

**Figure x.** A diagram depicting the proposed robot assembly for Prototype 1, and the software required for the functioning of the system on a Windows® PC.

### Proposed Assembly

All *‘cognitive’* processing (i.e. processing with regards to ‘*the cognition system’* viz. intelligent navigation and decision-making) will be accomplished solely on PC using the software set-up depicted in figure 1. The PC is the *client* as it executes the *client* application (*namely* ‘AVINSoR client’) whose function is to controlall other aspects of the system through commands (hardware *and* software).

The robot assembly simply acts as multifaceted *server* – providing sensory input (some of which are processed through the NXT brick) and allowing control of the actuator motors (through NXT brick’s signal processing capabilities). The operation of the NXT brick is configured and instructed through commands. [[1]](#footnote-1)

The purpose of Prototype 1 is to **(1)** assemble the robot and establish connectivity and **(2)** develop the fundamental application (AVINSoR Client) to a reasonable standard so that (further) development of the *‘cognitive’* processes can be pursued with ease in later prototypes. The AVINSoR Client will be developed with maximum *modularity* and *reusability* in mind, effective use of object-orientation will naturally allow easily identifiable code-objects/software-components designed for minimal coupling and maximum cohesion.

#### Hardware

With regards to hardware, nothing is developed. The LEGO® Mindstorms® NXT Kit is a modular hardware kit that will allow the control of attached motors, and the reception of digital data from the attached sensors, with absolutely no hardware development.

The webcam connection to the PC is completely independent of the NXT. The possibility of implementing wireless connectivity between the webcam and the PC was omitted for a later prototype – as time was prioritized for the development of AVINSoR Client – which will simply be referred to as the *software application* or the *client application* throughout this report).

The **sensors** of the AVINSoR Prototype 1 architecture are:

* The Webcam – visual data(intensity of red light, green light and blue light - per pixel)*. [[2]](#footnote-2)*
* The Ultrasonic Sensor – distance of objects.
* The Light Sensor – light intensity.
* The Sound Sensor – sound pressure.

The description associated with each sensor’s purpose is very basic and over-simplified. Ultimately, many things can be deduced from the data received from each sensor, especially when data is *correlated with other sensors* through a *trainable* pattern recognition algorithm.

Note that each sensor may have an angle limit, distance limit, or finite coverage area, out of which the sensor will *not* respond to *detect* the stimuli (due to obvious physical and mechanical reasons). This is the case with humans and animals as much as it is with electronic robots.

The **actuators** of the architecture are:

* Left (Servo) Motor  
  - controls the rotation of the left wheel.
* Right (Servo) Motor  
  - controls the rotation of the right wheel.

#### Software

**Drivers** willallow the operating system (Microsoft® Windows®) to detect and communicate with the hardware.Without the correct compatible drivers, hardware will not be recognized by the operating system (whose function is to provide *hardware abstraction*), and therefore applications will not be able to communicate with the hardware. Thankfully, all necessary drivers to communicate with the NXT Brick via Bluetooth® are provided by the manufacturer (LEGO®) and Microsoft® Windows®. The necessary drivers to communicate with the webcam are provided by the manufacturer (Logitech®) and Microsoft® (UVC webcam drivers are typically innate in Microsoft® Windows® installations).

**MATLAB** is a numerical computing environment widely used by engineers and scientists.

##### AVINSoR Client

This is fundamentally the entire software aspect of the AVINSoR system packaged into a single application.

The **client software** functions to:

* Establish communication with the NXT in order to send commands and receive data.
* Provide a GUI (Graphical User Interface):
  + *Get input* from the human user, in order to:
    1. Train the system to deduce useful information from the array of sensors (digital sensor data).
    2. Train the system to produce precise movement patterns.
  + *Show data* so that user is able to make smarter decisions with regards to *training* the robot. This feature is also *essential* for *testing* and *diagnosis* - during development *and* when in use by the end-user. Such data will include:
    1. Polled *sensor data*, and other NXT variables such as battery level.
    2. The *state* or *result* of any *‘cognitive’* process. This typically includes any machine learning algorithm for the purpose of emulating cognition in humans. This pattern classification, decision-making, object detection/recognition/tracking algorithm (wether *supervised, unsupervised or reinforced)*.
* Communicate with an instance of MATLAB® in order to initiate any complex mathematical processes, or to utilize the functionality of many useful toolboxes and libraries that may be available. This will involve:
  + *Execution* of functions through commands.
  + Transfer of data over to the server.  
    Passing-over of any relevant data to the server, *typically* through the reconstruction of data via (potentially iterative) formulation and execution of command strings.
  + *Retrieval of data from the server.*

#### Robot Assembly

##### Logitech® C270HD Webcam

|  |
| --- |
| http://www.logitech.com/assets/30032/2/logitech-hd-webcam-c260.png |
| Image x. The Logitech C270 webcam. [6] |

The webcam’s function is to provide visual input to the *cognition system*. In Prototype 1 this is a snapshot (i.e. a single image, captured on request). However, in later prototypes, when perception and cognition systems, as well as I/O communication with the NXT assembly has been refined to a standard capable of real-time processing, a continuous 30fps video stream may be used.

|  |  |
| --- | --- |
| Category | Specification |
| Connectivity | Hi-Speed USB 2.0 |
| Resolutions |  |
| Maximum Frame Rate |  |
|  | 30 Frames Per Second |

List of resolutions Table x. Summary of the Logitech® C270’s specification. [6]

**The Logitech C270HD was specifically sourced (over often cheaper alternatives) for the following reasons:** [1]

1. Logitech’s reputation to abide by the UVC device-class specifications (laid out by the USB committee in the relevant USB 2.0 documentation) in order to allow uniformity in communication and hence, compatibility with different software applications. REF.  
     
   This will allow us to capture frames directly from MATLAB®, provided that *MATLAB Support Package for USB webcams* is installed.
2. Logitech’s reputation to provide DirectShow® compliant WDM (Windows® Driver Model) drivers. This is a prerequisite if the device is to be used with MATLAB’s Image Acquisition Toolbox™.   
     
   The Image Acquisition Toolbox™ is a widely used toolbox, its functions often utilized by other MATLAB® libraries and toolboxes (particularly those *not* authored by MathWorks®). We are particularly interested in libraries that allow image processing and analysis. [2]  
     
   DirectShow® is essentially an API that allows applications to easily implement the functionality of high-quality AV capture and feedback. [[3]](#footnote-3) Apart from official DirectShow samples from Microsoft®, many open-source software libraries exist which further encapsulate the DirectShow API to allow an *even more effortless* method of implementing AV functionality into applications through many different programming languages and environments. One such library of interest to our C# client application may be DirectShow.NET – in case MATLAB® proves to be unsuitable, or capturing directly through the client application proves to be more effective for a given purpose. [3] [4] [5]  
     
   Thus, for the purpose of *versatility in development* and *upgradability* – especially as the course of rapid prototyping continues to improve upon this project, DirectShow® compatibility cannot be considered an option.

##### NXT Kit Assembly

The advantage of using the NXT framework over, say, a development board or a single board computer, is that it eliminates the need for creating custom hardware/software interfacing that would be necessary to control the actuators (servo motors), and get readings from various different types of analogue and digital sensors. Such tasks may sound trivial but developing a system that would function to the *equivalent* specification mentioned above (under ‘The LEGO® Mindstorms NXT 1.0 Kit’) *and* provide the equivalent level of encapsulation and concealment of low level hardware and software detail so that focus may be retained on the development of the *cognition* aspects (the *main* scope of this project), can be considered *unnecessarily risky* in a project with a limited time budget (such as this). To support this statement, it is worth mentioning that even the development of Prototype 1 extended *beyond* the time allocated for it - due to unforeseen technical issues requiring micro-iterations of design, development and testing. (refer to explained time allocation).

Essentially, the **NXT Brick** acts as a software *server* with its own (limited) processing ability. The server functions to interpret commands sent from the PC *(the client)* in order to **(1)** process the control signals necessary to accurately control the *actuators* connected, and **(2)** poll the connected *sensors* and transmit the data back to the client PC (upon request, *or* on specified time interval).

The components labelled ‘Left Wheel’ and ‘Right Wheel’ are **NXT Servo Motor Modules** connected to the Brick’s output ports, in the same manner as indicated. This will allow differential wheel steering in the robot. (See Appendix for different methods of steering in robotic vehicles).

The component labelled ‘Ultrasonic Sensor’ is an **NXT Ultrasonic Sensor** module, this module, through the NXT Brick, essentially produces an 8-bit digital signal ranging from 0 to 255 indicative of the distance (in centimeters) of the object within its range.

The component labelled ‘Light Sensor’ is an **NXT Light Sensor** module, this module, through the NXT brick, produces a normalized digital signal ranging from 0 to 100 (*percent*) indicative of the light intensity captured (within the phototransistors response angle – its response angle is explained later). The light sensor also features a red LED, which can be switched on. When the LED is switched on, the sensor functions to measures *reflected light* – which can be a good indicator of distance from the surface of the object ahead. When the LED is switched off, the sensor functions to measure *ambient light* (from the atmosphere).

The component labelled ‘Sound Sensor’ is an **NXT Sound Sensor** module, this module, through the NXT brick, produces a normalized digital signal ranging from 0 to 100 (*percent*) indicative of sound pressure. The NXT brick can be configured to normalize the signal in accord to either (1) the *‘dBA range’ or* (2) the *‘dB range’*. In the *typical* ‘dB’ mode the NXT Brick produces a signal that is equally sensitive to *all* frequencies (even audible frequencies). In ‘dBA’ mode, the signal produced is representative of only the frequencies sensitive to the human ear. [7] [8]

The sound sensor is capable of measuring sound pressure up to 90 decibels, approximately the equivalent sound intensity of a diesel truck 10 meters away, or a lawn mower close by. [7] [11] [12]

The LEGO® Mindstorms® NXT 1.0 Kit is explained in greater detail under **The LEGO® Mindstorms® NXT 1.0 Kit**.

### Hardware: The LEGO® Mindstorms® NXT 1.0 Kit

The AVINSoR project utilizes the LEGO® Mindstorms® NXT, a modular programmable robotics kit. The kit will simply be referred to as the *hardware framework* or *the NXT* throughout this report. The relevant technical details of the NXT will be discussed here forth[[4]](#footnote-4).

#### The Brick

The NXT is based around an encapsulated battery-powered processing device called the *‘brick’*, which features a monochrome LCD and 4 tactile buttons*.* *Sensor* and *actuator* modules can be attached to the *NXT* *brick* via the 4 *input ports* and 3 *output ports* (respectively).

|  |
| --- |
| http://cache.lego.com/e/dynamic/is/image/LEGO/9841?$main$ |
| Figure x. The NXT brick. Image courtesy of Lego® shop. |

##### Hardware

|  |
| --- |
|  |
| Figure x. A block diagram overview of the NXT brick, as featured on the Lego® Mindstorms® NXT Hardware Development Kit. |

See appendix x for full schematic of the NXT brick.

###### Processing

As illustrated in figure x, the NXT brick features two microprocessors. A *main processor*, and a *co-processor.*

**Main Processor**

Ultimately controls all *user-specific* functionality. *Executes the Virtual Machine.* Low-level activities are outsourced to the co-processor.

|  |  |
| --- | --- |
| Processor Name: | AT91SAM7S256 |
| Manufacturer: | Atmel® |
| Architecture: | 32-bit ARM® (*featuring* ARM7TDMI *CPU*) |
| Flash: | 256 kilobytes |
| RAM: | 64 kilobytes |
| Clock Speed: | 48 MHz |

Table 1. The basic specifications of the main processor. [1, p. 3]

**Co-Processor**

|  |  |
| --- | --- |
| Processor Name: | ATmega48 |
| Manufacturer: | Atmel® |
| Architecture: | 8-bit Atmel® AVR |
| Flash: | 4 kilobytes |
| RAM: | 512 bytes |
| Clock Speed: | 8 MHz |

Table 2. The basic specifications of the co-processor. [1, p. 3]

The co-processor is dedicated to the following tasks: [1, pp. 17-18]

* Power Management (Controlling distribution of power throughout the board, *including* the main processor).
* Creating PWM output signals for the three motor ports.
* Performing analogue-to-digital conversion on the analogue sensor data that may be present at the four input ports.

**Main Processor – Co-Processor communication**

An I2C bus allows the main-processor to communicate with the co-processor. Note that due to hardware limitations, the main-processor can only act as *master* within the I2C communication setup, and hence it is responsible for *initiating* any transmission or reception activity between itself and the co-processor in order to exchange data. The I2C bus is set up (in both MCUs) to communicate at 380Kbps. [1, p. 18]

Two section of memory are allocated in each microprocessor and updated every 2 milliseconds. This allows both processors to act (largely) independently. Both sections of memory reserved for communication protocol (in each MCU) are representations of a struct in the embedded C programming language: [1, pp. 18-20]

**IOTOAVR** essentially represents the section of memory in each microcontroller reserved for data sent *from* the main processor *to* the coprocessor.

**typedef** struct

**{**

UBYTE Power**;**

UBYTE PwmFreq**;**

SBYTE PwmValue**[**NOS\_OF\_AVR\_OUTPUTS**];**

UBYTE OutputMode**;**

UBYTE InputPower**;**

**}**

IOTOAVR**;**

|  |  |  |
| --- | --- | --- |
| Variable Name | Type (native range) | Description |
| Power | Unsigned Byte  (0 to 255) | Set to ‘0x00’ during normal communication; is only modified during *power down* and *firmware update*. |
| PwmFreq | Unsigned Byte  (0 to 255) | The pulse width modulation *frequency* of the three motor outputs. The LEGO® firmware typically uses a value of 8 (representing 8 KHz), but the acceptable range is from 0 (KHz) to 32 (KHz). |
| PwmValue[2] | Array of  Signed Byte  (-128 to +127) | The pulse width modulation *value* of each of the three motor outputs.   |  |  | | --- | --- | | Array Index (#) | Output | | 0 | Motor A | | 1 | Motor B | | 2 | Motor C |   The acceptable range is from -100 to +100.  The *magnitude* represents the output power intensity (as a *percentage*).  The polarity represents clockwise (-) or anticlockwise (+) rotation. |
| OutputMode | Unsigned Byte  (0 to 255) | This is a bitwise variable.   |  |  | | --- | --- | | Bit # | Output | | 0 | Motor A | | 1 | Motor B | | 2 | Motor C |   A value of ‘0’ indicates that the motor output should be left *floating* when no power is applied (i.e. PWM value = 0). This is referred to as “coasting”.  A value of ‘1’ will indicates that the motor output should be ‘short circuited’, i.e. both motor terminals will be held at ‘1’. This will force a “brake”. This is the *default* setting for all motors, and generally considered to be more accurate (as the motor will stop at a specific point rather than coast to a stop). However, this method *does* consume battery power. [2] |
| InputPower | Unsigned Byte  (0 to 255) | This is a bitwise variable.  The value associated with each sensor indicates *whether* the sensor should be supplied with 9V power, and *how*.   |  |  | | --- | --- | | Bit #s | Output | | 0, 1 | Sensor 1 | | 2, 3 | Sensor 2 | | 4, 5 | Sensor 3 | | 6, 7 | Sensor 4 |   A value of ‘00’ indicates that the sensor should not be supplied with 9v power. (This is the appropriate value if a sensor is *not* connected to the port, if a sensor is *passive*, or the sensor is *self-powered*). [[5]](#footnote-5)  A value of ‘01’ indicates that the 9v power to the sensor should be pulsed, so that power is *not* supplied to the sensor when data is being read from it. Duration: 3 milliseconds supply, 0.1 millisecond measuring time (no 9v supply). This mode is for sensors categorized as ‘*active sensors’*, a legacy category that allow sensors from old LEGO® Mindstorms Robotics Invention System kits to be used with the NXT brick. [1, p. 7]  A value of ‘11’ indicates that the 9v power should *always* be supplied to the sensor. This mode is for use with sensors categorized as *‘digital sensors’* such as the LEGO® NXT kit’s Ultrasonic Sensor. [[6]](#footnote-6) |

**IOFROMAVR** essentially represents the section of memory in each microcontroller reserved for data sent *from* the coprocessor *to* the main processor.

**typedef** struct

**{**

UWORD AdValue**[**NOS\_OF\_AVR\_INPUTS**];**

UWORD Buttons**;**

UWORD Battery**;**

**}**

IOFROMAVR**;**

|  |  |  |
| --- | --- | --- |
| Variable Name | Type (native range) | Description |
| AdValue[3] | Array of  Unsigned Word  (0 to 65535) | The array holds 10-bit values from the analogue-to-digital conversion of the input sensors’ analogue input. The raw values from the ATmega48’s 10-bit ADC converter range from 0 to 1023.   |  |  | | --- | --- | | Array Index (#) | Input | | 0 | Sensor 1 | | 1 | Sensor 2 | | 2 | Sensor 3 | | 3 | Sensor 4 | |
| Buttons | Unsigned Word  (0 to 65535) | Buttons 1, 2 and 3 are connected to the co-processor’s input pin (to an ADC channel – ADC3) via a resistor ladder. Hence the status of all three momentary switches can be deduced from the value of the analogue-to-digital conversion stored *here*.  Button 0 is connected to the PD3/INT1 of the ATmega45 co-processor, a decimal value of 2047 is *added* to *this value* when Button 0 is pressed.  See Keypad schematic in Appendix x, and m\_sched.h |
| Battery | Unsigned Word  (0 to 65535) | This *word* (16-bit value) holds data of **(1)** the measured battery level, **(2)** whether the brick is being powered by 6x AA batteries or an NXT brick battery-pack, *and* **(3)** the version of the co-processor’s firmware.   |  |  |  | | --- | --- | --- | | Bits (#) | Value |  | | 15 | ‘0’ – 6x AA batteries  ‘1’ – Battery Pack | Battery type. | | 13 – 14 | 0 to 3 (number - major version) | Co-processor firmware version. | | 10 – 12 | 0 to 7 (number - minor version) | | 0 - 9 | 0 to 1023 | Gives the battery’s voltage output (in millivolts) when multiplied by 13.848. | |

###### Interface

The NXT Brick features three output ports (for motors), and three input ports (for various types of sensors).

All ports utilize 6-pin modular connection (6P6C RJ12). The brick features *female* connectors, to allow cables extending from motor or sensor modules to be plugged *in* via a *male* connector.

Motor Ports

Table x depicts the signals featured in the 6-pin connection to/from an NXT motor module. Note that the outputs to the DC motors themselves (MA0 and MA1) are sourced from motor drivers U1 (LB1836M) and U2 (LB1930M) controlled by PWM outputs from the ATmega48 co-processor, whilst the digital input pulses from the *incremental rotary encoders* are fed directly to the main processor via a Schmitt trigger that functions to fully recover the signal from the effects of power-attenuation and noise. [[7]](#footnote-7)

|  |  |  |
| --- | --- | --- |
| Pin | | Description |
| # | **Name** |
| 1 | Mx0 | Pulse-width modulated power signal to the motor (pin 1). |
| 2 | Mx1 | Pulse-width modulated power signal to the motor (pin 2). |
| 3 | GND | Ground – shared by all output ports. |
| 4 | POWERMx | 4.3V power from the NXT brick – shared by all input and output ports. Max current of 20mA can be drawn. Short-circuit to GND causes the brick to reset. |
| 5 | TACHOx0 | Inputs to Schmitt trigger circuitry, which leads to the *main processor*. |
| 6 | TACHOx1 | Inputs to Schmitt trigger circuitry, which leads to the *main processor*. |

Table x. P descriptions of the output port. [1, p. 5]

TACHOx0 and TACHOx1 are used by the default LEGO® firmware to capture the two outputs from the incremental rotary encoded embedded within the motor module, and hence, deduce the *direction of rotation* of the motor, and *count the degrees of rotation*. This feature allows us to *limit* the motor rotation (in either direction) to a *specific* number of degrees *in code*. [3, pp. 51-52]

Sensor Ports

LEGO® Mindstorms® NXT recognizes three types of sensors: [1, pp. 6-7]

* Passive Sensors – i.e. sensors that do not have any special timing or measurement requirements. These sensors produce analogue outputs that lead to the *co-processor*, which executes analogue-to-digital conversion every 3mS.
* Digital Sensors – i.e. sensors that have their own microcontroller that handles sampling and any analogue-to-digital conversion necessary. Unlike passive sensors, these sensors use the I2C protocol to communicate *directly* with the *main processor* at 9Kbps. *Note* that the *main processor* is always the *master* in any I2C communication established.
* Active Sensors – *legacy category* to allow sensors from earlier LEGO® Mindstorms kits (specifically LEGO® Mindstorms Robotics Invention System) to be used with the NXT. These sensors produce analogue outputs that lead to the *co-processor*, which executed analogue-to-digital conversion every 3mS.

|  |  |  |
| --- | --- | --- |
| Pin | | Description |
| # | **Name** |
| 1 | ANx | Analogue input from signal. |
| 2 | GND | Ground – shared by all input ports. |
| 3 | GND |
| 4 | IPOWERx | 4.3V power from the NXT brick – shared by all input and output ports. Max current of 20mA can be drawn. Short-circuit to GND causes the brick to reset. |
| 5 | DIGIxI0 | Leads to digital I/O pin of the *main processor.* Typically used for 9.6Kbps I­2C communication with digital sensor by default firmware. May also be used to control aspects of the sensor (e.g. switching on LED in the NXT Light Sensor, s). |
| 6 | DIGIxI1 | Leads to digital I/O pin of the *main processor.* Typically used for 9.6Kbps I­2C communication with digital sensor by default firmware. May also be used to control an aspect of the sensor (e.g. switching on LED in the NXT Light Sensor). |

It may also be of interest that DIGIDI0 and DIGIDI1 (i.e. pins 5 and 6 of Sensor 4) also lead to an ST485 IC (RS485 communication chip) to allow the Sensor 4 port to function as a high-speed communication port – though LEGO® has not developed any devices that utilize this. High-speed communication settings in within default NXT firmware: [1, p. 8]

|  |  |
| --- | --- |
| Specification Criteria | Value |
| Communication Speed | 921.6Kbps |
| Data Bits | 8 bits |
| Stop Bits | 1 bit |
| Parity | 0 bits |

###### PC Communication

The NXT brick features USB and Bluetooth communication which may be used to update the firmware, transmit data or receive commands.

USB

The NXT brick features a full-speed (12Mbps) USB 2.0 communication port. This feature is embedded within the AT91SAM7S256 IC *(main processor)*.

Bluetooth®

The *main processor* connects to a standalone IC (CSR BlueCore™ 4 with external 8Mbit FLASH memory) in order to implement Bluetooth® functionality.

|  |
| --- |
|  |
| Figure x. A diagram illustrating the interface between the *main processor* and the BlueCore™ IC. SPI interface allows the BlueCore™ chip to be updated if ever deemed necessary, and is not used in normal operation. Reset pin re-initializes the IC on start up, and may be used by firmware to disable Bluetooth functionality. BC4-CMD and ARM7-CMD are used to indicate the type of data expected, whilst the UART is used to transfer data and commands between the two ICs. [1, pp. 13-14] |

The main processor may configure the BlueCore™ chip to “stream-mode” in order to transfer data at a baud-rate of 220Kbps (when connection is established), *or* in “command-mode” in order to receive commands to control the Virtual Machine. It is the *later* mode that will allow us control over the motors and sensors via the VM. The IC is configured to function as a Bluetooth® Class II device in order to save power (giving maximum communication range of 10 meters).

BecauseBluetooth® functionality has been implemented via Serial Port Profile (SPP) paradigm (whereby the communication link is considered a ‘serial port’ as far as software is concerned), it becomes very easy to establish a connection to the NXT via different PC software development frameworks / programming languages, as most feature libraries explicitly for serial port communication.

*More information is available in the LEGO® Mindstorms® NXT Hardware Development Kit documentation (pages 12-14, and Appendix 8).*

##### Software°

###### The Virtual Machine

RXE Executables

#### Actuator – Servo Motor Module

Schematic of the NXT motor module can be found in Appendix x.

##### Motor

##### Incremental Rotary Encoder

|  |
| --- |
| http://cache.lego.com/r/www/r/mindstorms/community/-/media/franchises/mindstorms/community/elements/sensors/nxt%20light%20sensor.png?l.r=-1451730025 |
| Figure x. LEGO® Mindstorms® NXT light sensor. Courtesy of Lego Shop. |

#### Sensors – Light Sensor

The light sensor functions to represent light intensity through an analogue output signal.   
This sensor may be configured to detect *ambient light* (i.e. light from the atmosphere with the LED off) or *reflected light* (i.e. light from the red LED reflecting off nearby surfaces facing the sensor, after it is switched on).

The component responsible for converting light intensity into current (analogue signal) is the SFH309 phototransistor, with a response angle of 24° from centre. [4]

The light sensor has no special timing or measurement requirements as it simply outputs an analogue signal and features no microprocessor of its own, hence, it is considered to be a *passive sensor*.

Schematic of the NXT Light Sensor can be found in Appendix x.

#### Sensors – Sound Sensor

|  |
| --- |
| http://cache.lego.com/e/dynamic/is/image/LEGO/9845?$main$ |
| Figure x. LEGO® Mindstorms® NXT light sensor. Courtesy of Lego Shop. |

The sound sensor functions to represent sound intensity through an analogue output signal. The sound sensor is capable

Schematic of the NXT Sound Sensor can be found in Appendix x.

#### Sensors – Ultrasonic Module

Schematic of the NXT Ultrasonic Sensor can be found in Appendix x.

### Creating Perceptions: Pattern Classification

The principle behind pattern classification algorithms is to differentiate different classes of objects by viewing them as having different *models* (mathematical descriptions - typically of sets of combinations of [processed] input signals). [1, p. 2]

A typical pattern recognition system can be partitioned into six components as shown in Figure 1.

|  |
| --- |
|  |
| **Figure 1.** Six components/steps of a typical pattern recognition system. |

**Sensing:** The input is usually a transducer (i.e. a camera, microphone array, etc.). The transducers themselves will have their own characteristics and limitations such as bandwidth, resolution, sensitivity, distortion, signal-to-noise ratio, latency, etc. [1, p. 9] Though the quality of sensors may have an impact on the performance of the pattern classification system, and thus the limitations of the sensor modules being used in this prototype should be considered and taken into account during the design and testing of the system, the scope of this project is *not* inclusive of sensor and sensor interface design.

**Pre-processing:** The input signals may have to be rectified to reduce impact of irrelevant variations, increase the signal-to-noise ratio, and wrap input signals to a “standard template” (normalization).

**Segmentation and Grouping:** The main purpose of segmentation is to hone in on the features of interest (or ultimately, the object of interest) that would be useful in the classification process. This in itself is one of the major problems in pattern recognition; how can you segment the input signals (i.e. image of many fishes on a conveyor belt) before they’ve been categorized? Also, how can you categorize the same input signals before they’ve been segmented? This requires the system to ‘know’ when there is just a background or “no category” object present. [1, p. 10]

**Feature Extraction:** The purpose of this component is to yield a representation of the input signals such that the task of the Classification component seems trivial. Feature extraction may be considered a part and parcel of the classification system – the distinction being practical rather than theoretical.

Ideally, features should be measurements which be very similar for objects in the same category, and very different for objects in different categories – leading to the idea of seeking ‘distinguishing features’. Features must also be invariant to irrelevant transformations of the input i.e. translation, rotation, and scale. Occlusion (parts of an object becoming hidden), projective distortion (with distance, angle, etc.), and complex transformations of objects (causing deformation of measurements even though the object remains the same – i.e. when fingers are clicking, the hand is still a hand) need to be considered when selecting features – hence it is very important to know the exact domain of the pattern classification system at this stage.

**Classification:** This components uses the feature vector provided (from previous component) to assign objects to categories/classes. Generally, this involves determining the probability for each of the possible categories. The problems associated with classification usually depend on the variability in the feature values for objects in the same category in relation to objects of different categories. [1, p. 12]

The classification component should also consider the fact that there may be occasions where, due to occlusion or another reason, not all of the features associated with a particular category may be present. To assume the value of the missing feature is zero or an average of the values previously seen does not (provably) yield an optimal result – how a classifier should be trained or used in this situation is one of the concerns of classifier design.

**Post-processing:** The post-processor decides on a recommended action dependant on the result from the classifier. The component should take into account the typical error rate of the classifier (percentage of patterns assigned to the wrong category); this could potentially act as a negative feedback mechanism for the entire pattern classification system in order to improve feature patterns for classification.

It is to be noted however, that in the current design of the cognition system, the decision making process is outsourced to a separate faculty, the decision-making faculty. However, as the decision-making faculty relies very much on the classifications deduced by the visuospatial faculty, it is important to consider that the boundaries of the post-processing component should not be limited to just the visuospatial faculty alone as the entire cognition system can be considered a holistic and highly coupled system (or at least it should be due to the level of interdependence of the faculties – see figure x). Therefore, the post-processing stage should take into account the cost (or risk) of each particular action that results from the classification.

#### Multiple Classifiers?

It may be possible that having multiple pattern classification systems yields a more accurate result. The most common approach would be a pattern in parallel, with a final “super”classifier that produces the *final* classification using all *previous* classifications. The inherent problem with this (and often a subject discussion) is the matter of establishing how to determine when the minority opinion is correct. [1, p. 13] This topic is discussed in greater detail in Roli’s lecture notes titled ‘Mini Tutorial on Multiple Classifier Systems’. [2]

#### The Design Process

|  |
| --- |
|  |
| ***Figure 2.*** *Steps involved in the design of a pattern recognition system.* |

**Data Collection: Because** pattern recognition and classification systems need to be trained and tested with a set of training data large enough to assure good performance and representation (of objects).

**Feature Choice:** This is a crucial design step and depends largely on prior knowledge of the domain, and access to (fully representative) example data.

**Model Choice:** Designing the mathematical model that would ultimately allow for the designation of objects into their appropriate categories. How feedback and supervision will be used in the mathematical model (if at all), would be ascertained.

We are interested exclusively in the implementation of a *multi-class classification model*, as we must ultimately allow for the capability of this sub-faculty to create as many classifications of objects as necessary; this would allow for greater versatility in the way that the entire autonomous navigation system could be trained.Effectively, multi-class classification models may comprise of one or many neural networks, and the most common approach to tackle the issue of modelling such classification system is to decompose the problem into multiple two-class classification problems. [3]

**Training:** The process by which the system uses data to determine the classifier (or “pattern class”) must be determined. There are three *types* of learning paradigms by which a pattern classification system may be *‘learn’* classification [1, pp. 15-17]

1. **Supervised Learning** – a *teacher* provides the category labels *or* cost for each pattern in the training set.
2. **Unsupervised Learning.** This method uses no explicit guidance from a “teacher”. Hence, the system forms clusters or “natural groupings” of the input patterns alone.
3. **Reinforcement Learning.** This method of learning essentially involves a *teacher* reinforcing correct outputs, and discouraging incorrect outputs. I.e. the only input from the *teacher* is whether or not the outcome of the classifier is right or wrong.

**Evaluation:** An error rate must be obtained for the classification system in order to evaluate its performance and identity the need for improvement in its components. [1, p. 15]

**A note on the consideration of Computational Complexity**

It is important that the labelling time and storage requirements are considered when implementing the pattern classification system – with respect to the entire framework, including robot hardware, user PC and *where exactly the pattern classification code will be executed, how the data will be stored, how data will would be transferred (wirelessly) and whether it is feasible to consider custom logic to enhance overall performance (with respect to time).*

#### Bayesian Decision Theory

The Bayesian decision theory is considered a fundamental statistical approach to pattern classification. The theory involves simple computation of algebraic equations in order to quantify the trade-offs between various classification decisions using probability and cost (of decisions).

##### Procedure

The Bayesian Decision Theory can be utilized for the purpose of classification in the following manner [1, pp. 21-64] [4]:

1. Define a discrete number of categories, for example:
2. Assume a *priori probability* i.e. the likelihood of the next “catch” belonging to a particular category in terms of a probability. This is expressed as a *mass probability function*, P(•), which is in essence a probability value for a discrete variable (in this case ), where each value of the variable has its own probability value (i.e. each value of discrete variable will have a probability value describing it’s prior/likelihood of occurrence).   
    **Rules:**

, and

1. Rarely are decisions made using the *priori probability* alone.Instead, we supplement the classifier with other variables (measurements and indicators to *help* with the classification problem). These are expressed as *probability density functions*, in the form p(•), which essentially indicates probability for a continuous variable – the function essentially returns the *probability* of a defined *range* (i.e. some *unit distance*). This can be expressed as:  
     
    or   
     
   **Rules:**  
     
   , and  
     
      
     
   (i.e. the probability density function is normalized so that the area under each curve is exactly 1)  
     
   Ultimately, if x is the continuous variable and the class in question, the *class-conditional probability density* function can be expressed thus: – “the density function *given* the state of nature ”.
2. Bayes formula is used to deduce the *mass probability* of the object in question belonging to a particular category, given a particular *feature of interest* **x** (the measurement or indication to support the *priori probability*):

So…

##### Generalization of the Bayes Formula

Bayes Formula can be expressed (very loosely) in English in order to describe its generic operation:

***Evidence*** is essentially the *scale factor* that, in the equation, assures us that . Essentially a constant if all feature variables are known:

(visio illustration here)

##### Error & Decision Making

For , the error for any value of **x** is deduced in the following way:

If we go by the convention and define as a rule, that is selected so long as that , else is selected, then, the same equation becomes: .

The overall error in classification (termed *average probability of error*) is given by:

*Where denotes the probability of the and value occurring simultaneously.*

If we eliminate “evidence”, i.e. , from the equation – we ultimately end up with a more concise expression of (exactly the same) decision rule:

Decide if , else decide in favour of .

###### Risk Minimization

A general decision rule describes every possible action that can be taken for every possible observation of the continuous variable **x**. The risk associated with each action is denoted. Hence, the overall risk of the classifier is given by:

Therefore, minimizing for every **x**, we can reduce the overall risk of the system. This can be achieved by the computation of the following equation:

Where is the loss function that indicates how costly each action is, and ultimately allows the conversion of a probability into a decision.

*The minimum overall risk is called the Bayes risk –indicative of best possible performance of the classifier, and is denoted R\*.*

###### Insight into Bayesian Decision Making

If for any given value of **x**, we set , then the ultimate decision hinges entirely on the *priori probabilities*.

If, instead, we set – i.e. both states of nature are *equally* probable, *then* the decision hinges entirely on *likelihoods*.

In general *both o t*hese factors are important, and the purpose of Bayes decision rule is to combine both in order to achieve *minimum* probability of error.

##### Application in AVINSOR Prototype 1

The proposed use of the Bayes classification algorithm is illustrated in figure x.

###### Input streams - VARx

In the AVINSoR framework, the Bayes classification algorithm is used to classify the values of input variables (i.e. the different streams of input data) amongst a number of differentcategories, to create a number of *different* ‘perceptions’ that will ultimately aid the navigation system in decision-making. [[8]](#footnote-8) All input variables (whether from sensors, or other sources) are referred to as *input streams*.

###### The Bayes Classifier Module

AVINSoR Client is designed to allow the user to instantiate an (ideally) unlimited number of *Bayes Classifier Modules*– each to be associated with a particular ‘perception’. Hence the term *Bayes Classifier Module* should be considered synonymous to ‘perception’ beyond this point.

The client allows the user to define:

1. The different *input streams* from the system to be associated with the Bayes Classifier Module.
2. The different *categories* that the Bayes Classifier Module should classify the input values (of the associated input streams) amongst.

Bayes Classifier Object

For each input stream associated with a Bayes Classifier Module in the *client* application - a Bayes Classifier object is instantiated in *MATLAB®*. A MATLAB Bayes Classifier object functions to generate and maintain all probability, risk, and action matrices that associate the particular input variable stream with the defined list of categories.

Through the client, the user can associate any potential value that could be seen at an *input variable stream* with the *maximum likelihood* *of occurrence of a particular category*. This processes is referred to as *“creating an association” (*between the *input value* and a *category*,for a particular *‘perception’).*  This is achieved through the Bayes Classifier object which represents the input stream of the ‘perception’ in MATLAB - essentially a Gaussian curve should be created at the index of the category of the *likelihoods* matrix, which peaks at the *associated* value.

Note: for each Bayesian Classifier, each category also has a *prior probability* associated with it, the default value of the *priory* (upon instantiation) is the reciprocal of the number of categories (i.e. all are equivalent, hence decision is made *entirely* on the basis of the *likelihoods* matrix). The priories can be changed by the user if a category is deemed to have a higher or lower likelihood of occurrence regardless of the actual input value seen. This will be achieved through the client GUI – which will provide mathematical abstraction and some automation to guide the process.

###### Result

The decision logic of a Bayes Classifier Module essentially returns the *mode* (most frequently occurring) *action* *category* (*viz.* decision) of the Bayes Classifiers - *if* there is an *obvious* mode (i.e. the frequency of occurrence of a particular action is *exclusively* greater than all other actions). On the case of there being no obvious mode, the decision logic returns the action category of the Bayes Classifier whose action is associated with the *minimal risk* value. In the case of their being more than one Bayes Classifier with the same *minimal* level of risk, the *action category* of the *first to be iterated over* will be returned.



##### Object-Orientated Design



Figure x. UML Class diagram of the classes that will be used to realize the Bayes Classifier Module in AVINSoR Client.

###### Class Descriptions

**ClassVariable** is the class that represents a variable associated with classification – only a single instantiation should be required per input stream *in* the system. All classification variables of interest to a Bayes Classifier Module (object) are passed through as reference via the **ModuleVariables** parameter.

**ModuleVariables** and **ModuleCategories** are essentially enumerable collections that should inherit from .NET Framework’s iEnumerable class.

**ClassCategory** is the class the represents a classification category to be used by an instantiation of a Bayes Classifier Module. Note that unlike ClassVariable, ClassCategory is exclusive to a particular instantiation of a BayesClassifierModule, hence its complementary collection type ModuleCategoriesdoes not have to be instantiated outside of the classifier module and there is no need for it to be passed as a parameter. Instead, the collection of categories is initialized upon the instantiation of the classifier and can be accessed publicly to add/remove/modify a particular category.

**MatlabInterface** is a static class that initializes the global connection to the MATLAB® COM Automation Server and provides a reference to it. The class will also feature methods that may be useful when transferring data between the client application and MATLAB® server.

**BayesClassifierModule** is the class that represents a module like that depicted in figure x. Though its properties and behaviours are self-explanatory through the UML alone

Behaviours

BayesClassifierModule(…)

Represents the Bayes Classifier Module.

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| server | MLApp | Reference to the global MATLAB COM Automation Server. |
| moduleVariables | ModuleVariables | A reference to all the variables (input data streams) to be associated with this particular Bayes Classifier Module. |

**Returns** a BayesClassiferModule object (class constructor).

Associate(…)

Associates a particular (specified) value of an input stream with the maximum likelihood of belonging to the specified classification category.

If the referenced variable or category does not exist in the BayesClassifierModule’s collections, an exception should be thrown with a notification (for the GUI).

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| variable | ClassVariable | Object reference to the variable. |
| category | ClassCategory | Object reference to the category. |
| value | int | Value of *variable* to associate with *category*. |

SetPriory(…)

Adjusts the prior probability of the referenced category with respect to the specified variable.

If the referenced variable or category does not exist in the BayesClassifierModule’s collections, an exception should be thrown with a notification (for the GUI).

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| variable | ClassVariable | Object reference to the variable. |
| category | ClassCategory | Object reference to the category. |
| priory | double | The prior probability. |

GetDecision(…)

Gets the ultimate classification decision made by the Bayes Classifier Module (via *decision logic*) using values currently seen (or last sampled) in the input stream variables associated with the module.

**Parameters:** *none.*

**Returns** a ClassCategory object reference to the resultant category.

GetDecision(…)

Gets the ultimate classification decision made by the Bayes Classifier Module (via *decision logic*).

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| values | int[] | An integer array of the values to use with respect to the input variables. The integer at index 0 should be complimentary to the variable at index 0 of the module’s variable collection.  Note that this method could be implemented using a custom collection class of (for example) ‘VariableValue’ objects with the purpose of associating an integer value with a referenced input stream variable, hence eradicating ambiguity in indexes. |

**Returns** a ClassCategory object reference to the resultant category.

### Creating Perceptions: Visual Object Recognition

### Creating Movement Patterns

#### From Cartesian Coordinates to Differential Motor Outputs

Differential wheeled robots have two powered wheels, hence a motor drives the *left* wheel, and a motor drives the *right* wheel.

Cartesian Drive Mapping is the system by which motor output signals for both motors of a differential wheeled robot (left wheel *and* right wheel) are deduced from a Cartesian coordinate input. As explained earlier, the NXT framework motor output command consists of a *tachocount limit* (number of degrees of rotation, at which to stop), *absolute power* (indicative of speed)*,* and *polarity* (the direction of rotation). *Our implementation of the Cartesian Drive Mapping algorithm must calculate all.*

Fortunately, with little research, two implementations of such an algorithm where found. [1] [2] The later referenced implementation sourced from ‘a drop in the digital ocean’ is essentially a C and C# translation of the original ‘GoodRobot’ implementation in JavaScript. Note that the referenced implementations have a very rudimentary level ofmotor control. Because the implementation of the Cartesian Drive Mapping algorithm *only* varies Mpower, it is the *time* that the particular signal is *held* that determines the number of rotations of the wheel and hence the distance covered by that particular wheel (in either direction). As the reference implementations are for use in robotic systems designed to be manually controlled by a human user *through a joystick module* (which basically feeds the algorithm with a Cartesian coordinate), the human user acts as visual *feedback* - holding that *particular* joystick position (the calculated Mpower values for each motor) until he or she is satisfied with the distance and position achieved.

If the Cartesian Drive Mapping algorithm is to be used by an autonomous system, a greater level of control is required over the distance a wheel travels with any particular motor output signal (i.e. the time the signal is held should be just enough to achieve a particular distance).

Fortunately, as mentioned earlier, the LEGO® Mindstorms® NXT framework features a motor encoder, which allows the *tachocount* to be set to a constant. The encoder will allow a particular motor output signal to be applied to the motor until the *tachocount* (number of degrees) is reached. This will allow a greater level of control over the distance the motor covers with each motor output signal, though the distance will essentially be expressed as the *number of wheel rotations* until it is linked to the wheel’s *radius* –obviously both wheels would have the same radius. Linked to a radius, we can algebraically deduce the *unit distance* (e.g. centimetres or meters) the robot will travel - associating a step change in the Cartesian grid (Δx and Δy) with a step change in unit distance. [[9]](#footnote-9)

##### Relationship between Physical and Internal Variables

Before discussing the operations of the Cartesian Drive Mapping algorithm, let us make certain the relationship between the Cartesian coordinate *(input)*, the physical (real world) distance travelled, and the components of the generated motor output command.

**Let:**

*The Cartesian grid (exists only in software)*

Size of Cartesian grid from origin (0, 0) farthest points from origin would be (±G, ±G),   
i.e. in the NW, NE, SE, SW direction at polar radius *r*(±G, ±G).

*The physical distance the robot is expected to travel (hypothetically)*

*Real world* distance (cm)   
*Real world* time (s)   
*Real world* speed (cm/s)

*The physical wheel*

Wheel radius (cm)   
wheel diameter (cm)   
wheel circumference (cm) =

*Motor output signals (exist only in software / transmitted commands, but converted into physical signal by NXT hardware)*

The *tachocount limit* assigned to a motor output signal

The *power* assigned to a motor output signal (polarised) **But,** for the purpose of understanding relationships between the different variables involved, we will use:

The *magnitude power* assigned to a motor output signal (i.e. absolute of *power*)

The *polarity* of

Note that

###### The relationship between distance travelled, circumference of the wheel and tachocount limit

###### The relationship between the Cartesian grid and the distance travelled

As mentioned earlier:

The maximum possible *real world* distance that the robot can move with any single motor output command depends on the maximum possible value of. is of type *unsigned integer*, at least according the interfacing libraries that we utilize to establish a communication between the NXT and our PC (RWTH toolbox and MindSqualls library). The maximum value of an unsigned integer type in both MATLAB *and* C# is 99,999.

It may not actually be appropriate to have the Cartesian coordinate (±G, ±G) correspond to the colossal distance represented by Dmax as it may measure up to several meters, and therefore be inappropriate or inconvenient for certain applications. Hence it is necessary to allow the *change in* *distance per unit change in radius [of Cartesian coordinate input (x,y)]* to be manually assigned in code. Note that radius *r* is only a very rough linear approximation of the actual travel trajectory of the robot.

Where is essentially the calculated to induce the change in distance per unit change in the magnitude value of x or y. Hence, re-arranging the above equation:

Wheredenotes an assigned distance value in centimetres. However, a validation statement would be required (in software) to only accept a value of if the value deduced is (maximum acceptable value for unsigned integers).

###### The relationship between the absolute value of motor power with speed, time and distance

In our software interface to the NXT, *power* is of type *signed byte*, therefore:

-128

Therefore, has a total of 256 different values.

Because is even, the length of the integer x-axis and y-axis would be odd, as the first value would be (0, 0) and max values would be (±G, ±G). If we are to vary

Because we ] expect the size of the x-axis and y-axis to be odd, from the origin to the maximum value (because we expect G to be an even number). We can scale |Mpower| to the following:

-127

The distance travelled by a wheel upon a command is ultimately determined by the *tachocount* *limit* associated with the motor command (N), and the circumference of the wheel.

*Ex 1.* The time taken to reach the required distance will be proportional to *magnitude power*. The greater the *magnitude power*, the *faster* the distance will be achieved.

Ex 2. However, the *exact* mathematical function that relates speed (cm/s) to *magnitude power* remains unknown, because the change in the time period taken to achieve a distance over a unit change in *magnitude power* remains unknown.

This makes *time* the variable that needs to be measured (as accurately as possible) before a *real world* speed (i.e. in cm/s) is associated with a *magnitude power*. In terms of implementation, *time* could either:

1. *Be assigned as a constant* in code *–* this would require the time taken for the robot to achieve a particular distance (point) to be measured, averaged and (eventually) deduced experimentally.
2. The time taken to achieve a particular distance (point) could be deduced *through a carefully designed calibration procedure*.

*However,* measuring time (and relating that to a distance) would not take into account changes in velocity (*acceleration* or *deceleration*)of the motor, particularly with regards to change in distance of the target point.

When the relationship between the change in *magnitude power* and *the time taken to achieve a particular distance* *(speed)* is determined, it may become easier to control the exact *real world* speed of the motor. But introducing another approximated variable (T) seems rather unnecessary, as it will not improve the function or performance of the system. Thus, real world speed and time should be left out of the equation, the assumption that is adequate, where is considered 0 cm/s in terms of *real world* speed, and is considered

###### Variable relationship summary: Cartesian Driving Mapping

The Cartesian coordinate input is ultimately supposed to be determine the *real world* angle (direction) and *radius* (distance) of the target position. This will obviously be different to the polar angle and radius deduced from the Cartesian input (to model the non-linear trajectory of each individual wheel would be very time consuming).

The target position is achieved through the action of the left motor and right motor; we can control the *tachocount limit*, *polarity,* and *motor power* of both. We know that:

**(a)** *Polarity* determines the direction (clockwise or anticlockwise) a wheel should move in order to achieve the target position (+1 = forward, -1 = reverse, 0 = no movement).

**(b)** T*achocount limit* determines the *number of degrees* the wheel should rotate in order to reach the expected distance. As calculated earlier: centimetres.

**(c)** *Magnitude power* determines the speed at which the target distance is achieved (through an arbitrary relationship that hasn’t been modelled). Inherently, Speed has *nothing* to do with *accuracy* of movement.  
  
Whether actual *real world* *time* (in seconds) taken to achieve a distance is deduced via calibration *or* via trial and experimentation (hard-coding), there *is* going to be a degree of inaccuracy.

Because *Tachocount limit* and *polarity* are sufficient is controlling the wheels to a position indicated by the Cartesian input coordinate, we leave |Mpower| to simply act as a scale of speed that can be adjusted by the user.

##### Differential Wheel Drive and the Cartesian Grid

As illustrated in figure 1, the grid used in Cartesian Drive Mapping is always *square*. Therefore, both axes, x *and* y, will have the same absolute maximum value. The decision behind the size of the *input grid* will ultimately affect the fidelity of possible movements (and movement patterns) as a greater size would equate to a greater *resolution*. Thus: The larger the size of the grid, the greater the magnitude, therefore implying greater resolution, which would ultimately result in a greater number of possible movements i.e. motor output combinations (this may not necessarily be advantageous during development).

|  |
| --- |
|  |
| **Figure 1.** An illustration depicting the Cartesian grid. The robot is denoted by the block arrow facing north, positioned at the origin. As far the Cartesian Drive Mapping algorithm is concerned, the position of the robot does *not* change, this is regardless of any physical movement of the robot; hence any input coordinate presented will be respective to the robot’s *current* position. |

It is important to note that in differential mode driving, one wheel is always responsible for the distance the robot travels towards one of the four possible 45° points. The 45° point will correspond to the North East, South East, South West and North West direction.

|  |
| --- |
|  |
| Figure X. For a Cartesian coordinate to be of true NE, SE, SW or NW direction, |φ| = |ϴ| = 45°, because |x| = |y|. |

|  |  |  |
| --- | --- | --- |
| **Left Motor** | **Right Motor** | **Direction of Travel** |
| On (+) | Off | NE |
| On (-) | Off | SW |
| Off | On (+) | NW |
| Off | On (-) | SE |

Table x. The table shows the direction of travel (relative to the robot’s current position at the origin, facing north) in accordance to the motor active, and the rotational direction (- indicating reverse).

|  |  |
| --- | --- |
|  |  |
| Figure x (left) illustrates movement in NE and SW direction are controlled by movement of the left wheel alone.  Figure y (right) illustrates movement in NW and SE direction are controlled by movement of the right wheel alone.  Red wheel indicates the *distance wheel*. | |

Figure x and y illustrate that the polar radius of the coordinate corresponds to a real world distance ***D***. Note that the *real world* distance ***D*** at which the robot achieves the 45° angle is ultimately dependent on the *time* that motor power signal is held. If the signal is held longer than necessary, the robot would travel *beyond* the *real world* point at which the 45° angle is achieved. A motor encoder (via *tachocount limit*) automatically allows time to be limited by a given distance or number of degrees.

Observing and studying the role of the *distance wheel*, it becomes clear that the role of the opposite wheel is then, to deviate/turn-away from the |φ| = 45° angle: either towards |φ| = 0° - to force a sharp turn, or |φ| = 90° - to force straight travel with *no* turn at all. See figure x.

|  |
| --- |
|  |
| Figure x. The diagram illustrates the role of the *turn wheel* in modifying the direction (absolute angle) of the movement towards the direction of the x-axis (|φ|=90°), or the y-axis (|φ|=0°). The orange wheel is the appropriate *distance wheel* for the quadrant (i.e. closest associated 45° point). The *turn wheel* is black if inactive, blue if at *opposite* polarity to the *distance wheel*, red if at *same* polarity to the *distance wheel.* Because |φ| of all points is 45° apart – hence turn wheel, tachocount is kept the same. |

##### Understanding the Cartesian Drive Mapping Algorithm

The GoodRobot article briefly explains the derivation of the algorithm. [2]

The algorithm’s operation and derivation are discussed here forth.

###### (1) Prior Assumptions

Initially, a list of assumptions are made from prior knowledge of differential wheeled systems, and the direction of movement they induce: [2]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Direction of Movement | Point  (Figure 1) | Left Motor | | Right Motor | |
| **Magnitude Power** (indicating speed) | **Polarity** (indicating direction) | **Magnitude Power** (indicating speed) | **Polarity** (indicating direction) |
| N | d | P = Pmax | + | P = Pmax | + |
| S | f | P = Pmax | - | P = Pmax | - |
| E | h | P = Pmax | + | P = Pmax | - |
| W | b | P = Pmax | - | P = Pmax | + |
| NE | g | P = Pmax | + | 0 | 0 |
| SE | i | 0 | 0 | P = Pmax | - |
| SW | c | P = Pmax | - | 0 | 0 |
| NW | a | 0 | 0 | P = Pmax | + |

**Table 1.** A table of motor values we would assume for movement towards a particular direction.

***P*** is some arbitrary power value, indicative of the speed of motor rotation. Because we attempt to link this concept to the Cartesian grid in Figure 1, and the example points are at the edge of the grid (maximum of x and y, i.e. at + gridSize or - gridSize), we assume maximum speed due to greatest distance from origin. If the points remained in the same direction (angle from the origin) but decreased in magnitude, they would move closer to the origin, hence the power would decrease (tending to 0 as the point reaches the origin).

###### (2) Deduction of Relationships

Hence, the following relationship can be deduced between the Cartesian coordinate (input), its equivalent conversion to polar form, and the correct motor power signals to send (output). At this stage, we will assume that the scale of motor power signals (output) are equivalent to the scale of the axes of the square Cartesian grid (input): [[10]](#footnote-10)

* Variable x (i.e. the x axis) represents the *intended* *turn* (i.e. left or right).
* Variable y (i.e. the y axis) represents the *intended* backward/forward *movement*.
* When x = 0, this is indicative of no deviation from the centre line, i.e. the robot is to drive absolutely straight ahead, or reverse, with absolutely noleft or right turn. Hence, there would be *no* difference between left and right motor outputs (in either direction *or* magnitude power).
* When y = 0, this is indicative of no backward or forward motion, and therefore any movement produced is *solely* a turn/rotation. Thus, when y = 0, the magnitude power at both motors is the same, but at *opposite* polarity).
* When x = y, this indicates a 45° turn, where one motor is inactive (off / no motion), and the other motor is moving.

|  |
| --- |
|  |
| **Figure x**. Table of sample Cartesian coordinates (from figure 1), the equivalent polar coordinate, and the motor output values we expect to see from *prior* knowledge of differential wheeled driving. |

Hence we can also make assumptions about the algorithm’s operation before implementation.

* One wheel is used for turning, and the other for overall movement (to a distance). However,
  + When x = 0: *turning wheel* has the same polarity *and* absolute motor power as the *distance wheel*.
  + When y = 0: *turning wheel* has the same absolute motor power as the *distance wheel*, butwith *opposite* polarity.
* *Absolute φ* gives us as indication of how close the input coordinate lies to a 45° point (see figures x, x, and x). |φ| = 90° will therefore be indicative of no turn, i.e. *turn wheel* output = *distance wheel* output; |φ| = 0° will be indicative of full sharp turn, i.e. *turn wheel* output = - *distance wheel* output.
* Exactly which wheel is designated *turn wheel* or *distance wheel* is determined by the quadrant in which the Cartesian coordinate lies.

###### (3) Operation

1. Assume the range of all possible motor power outputs in the form of an array ranging between   
    and.
2. First, the input coordinate is classified as belonging to a particular quadrant (1,2, 3 or 4) using the following rule:

will allow us to determine which wheel is the *turn wheel* and which wheel is the *distance wheel* (later).

1. The *motor power output to the distance wheel* is determined essentially by the maximum absolute value of x and y.
2. Now, to calculate the *motor power output to the turn wheel.*  
   The input coordinates (x, y) is converted to polar form (radius *r*, angle θ). Essentially, it is the *magnitude* of θ that we are interested in – as its value is indicative of the point’s proximity towards the NE, SE, SW, or SE point (depending on the quadrant).
3. The angle θ is in radians, and should be converted into degrees (denoted φ).  
     
      
     
   where

This absolute angle would range anywhere from 0 to 90 degrees - i.e. scaled to fit to the 1st quadrant, with any point of being deemed to lie in the 45° direction appropriated by the actual quadrant of input (x, y).

1. We already know that**:**
   1. *The polarity of the turn wheel should be 0 when, as the movement towards a 45° point is controlled solely by the distance wheel.*
   2. *Polarity would be +1 when as the polarity of the turn wheel would be the same as the polarity of the distance wheel.*
   3. *Polarity would be -1 when as the polarity of the turn wheel would be the opposite to that of the distance wheel.*

Hence the output polarity of the *turn wheel* can be represented as a coefficient. The advantage of a coefficient with a range is that it can be used to scale the motor outputpower by the ‘severity’ of the turn (in order to accurately control deviation from the 45°):  
 (where polarity is ‘-‘, value of moves from -1 to 0)   
or (where polarity is ‘+’, value of moves from 0 to +1).

The ***turn coefficient*** denotes the degree to which the input coordinate deviates from the 45° line, and to which direction. Because a turn-coefficient of 0 denotes the point exactly 45° from the origin, means the turn is less than a 45° turn, and therefore the turn wheel would have to provide supplementary (same polarity) movement towards the direction of the distance wheel in order to *cut* from the 45° turn that the direction wheel would be making if it were to move *on its own* (this type of turn can be called an “soft” turn, as the turn < 45°) – hence when , the robot should appear to travel absolutely straight.

On the other hand, if, the turn is greater than 45°, and therefore the turn wheel would have to offset the angle *beyond* the 45° that the direction wheel would achieve on its own, this is accomplished by travelling in the reverse direction to the direction wheel (this type of turn may be referred to as an “sharp” turn, as the turn > 45°).

|  |
| --- |
|  |
| **Figure 2.** Plot illustrating the linear relationship between the magnitude angle () and the calculated turn coefficient. |

1. The power output value for the motor responsible for the *turn wheel* is calculated thus:

Where is the ***magnitude distance* from the 45° lines** (NE to SW, NW to SE), which (obviously) intersect at the origin. Note that. Ultimately, it is the*magnitude distance*scaled using the *turn coefficient* that gives us the magnitude of the motor output to the *turn wheel*.

When the coordinate is at a 45° angle from the origin, |x| and |y| are exactly the same, hence the difference is 0.Coincidently *and* when. However, as φ approaches 90° or 0°, is maximum, indicating maximum magnitude power for straight travel and sharp turns.   
  
Note that because the deduction of the motor output of both wheels is dependent solely on the *magnitude* values of (x, y) and the *magnitude* of φ of its equivalent polar form (r, θ).

|  |
| --- |
|  |
| **Figure x, Figure x.** A colour surface plot (top view and angled view, respectively) depicting the value of *magnitude distance*in relation to the value of x and y. Note that gridSize = 100, so that would be the absolute value (min and max) of x, y and the *magnitude distance*. More importantly, note that the value of magnitude difference tends to 0 as the absolute values of x and y become equal (indicating |φ| = 45°). Note the symmetry – the first quadrant of the Cartesian grid represents the *magnitude distance* of all coordinates. |

1. The ***quadrant*** of the input coordinate tells us the *rough* direction as to which the robot is to move. Hence, we can select the wheel that will be the *turn* wheel and the *distance* wheel.

|  |  |  |  |
| --- | --- | --- | --- |
| Quadrant of Input Coordinate | | Turn (- or + angular deviation from 45° point) | Distance to 45° point |
| Quadrant # | **Associated direction / |φ| = 45° point** |
| 1 | NE =  (G, G) | Right Wheel | Left Wheel |
| 2 | SE =  (G, -G) | Left Wheel | Right Wheel |
| 3 | SW =  (-G, -G) | Right Wheel | Left Wheel |
| 4 | NW =  (-G, G) | Left Wheel | Right Wheel |

Hence,

Flowchart

The following flowchart depicts the operation of the *original* (GoodRobot) algorithm.  


##### The AVINSoR Modification

Adjust tachocount limit and polarity not absolute motor power and polarity

Under ‘Relationship between Physical and Internal Variables’, we concluded that the AVINSoR implementation of the Cartesian Drive Mapping algorithm should adjust the *tachocount limit* and *polarity of motor power* in accordance to the position indicated by the Cartesian coordinate (input) rather than *absolute motor power* and *polarity*. This is because adjust *tachocount limit* is more effective at relating positions on a Cartesian grid with a ‘*real world’* distance.

Another reason to support this modification is that lower motor power values may not be suitable for a particular terrain, or may produce incredibly slow and ineffective movement. This is okay for robots whose movement is controlled by a human user, as the human user (acting as visual feedback) can always adjust the speed manually via the joystick; if the Cartesian inputs are being generated by an artificial intelligence algorithm or some other form of *code* than this will not be recognised. Hence the *absolute motor power* is defined as a suitable or practical range that the end-user may modify.

Operation

The operation of the modified Cartesian Drive Mapping algorithm can be described by the flowchart in figure x. Figure y describes how the data from the output *object* of the *CalculateDrive(…)* method is used to generate and execute motor output commands at variable values of magnitude motor power. The object oriented design of the Cartesian Drive Mapping algorithm is described in greater detail under ‘Object Oriented Design of the Cartesian Drive Mapping algorithm’.



Figure x. A flowchart describing the *CalculateDrive(…)* method of the ‘CartesianDriver’ object.



Figure y. A flowchart describing the *Play(…)* method of the ‘CartesianDriver’ object.

##### Object Oriented Design of the Cartesian Drive Mapping algorithm

As all output commands to the NXT-based robot assembly are computed from a client-pc (via a Bluetooth® connection). This means that the Cartesian Drive Mapping algorithm will be embedded with the client-side *application*, which is designed and implemented using the object-orientation paradigm.



Figure x. A UML diagram describing the classes to be implemented to allow Cartesian Drive Mapping in the client-side *application.*

###### CartesianDriver

Properties

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Controller | NxtController | Points to the ‘NxtController’ object used by the *play(…)* method to generate and execute motor output commands. *Default value is null.* |
| minAbsMotorPower | Unsigned Integer | The minimum allowable motor power to be set when wheel(s) are moving. Default is 30. |
| maxAbsMotorPower | Unsigned Integer | The maximum allowable motor power to be set when wheel(s) are moving. Default is 127. |

Behaviours

CartesianDriver(…)

Class Constructor. Creates a CartesianDriver object. Automatically sets motor power (motorPower) to maximum allowable motor power value.

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| gridSize | Unsigned Integer | The size of the Cartesian grid from the origin. This will determine the minimum and maximum value of x and y.  Hence determines internal properties: minXYVal (= -gridSize), maxXYVal (= +gridSize). |
| wheelRadius | Double | The radius of the robot’s wheels, in centimetres.   Hence determines internal properties: wheelRadiusCM (= wheelRadius), distancePerDegreeCM (= wheelRadius/360). |
| minAbsMotorPower | Unsigned Integer | *Optional.* The minimum allowable motor power to be set when wheel(s) are moving. Default is 30. |
| maxAbsMotorPower | Unsigned Integer | *Optional.* The maximum allowable motor power to be set when wheel(s) are moving. Default is 127. |

**Returns** the created ‘CartesianDriver’ object.

Calibrate(…)

Calibrates the ‘CartesianDriver’ object to determine the number of degrees of wheel rotation per unit change in x or y of the input coordinate. Hence, this method determines the internal property ‘degreesPerStep’. Note that this method should throw an exception if the calibration fails to determine an appropriate value for ‘degreesPerStep’.

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| delta | Object | This should be an unsigned integer if mode is passed ‘SetDeltaDPerUnitAbsXY’ or ‘SetDeltaDOfMaxXY’, or double otherwise. If the parameter is passed an object which is not of the correct type, an exception should be raised before ‘degreesPerStep’ is calculated. |
| mode | CalibrationMode | Determines how ‘degreesPerStep’ should be calculated. |

Drive(…)

Generates a motor output command for the left and right motors in accordance to the Cartesian coordinate *input*. Gets NXT to execute motor output commands to induce movement.

Method should raise an exception if x or y value < minXYval or > maxXYval.

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| x | Integer | The x value of the Cartesian coordinate. |
| y | Integer | The y value of the Cartesian coordinate. |
| inhibitPhysicalMovement | Boolean | When ‘true’, the execution of motor output commands on the NXT are inhibited (no physical movement of the robot). |

**Returns** a ‘CartesianDriverOutputs’ object consisting of calculated motor outputs.

SetAbsMotorPower(…)

Sets value of (absolute) motor power to use when motor output commands are being executed by the NXT.

The method should throw an exception if [absPower < minAbsMotorPower] or [absPower > maxAbsMotorPower].

Hence this method determines internal property ‘absMotorPower’.

**Parameters:**

|  |  |  |
| --- | --- | --- |
| inhibitPhysicalMovement | Unsigned Integer | When ‘true’, the execution of motor output commands on the NXT are inhibited (no physical movement of the robot). |

Play(…)

Execute the motor output commands represented by a ‘CartesianDriverOutputs’ object.

Execution of motor commands can only take place if ‘Controller’ is not *null* (i.e. an ‘NxtController’ object is associated with *this* object), otherwise an exception is thrown.

See figure y - flowchart describing the method’s operations.

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| cartOutputObj | CartesianDriverOutputs | The ‘CartesianDriverOutputs’ object that represents the motor output commands to be generated and executed. |

Internal Behaviours

CalculateDrive(…)

Essentially encapsulates the mathematical operation of the Cartesian Drive Mapping algorithm, particularly the conversion of Cartesian coordinate to an output variable for the left and right motors.

In our case, the output variables will be the *tachocount limit* for the left and right motor.

See figure x - flowchart describing the method’s operations.

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| x | Integer | The x value of the Cartesian coordinate. |
| y | Integer | The y value of the Cartesian coordinate. |
| minOutputVar | Integer | The minimum tachocount limit. This should be passed a 0. |
| maxOutputVar | Integer | The maximum tachocount limit. This should be passed degreesPerStep × maxXYVal. |

**Returns** an integer array consisting of an integer output calculated for the left motor, and an integer output calculated for the right motor (tachocount limits).

ReMap(…)

This is borrowed as-is from *a drop in the digital ocean*’s “tank-drive” implementation. [1] The method is used in *CalculateDrive(…)* to map values from the Cartesian range to the appropriate range for tachocount limit.

Note that the input parameters and the output are *floating-point values*.These are rounded to the nearest integer in *CalculateDrive(…)*.

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Value | Float | The number in the *input scale* whose equivalent is to be sought in the *output scale*. |
| Min | Float | The minimum value of the *input scale*. |
| Max | Float | The maximum value of the *input scale*. |
| From | Float | The minimum value of the *output scale*. |
| To | Float | The maximum value of the *output scale*. |

**Returns** a floating-point value of the *output-scale* equivalent.

###### CalibrationMode

This is essentially an enumeration of the different ways that a CartesianDriver object can be calibrated – in order to deduce the number of degrees the motor should rotate per unit change of (magnitude) x or (magnitude) y on the Cartesian grid.

SetDeltaDPerUnitAbsXY

Denotes that the CartesianDriver should be calibrated such that *a particular distance is achieved for a unit change in (magnitude of) x or y*.

SetDeltaNPerUnitAbsXY

Denotes that the CartesianDriver should be calibrated such that the *wheel rotates a particular number of degrees per a unit change in (magnitude of) x or y*.

SetDeltaDOfMaxXY

Denotes that the CartesianDriver should be calibrated in such a way that *the particular distance per unit change in (magnitude of) x or y* allows a certain distance to be achieved when (x, y) = (±G, ±G).

SetDeltaNOfMayXY

Denotes that the CartesianDriver should be calibrated in such a way that *the particular number of degrees a wheel rotates per unit change in (magnitude of) x or y* allows a certain number of degrees rotation to be achieved when (x, y) = (±G, ±G).

###### CartesianDriverOutputs

This is essentially the object that stores the relevant data to produce the correct left and right motor output commands for the position indicated by the particular Cartesian coordinate for which it was produced.

Properties

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| MPolLeft | MotorPolarity | *Read only.* The polarity of the left motor. |
| MPolRight | MotorPolarity | *Read only.* The polarity of the right motor. |
| TachoLimitLeft | Unsigned Integer | *Read only.* The number of degrees for the left motor to rotate. |
| TachoLimitRight | Unsigned Integer | *Read only.* The number of degrees for the right motor to rotate. |
| DelayInSeconds | Unsigned Integer | *Read only.* The delay (in seconds) before this command is sent to the NXT for execution. This is for use when the object is a part of a collection to form a movement pattern, and is only relevant when the object is *not* the first in a collection. |

Behaviours

CartesianDriverOutputs(…)

Class constructor. Creates a ‘CartesianDriverOutputs’ object.

**Parameters:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| lMotPol | Integer | The x value of the Cartesian coordinate. |
| rMotPol | Integer | The y value of the Cartesian coordinate. |
| lMotTachoLim | Integer | The minimum tachocount limit. This should be passed a 0. |
| rMotTachoLim | Integer | The maximum tachocount limit. This should be passed degreesPerStep × maxXYVal. |
| delayS |  |  |

**Returns** the created ‘CartesianDriverOutputs’ object.

1. Explain client-server. [↑](#footnote-ref-1)
2. See Appendix for RGB, CYMK all that schmuk. [↑](#footnote-ref-2)
3. API an abbreviation for Application Programming Interface. Essentially a software library for the specific purpose of application development.  
   AV an abbreviation for Audio and Video. [↑](#footnote-ref-3)
4. The scope of the AVINSoR project is concerned with the design and implementation of an *intelligent* *autonomous navigation system.* The LEGO® Mindstorms® NXT kit is to *demonstrate* the [↑](#footnote-ref-4)
5. A sensor is *passive* if it does not require power (i.e. LEGO® NXT kit’s Touch Sensor, Light Sensor, Sound Sensor and Temperature Sensor).

   Though LEGO® does not manufacture ­*self-powered* sensors (i.e. sensors that draw power from external power sources), such sensors *may* exist for the NXT, and there is a possibility that such sensors *may* be developedby users (if found necessary for a particular application e.g. where a sensor requires greater power than can be sourced from the NXT’s own power supply; or *if* NXT’s power supply, power management, or *both*, need to be superimposed in a complex application). [↑](#footnote-ref-5)
6. [↑](#footnote-ref-6)
7. See NXT brick hardware schematic in appendix a. [↑](#footnote-ref-7)
8. Example of a *‘perception’* may be: distance, reflectivity, loudness, etc. [↑](#footnote-ref-8)
9. Furthering our interest in more accurate control of the motors, and therefore the accuracy of *our* implementation of the Cartesian Drive Mapping algorithm, a particular RXE executable is used to read motor control commands and act as a synchronous *feedback* control system (i.e. for both motors) – producing a visible difference in the accuracy of movements (see *testing*). This method of motor control is covered in greater depth under System Implementation (page x). [↑](#footnote-ref-9)
10. The Cartesian grid’s absolute minimum value (-gridSize) and maximum value (+gridSize) may be mapped to the integer range indicated by minimum power output to the motor (motorOutputMin) and maximum power output to the motor (motorOutputMax).  
    *This will be particularly useful in converting a grid induced value (ranging from –gridSize to +gridSize) into a motor output value (ranging from motorOutputMin to motorOutputMax).* [↑](#footnote-ref-10)