TxD);

const int n = 20;
int rx[n]; int ry[n]; int rz[n]; int rh[n];
int i = 0;
int tx = 0; int ty = 0; int tz = 0; int th = 0;
int m[5];
float h = 0;
float Angx;
float Angy;
float hp = 40;
int L = 3;
float p = 1;

void setup()
{
  accelemeter.init();
  Serial.begin(115200);
```

```
  pinMode(RxD, INPUT);
  pinMode(TxD, OUTPUT);
  setupblueTC();
  delay(15000);
  pinMode(13, OUTPUT);
  pinMode(11, OUTPUT);
  pinMode(9, OUTPUT);
  pinMode(7, OUTPUT);
  pinMode(5, OUTPUT);
  for (int m = 0; m < n; m++){
    rx[m] = 0;
    ry[m] = 0;
    rz[m] = 0;
    rh[m] = 0;
  }
}
void loop()
{
  int8_t x;
  int8_t y;
  int8_t z;

  tx = tx - rx[i];
  ty = ty - ry[i];
  tz = tz - rz[i];
  th = th - rh[i];

  accelemeter.getXYZ(&x,&y,&z);
  rx[i] = x;
  ry[i] = y;
  rz[i] = z;
  rh[i] = analogRead(A1);

  tx = tx + rx[i];
  ty = ty + ry[i];
  tz = tz + rz[i];
  th = th + rh[i];

  i = i + 1;

  if (i >= n)
```

```
  i = 0;

h = th/n*17.774*5/1024-27.64;
Angx = atan2(tx,tz)*180/PI;
Angy = atan2(ty,tz)*180/PI;

if(blueTS.available()){
  blueTS.print(Angx); blueTS.print(",");
  blueTS.print(Angy); blueTS.print(",");
  blueTS.print(h); blueTS.print(",");
  for (i = 0; i < 5; i++){
    blueTS.print(m[i]); blueTS.print(",");
  }
  blueTS.println();
}

if (Angx > L && Angy < L || Angy < -L && Angx > -L
|| abs(Angx) < L && abs(Angy) < L && h < (hp - p)){
  digitalWrite(13, HIGH);
  m[1] = 1;
}else{
  digitalWrite(13, LOW);
  m[1] = 0;
}
if (Angx > L && Angy > -L || Angy > L && Angx > -L
|| abs(Angx) < L && abs(Angy) < L && h < (hp - p)){
  digitalWrite(11, HIGH);
  m[2] = 1;
}else{
  digitalWrite(11, LOW);
  m[2] = 0;
}
if (Angx < -L && Angy > -L || Angy > L && Angx < L
|| abs(Angx) < L && abs(Angy) < L && h < (hp - p)){
  digitalWrite(9, HIGH);
  m[3] = 1;
}else{
  digitalWrite(9, LOW);
  m[3] = 0;
}
if (Angx < -L && Angy < L || Angy < -L && Angx < L
```

```
  || abs(Angx) < L && abs(Angy) < L && h < (hp - p)){
    digitalWrite(7, HIGH);
    m[4] = 1;
  }else{
    digitalWrite(7, LOW);
    m[4] = 0;
  }

  if (abs(Angx) < L && abs(Angy) < L && h > (hp + p)){
    digitalWrite(5, HIGH);
    m[0] = 1;
  }else{
    digitalWrite(5, LOW);
    m[0] = 0;
  }

  delay(100);
}

void setupblueTC(){
  blueTS.begin(9600);
  blueTS.print("\r\n+STWMOD=0\r\n");
  blueTS.print("\r\n+STNA=SeeedBTSlave\r\n");
  blueTS.print("\r\n+DLPIN\r\n");
  blueTS.print("\r\n+STOAUT=1\r\n");
  blueTS.print("\r\n+STAUTO=0\r\n");
  delay(2000);
  blueTS.print("\r\n+INQ=1\r\n");
  Serial.println("The slave bluetooth is inquirable!");
  delay(2000);
  blueTS.flush();
}
```

Apéndice C

Código Processing

```
import controlP5.*;
import processing.serial.*;
import processing.opengl.*;

Serial myPort;

int baudRate = 115200;

ControlP5 cp5;

Knob myKnobX;
Knob myKnobY;
Slider mySliderP;
Slider mySliderR;
Slider mySliderA;
Slider mySliderB;
Slider mySliderC;
Toggle myButton1;
Toggle myButton2;
Toggle myButton3;
Toggle myButton4;
Toggle myButton5;
Toggle myButton6;
Toggle myButton7;

float angx, angy, h, led0, led1, led2, led3, led4, p;
int draftP, mang, mh, tactive, index = 0, k = 0;
```

```
boolean turn = false, mode = false;
boolean pump1 = false, pump2 = false, pump3 = false, pump4 = false,
byte out[] = new byte[11];

void setup(){
  size (800,800,OPENGL);
  smooth();
  noStroke();

  myPort = new Serial(this, "COM1", baudRate);

  cp5 = new ControlP5(this);

  Group g1 = cp5.addGroup("g1")
                .setPosition(25,25)
                .setBackgroundHeight(350)
                .setWidth(200)
                .setBackgroundColor(color(255,50))
                .setLabel("ANGLE")
                ;
  myKnobX = cp5.addKnob("Angle X")
               .setRange(-15,15)
               .setPosition(40,20)
               .setRadius(60)
               .setNumberOfTickMarks(6)
               .setTickMarkLength(5)
               .setViewStyle(2)
               .setGroup(g1)
               ;
  myKnobY = cp5.addKnob("Angle Y")
               .setRange(-15,15)
               .setPosition(40,180)
               .setRadius(60)
               .setNumberOfTickMarks(6)
               .setTickMarkLength(5)
               .setViewStyle(2)
               .setGroup(g1)
               ;

  Group g2 = cp5.addGroup("g2")
                .setPosition(575,25)
```

```
                .setBackgroundHeight(350)
                .setWidth(200)
                .setBackgroundColor(color(255,50))
                .setLabel("DRAFT")
                ;
mySliderP = cp5.addSlider("draftP")
                .setPosition(25,25)
                .setSize(40,300)
                .setRange(20,40)
                .setNumberOfTickMarks(11)
                .setGroup(g2)
                .setLabel("DRAFT PROGRAMM")
                ;
mySliderR = cp5.addSlider("draftR")
                .setPosition(125,25)
                .setSize(40,300)
                .setRange(0,50)
                .setValue(36.35)
                .setGroup(g2)
                .setLabel("DRAFT REAL")
                ;

Group g3 = cp5.addGroup("g3")
                .setPosition(250,25)
                .setBackgroundHeight(350)
                .setWidth(300)
                .setBackgroundColor(color(255,50))
                .setLabel("PUMPS")
                ;
myButton1 = cp5.addToggle("turn")
                .setPosition(50,25)
                .setSize(50,25)
                .setGroup(g3)
                .setMode(ControlP5.SWITCH)
                .setLabel("OFF/ON")
                ;
myButton2 = cp5.addToggle("mode")
                .setPosition(175,25)
                .setSize(50,25)
                .setGroup(g3)
                .setMode(ControlP5.SWITCH)
```

```
                .setLabel("AUTO/MANUAL")
                ;
myButton3 = cp5.addToggle("pump1")
                .setPosition(45,100)
                .setSize(25,25)
                .setGroup(g3)
                .setLabel("PUMP 1")
                ;
myButton4 = cp5.addToggle("pump2")
                .setPosition(85,100)
                .setSize(25,25)
                .setGroup(g3)
                .setLabel("PUMP 2")
                ;
myButton5 = cp5.addToggle("pump3")
                .setPosition(125,100)
                .setSize(25,25)
                .setGroup(g3)
                .setLabel("PUMP 3")
                ;
myButton6 = cp5.addToggle("pump4")
                .setPosition(165,100)
                .setSize(25,25)
                .setGroup(g3)
                .setLabel("PUMP 4")
                ;
myButton7 = cp5.addToggle("pump0")
                .setPosition(205,100)
                .setSize(25,25)
                .setGroup(g3)
                .setLabel("PUMP 0")
                ;
mySliderA = cp5.addSlider("mang")
                .setPosition(25,150)
                .setSize(200,10)
                .setRange(0,8)
                .setValue(2)
                .setNumberOfTickMarks(9)
                .setGroup(g3)
                .setSliderMode(Slider.FLEXIBLE)
                .setLabel("MARGIN ANGLE")
```

```
                    ;
  mySliderB = cp5.addSlider("mh")
                    .setPosition(25,200)
                    .setSize(200,10)
                    .setRange(0,8)
                    .setValue(2)
                    .setNumberOfTickMarks(9)
                    .setGroup(g3)
                    .setSliderMode(Slider.FLEXIBLE)
                    .setLabel("MARGIN DRAFT")
                    ;
  mySliderC = cp5.addSlider("tactive")
                    .setPosition(25,250)
                    .setSize(200,10)
                    .setRange(0,10)
                    .setValue(4)
                    .setNumberOfTickMarks(11)
                    .setGroup(g3)
                    .setSliderMode(Slider.FLEXIBLE)
                    .setLabel("TIME ACTIVE")
                    ;
}

void draw(){
  background(0);
  lights();

  if (turn == true){
    myButton1.setColorActive(color(255,0,0));
  }else{
    myButton1.setColorActive(color(0,255,0));
  }

  led0 = int(led0*100);
  led1 = int(led1*255);
  led2 = int(led2*255);
  led3 = int(led3*255);
  led4 = int(led4*255);

  myKnobX.setValue(angx);
  myKnobY.setValue(angy);
```

```
mySliderR.setValue(h);

out[0] = byte(turn);
out[1] = byte(mode);
out[2] = byte(pump1);
out[3] = byte(pump2);
out[4] = byte(pump3);
out[5] = byte(pump4);
out[6] = byte(pump0);
out[7] = byte(draftP);
out[8] = byte(mang);
out[9] = byte(mh);
out[10] = byte(tactive);

if (myPort.available() > 0){
  for (int index = 0; index < 11; index++){
    myPort.write(char(65+index));
    myPort.write(out[index]);
  }
 }

 delay(25);

pushMatrix();
//Plancha
translate(400,450,0);
rotateX(radians(angx));
rotateZ(radians(angy));
fill(100+led0,100-led0,100-led0);
box(210,6,210);

//Cilindro 1
translate(75,78,75);
fill(255-led3,255-led3,255);
cylinder(16.5,150,100);

//Cilindro 2
translate(-150,0,0);
fill(255-led4,255-led4,255);
cylinder(16.5,150,100);
```

```
  //Cilindro 3
  translate(0,0,-150);
  fill(255-led1,255-led1,255);
  cylinder(16.5,150,100);

  //Cilindro 4
  translate(150,0,0);
  fill(255-led2,255-led2,255);
  cylinder(16.5,150,100);
  popMatrix();

}

void serialEvent(Serial myPort){
  String inStr = myPort.readStringUntil('\n');

  if(inStr!=null){
    inStr = trim(inStr);
    float[] values = float(split(inStr,","));
    if(values.length >= 9){
      angx = values[0];
      angy = values[1];
      h = values[2];
      led0 = values[3];
      led1 = values[4];
      led2 = values[5];
      led3 = values[6];
      led4 = values[7];
    }
  }
}

void cylinder(float r, float h, int sides){
  float angle;
  float[] x = new float[sides + 1];
  float[] z = new float[sides + 1];

  for(int i=0; i < x.length; i++){
    angle = TWO_PI/(sides)*i;
    x[i] = sin(angle)*r;
    z[i] = cos(angle)*r;
```

```
  }
  beginShape(TRIANGLE_FAN);
  vertex(0, -h/2, 0);
  for(int i=0; i < x.length; i++){
    vertex(x[i], -h/2, z[i]);
  }
  endShape();

  beginShape(QUAD_STRIP);
  for(int i=0; i < x.length; i++){
    vertex(x[i], -h/2, z[i]);
    vertex(x[i], h/2, z[i]);
  }
  endShape();

  beginShape(TRIANGLE_FAN);
  vertex(0, h/2, 0);
  for(int i=0; i < x.length; i++){
    vertex(x[i], h/2, z[i]);
  }
  endShape();
}
```