Sensors

Prof. Riccardo Berta

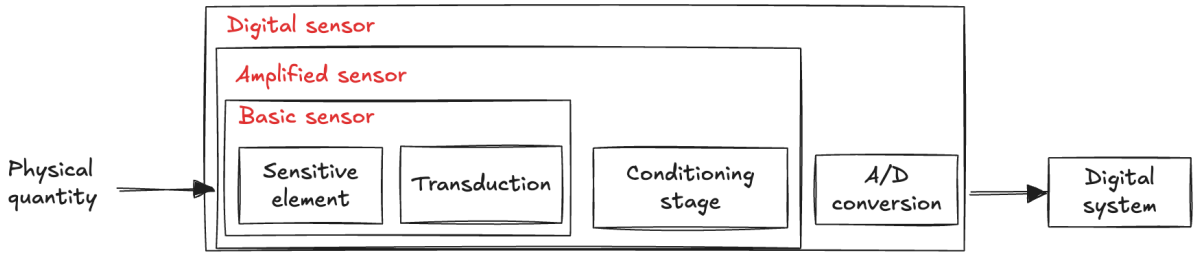
2025-03-20

Contents

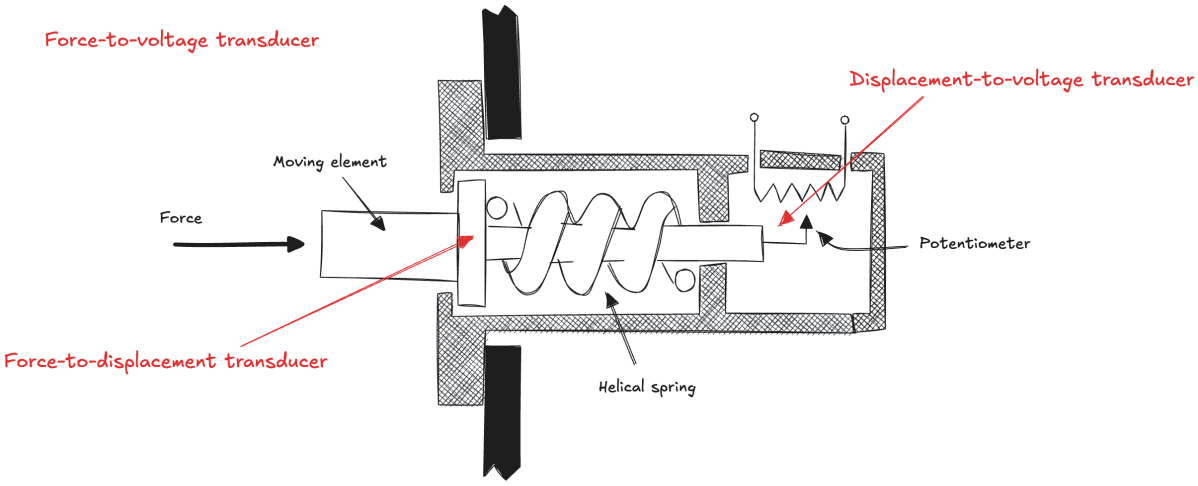
1 Sensors

A sensor is a device that detects **physical properties or changes in the environment** (such as temperature, light, motion, pressure, or sound) and converts this information (called **stimulus**) into **an electrical signal** that can be read and processed by an electronic system. The following image presents

a simplified block diagram of a digital sensor system, detailing its components and their roles in the measurement process:



A physical quantity, which represents a real-world phenomenon the sensor is designed to measure, such as temperature, pressure, or light intensity. This quantity is detected by a sensitive element, the core component of the sensor that interacts directly with the physical environment. The sensitive element converts the physical input into a measurable electrical signal, a process known as **transduction**. In some cases, the resulting signal may be too weak for effective processing, necessitating a **conditioning stage**, often involving amplification. This step ensures that the signal reaches an appropriate strength for further handling. The next step is **analog-to-digital conversion (A/D)**, where the analog electrical signal is transformed into a digital format. This conversion is essential for compatibility with digital systems, which interpret and process the data. Finally, the digital system takes over, performing operations such as filtering, calibration, data analysis, or control to make the information usable for applications. As a simple example, we can consider a force transducer setup:



An external force is applied to a moving element, which is typically a piston or a similar component. This force causes the moving element to compress a helical spring. As the spring compresses, it results in the displacement of the moving element. This displacement is then measured by a displacement-to-voltage transducer, which often uses a potentiometer, a type of variable resistor. As the moving element shifts, it alters the resistance of the potentiometer, which in turn changes the output voltage.

The change in voltage is directly proportional to the amount of displacement. As a result, the output voltage provides a measure of the force applied to the system.

The electrical signal generated by the sensor depends on its type and the amount of information it needs to convey. Some sensors, such as photoresistors and piezo knock sensors, are made from materials that change their electrical properties in response to physical changes. Other sensors, however, are more advanced electronic modules that include their own microcontroller to process data before transmitting the signal to the microcontroller for further use. All sensors can be broadly classified into two categories: passive and active. A **passive sensor** operates without requiring any external energy source. It directly converts the energy from an external stimulus into an electrical signal, making it self-sufficient in generating an output. For instance, a thermocouple generates a voltage in response to a temperature difference, a photodiode produces current when exposed to light, and a piezoelectric sensor generates an electric charge under mechanical stress. In contrast, **active sensors** depend on an external power source, known as an excitation signal, for their operation. The sensor modifies this signal to produce the desired output. For example, a thermistor, which is a temperature-sensitive resistor, does not generate an electrical signal by itself. Instead, when an electric current is passed through it, its resistance can be measured by observing the variations in current or voltage across it. These variations, expressed in ohms, correspond to temperature values through a predefined relationship.

1.1 Specifications

A sensor performance is defined by a set of **key parameters** that characterize its capabilities. These specifications determine how well a sensor can convert real-world signals into electrical signals for further processing or feedback. In this section, we will explore the primary specifications of sensors, focusing on how they impact sensor selection and performance. We will examine concepts like accuracy, response time, linearity, and noise levels, among others. Notice the importance of environmental factors, such as temperature and humidity, that can affect sensor performance in real-world applications. By understanding specifications, we can make **informed decisions** when designing systems that rely on accurate and reliable sensor data.

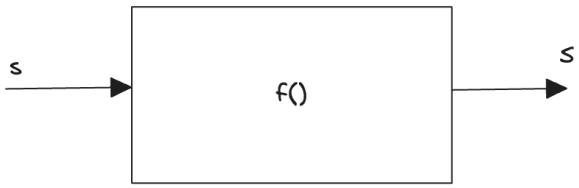
1.1.1 Transfer function

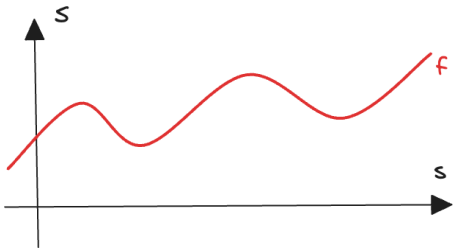
An ideal output-stimulus relationship exists for every sensor, representing how the output would perfectly correspond to the input if the sensor were manufactured under **ideal conditions**. In such a theoretical scenario, the sensor would always provide an accurate and true representation of the measured stimulus. This ideal relationship between the output and the stimulus is captured by the **transfer function**, which defines how the electrical signal produced by the sensor depends on the stimulus.

Mathematically, this can be written as

$S = f(s)$

where S is the output signal (one of the characteristics of the output electric signal used by the data acquisition devices as the sensor output, like amplitude, frequency, or phase, depending on the sensor properties and s is the input stimulus.



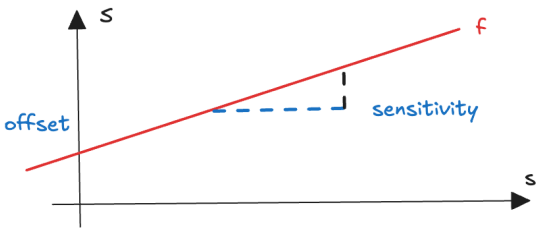


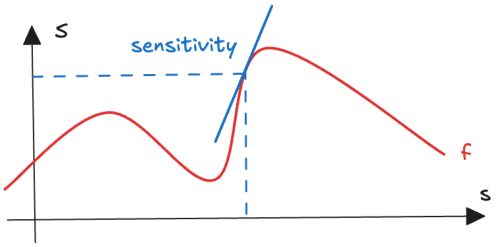
The transfer function might be a **simple linear relationship**, such as

$S = a + b \cdot s$

where a represents the intercept or the output signal at zero input (**offset**), and b is the slope, also known as the **sensitivity** of the sensor. It reflects how strongly a sensor output reacts to changes in its input. Intuitively, it measures how much the sensor "notices" and responds to variations in the stimulus. A sensor with high sensitivity will produce a significant change in output even for a small change in the input, making it excellent for detecting subtle variations. Conversely, a sensor with low sensitivity requires larger changes in the input to produce noticeable changes in the output, making it less responsive to minor fluctuations. Alternatively, the function may be **non-linear**, taking forms such as logarithmic, exponential, or power functions. For non-liner sensors, sensitivity varies with the input value. At a specific input value, the sensitivity can be defined as the derivative of the output signal with respect to the input:

$b = \frac{dS}{ds}\Big|_{s_0}$





In this cases the transfer function may still be approximated **as linear over small input ranges**. This approximation leverages the fact that, for sufficiently small changes in the input, the function behaves almost like a straight line. Mathematically, the transfer function can be locally approximated by a linear equation using a first-order Taylor expansion around a specific input value:

$$S \approx f(s_0) + \frac{df}{ds}\Big|_{s_0} \cdot (s - s_0)$$

However, this linear approximation is valid only for small changes in the input, as the function may deviate significantly from linearity for larger variations.

For broader ranges, a **piecewise approximation** involving several linear segments is often used to model the non-linear behavior. The transfer fucntion is divided into segments with inout ranges:

$$[s_0, s_1], [s_1, s_2], \dots, [s_{n-1}, s_n]$$

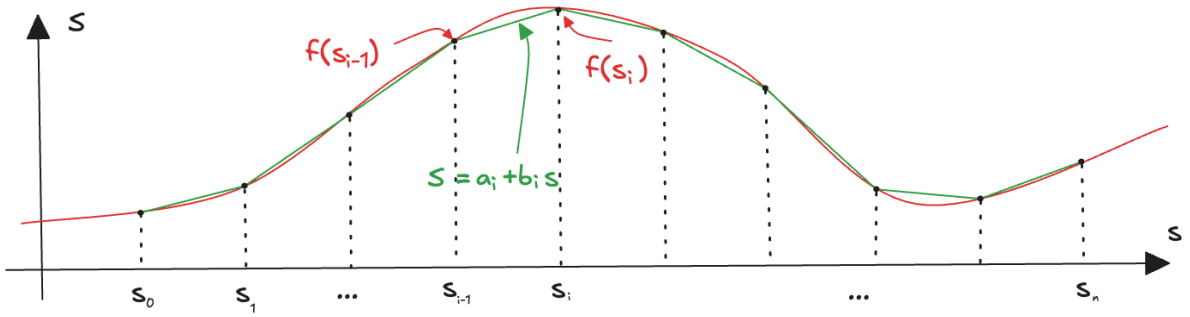
In each segment, the transfer function can be approximated as:

$S = a_i + b_i \cdot s$ for $s \in [s_{i-1}, s_i]$

To ensure continuity between segments, the values of intercept and sensitivity of each linear segment are determined such that the endpoints of each segment match the actual transfer function value:

$$f(s_{i-1}) = a_i + b_i \cdot s_{i-1}$$

$$f(s_i) = a_i + b_i \cdot s_i$$



This ensures that the piecewise approximation closely follows the original non-linear transfer function across the entire input range. The accuracy of the piecewise approximation improves as the number of segments increases, but at the cost of increased complexity in modeling and computation.

The transfer function can be also **multi-dimensional**, involving multiple input variables. In this case, the transfer function would be a function of multiple variables, such as

$$S = f(s_1, s_2, s_3, ...)$$

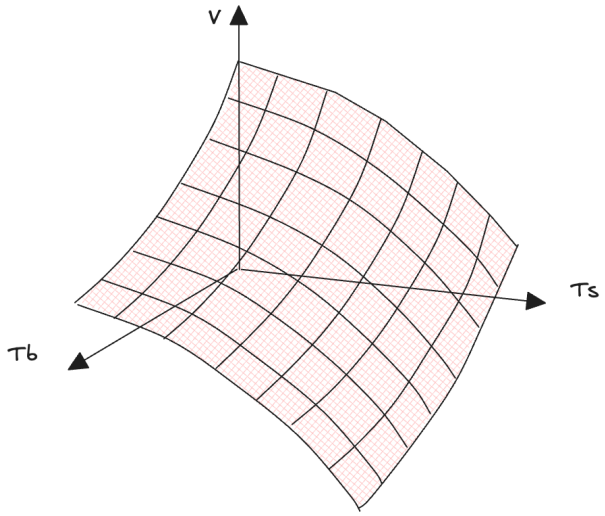
The sensitivity can be defined as the **partial derivative** of the output signal with respect to the particular input variable:

$$b_i = \frac{\partial S}{\partial s_i}\Big|_{s_0}$$

A common example is the transfer function of a **thermal radiation sensor**, where the output voltage V depends on both the absolute temperature of the measured object (Tb) and the absolute temperature of the sensor's surface (Ts). The relationship is described by the equation:

$V = G(T_b^4 - T_s^4)$

Where G is a constant that depends on the sensor's properties. We can show the transfer function as a 3D plot:



This transfer function shows not only non-linearity, represented by the fourth power dependence, but also a dependence on the sensor own surface temperature. To determine the sensitivity to the object temperature, the partial derivative of the output voltage with respect to Tb is calculated as:

$b = \frac{\partial V}{\partial T_b} = 4GT_b^3$

Summarizing, the transfer function is a crucial concept in sensors, as it provides a mathematical model for understanding how the sensor responds to different stimuli. By characterizing the transfer function, engineers can determine several sensor properties, like sensitivity, linearity, and range of operation, enabling them to optimize the sensor performance for specific applications.

1.1.2 Dynamic Range

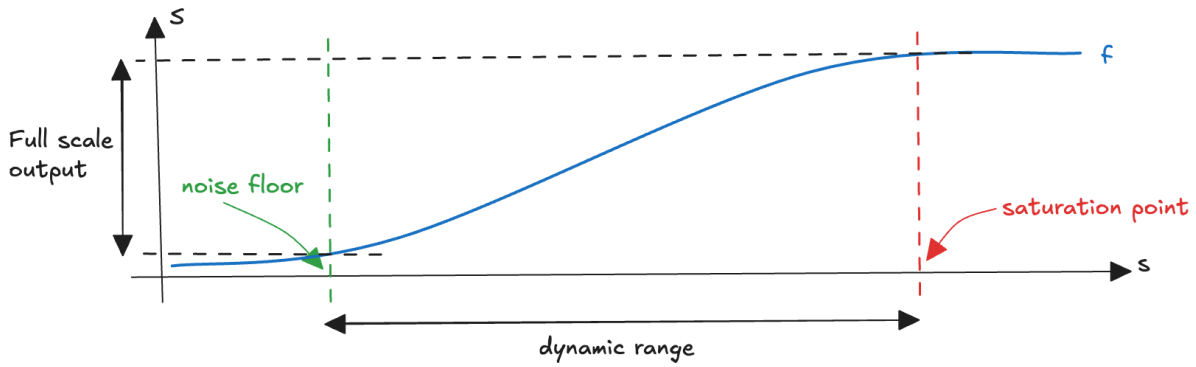
The **dynamic range** quantifies the sensor ability to **operate effectively across a spectrum of signal intensities** without distortion or loss of information. The upper limit is determined by its **saturation point**, the maximum signal intensity that the sensor can handle before it becomes incapable of producing a valid output. Every sensor has its operating limits. Even if it is considered linear, at some levels of the input stimuli, its output signal no longer will be responsive, a further increase in stimulus does not produce a desirable output. It is said that the sensor exhibits a saturation. Beyond this point, the response plateaus, leading to a loss of detail in high-intensity regions. On the other end, the lower limit is defined by the **noise floor**, the smallest signal the sensor can distinguish from inherent electronic noise, which arises from thermal or electrical fluctuations within the sensor system.

The **Full-scale output** is often also specified together with the dynamic range and it is defined as the difference between the electrical signals measured when the maximum and minimum stimulus are applied as input. The full-scale output represents how the dynamic range is mapped to the output signal:

$$DR = s_{max} - s_{min}$$

$$FS = S_{max} - S_{min}$$

Where s_{max} is the saturation point and s_{min} is the noise floor and S_{max} and S_{min} are the corresponding output signals



A high dynamic range is particularly desirable. For instance, in imaging sensors, a wide dynamic range allows the capture of scenes with both bright and dark areas, preserving details in both shadows and highlights. Similarly, in audio sensors like microphones, a high dynamic range ensures the accurate representation of both faint whispers and loud sounds without distortion. For sensors with very broad and nonlinear response characteristic, the dynamic range is often expressed in **decibels**, which is a logarithmic scale used to express the ratio between two values. The logarithmic nature compresses large ranges, making them more manageable:

$$DR_{dB} = 10 \cdot \log_{10}\left(\frac{s_{max}}{s_{min}}\right)$$

We can use decibels for the output signal as well. If the sensor output spans several orders of magnitude, expressing the output in decibels can help compress the range into a more compact and readable form:

$$S_{dB} = 10 \cdot \log_{10}\left(\frac{S}{S_{ref}}\right)$$

where S_{ref} is a reference signal, often the smallest measurable signal or the sensor noise floor.

The logarithmic nature of the decibel scale is particularly useful for describing situations where the **human perception** follows a similar logarithmic pattern. The human eye or ear, have dynamic ranges that align well with the logarithmic nature of decibels. For instance, the human ear perceives loud-

ness on a logarithmic scale, so decibels are an intuitive way to describe the perceived loudness of sounds.

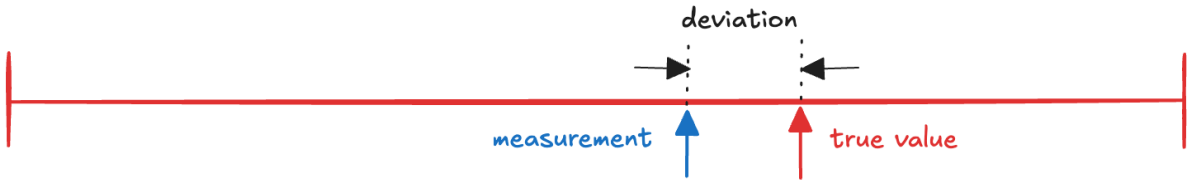
1.1.3 Accuracy

A sensor is a real device to measure a physical quantity, and as such, it is subject to **imperfections** and **uncertainties** that can affect its measurements. We define as **deviation** the difference between the value of the physical quantity measured by the sensor and the actual value of the quantity. Mathematically, it can be expressed in terms of absolute error and relative error. The **absolute error** represents the difference between the measured value and the true value:

$$\text{Absolute Error} = \text{Measured Value} - \text{True Value}$$

The **Relative error** expresses the deviation of the sensor as a fraction of the true value. It provides insight into how significant the error is in comparison to the scale of the measured quantity:

$$\text{Relative Error} = \frac{\text{Absolute Error}}{\text{True Value}}$$



As an example, consider a displacement sensor that ideally generate 1 mV per 1-mm displacement. Its transfer function is linear with a slope of 1 mV/mm:

$$S[mV] = 1[mV/mm] \cdot s[mm]$$

Howerver, in an experimental setup, a displacement of 10 mm produced an output of 10.5 mV. Converting this value into the displacement using the inversed transfer function:

$$s[mm] = \frac{S[mV]}{1[mV/mm]} = 10.5[mm]$$

The deviation is then:

$$\text{Deviation} = 10.5[mm] - 10[mm] = 0.5[mm]$$

The accuracy of the sensor is defined as the **maximum deviation** between the measured value and the true value of the physical quantity. It refers to **how closely** the output of a sensor matches the true value of the measured quantity. It is one of the most important specifications when evaluating a sensor, as it determines how well the sensor's reading represents the actual physical quantity. In practice, sensor accuracy is not always perfect due to various sources of error, such as calibration inac-

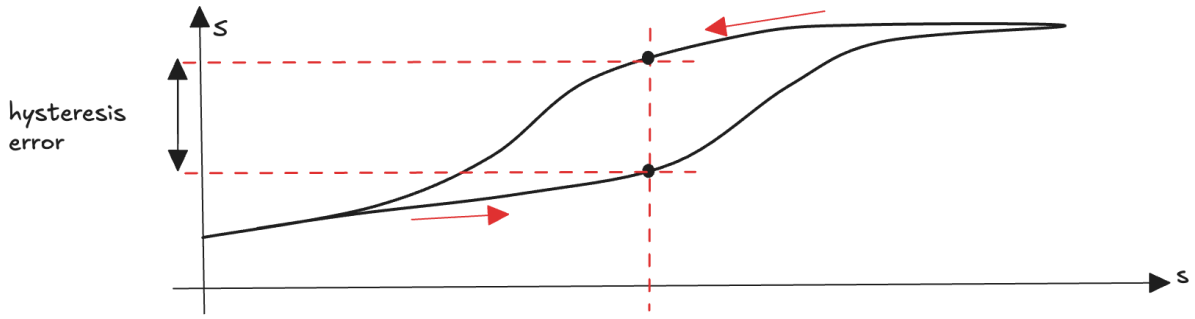
curacies, noise, or environmental factors. In many applications, the accuracy of a sensor is described as a percentage of the sensor full-scale reading:

$$\text{Accuracy} = \frac{\text{Max Absolute Error}}{\text{Full-scale Reading}} \times 100\%$$

1.1.4 Hysteresis

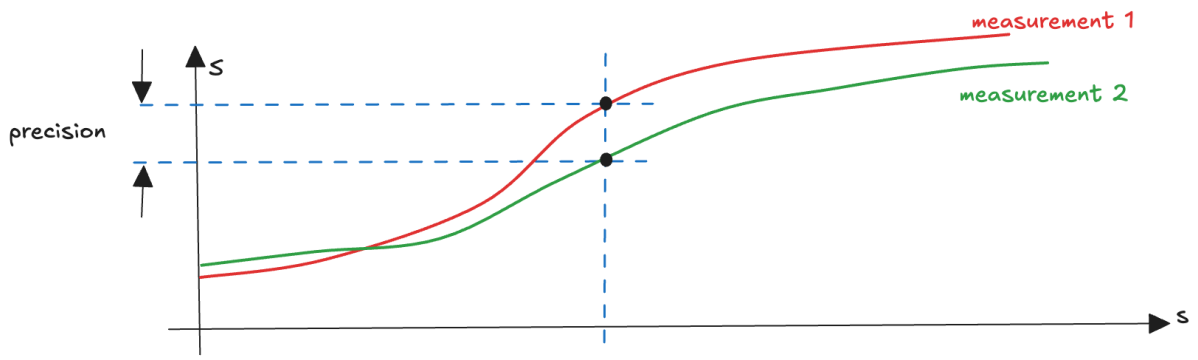
An **hysteresis error** is a deviation of the sensor output at a specified point of the input signal when it is approached from the **opposite directions**. For example a sensor that measures the position of a moving object may produce different output values when the object is moving towards the sensor than when it is moving away from it. It is often caused by mechanical or thermal effects within the sensor that introduce **a memory effect** in the output signal. Hysteresis can be quantified by measuring the difference between the output values when the input signal is increasing and when it is decreasing. The hysteresis error is defined as the **maximum difference** between the two output values:

$$\text{Hysteresis Error} = \max\left(S_{\text{increasing}} - S_{\text{decreasing}}\right)$$

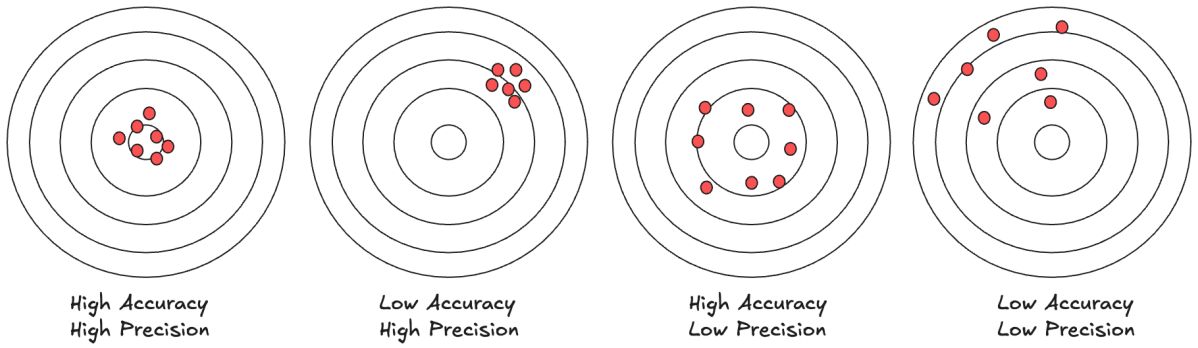


1.1.5 Precision

The **precision** of a sensor refers to the ability of a sensor to produce consistent output readings when the same input stimulus is applied multiple time under identical conditions. In simple terms, if you measure the same physical quantity multiple times with the same sensor, the sensor should give you very similar results each time, even if we perform the measurements at different moments. This reflects the **repeatability** of the sensor performance, which is a crucial aspect of its overall reliability.



Accuracy and precision are closely related but distinct concepts. Accuracy refers to how close the sensor output is to the true value of the measured quantity, while precision focuses on the consistency of the sensor output when measuring the same quantity multiple times. A sensor can be accurate but not precise, if it consistently produces the same error in its measurements. Conversely, a sensor can be precise but not accurate if it consistently produces the same output value that is different from the true value:



1.1.6 Resolution

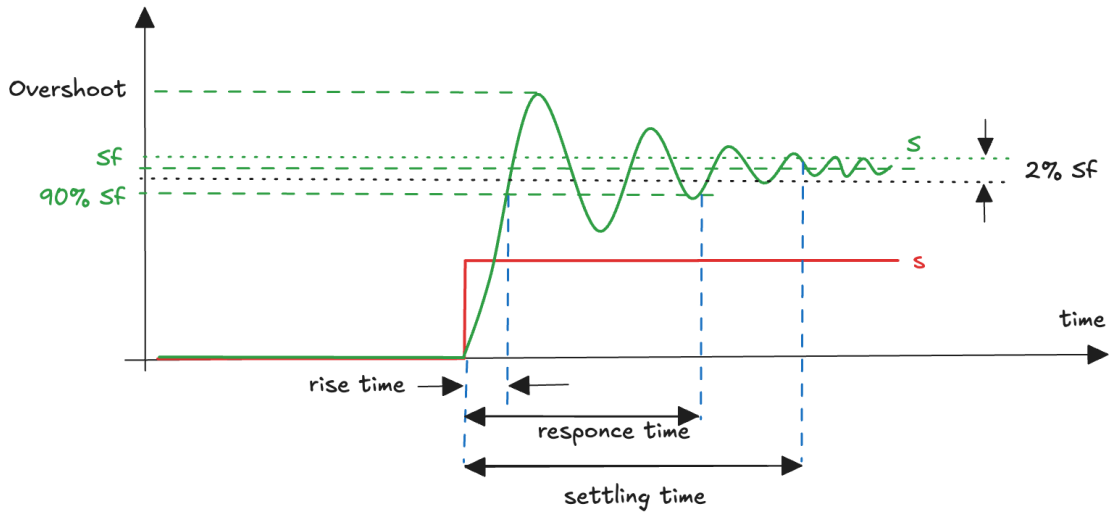
The resolution is **the smallest detectable change** in the measured quantity that the sensor can distinguish. It defines the sensor ability to detect fine details. For analog sensors, the resolution is influenced by the noise and the precision of the measurement system. In digital sensors, the resolution is typically determined by the **number of bits** in the sensor's analog-to-digital converter (ADC).

1.1.7 Dynamic characteristics

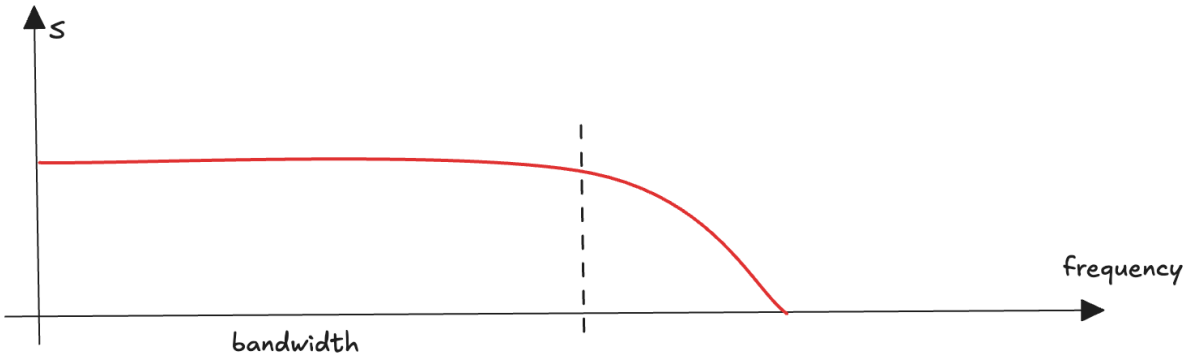
The dynamic characteristics of a sensor refer to **how the sensor responds to changes in the input signal over time**, particularly its ability to track rapidly changing or varying inputs. These characteristics describe how well the sensor can detect, follow, and accurately measure signals that are not

constant, such as fluctuating or transient inputs. Key dynamic characteristics include:

- **Rise Time**: the time it takes for the sensor output to go from 0 to an higher value, often 90% of the final output. It focuses only on the initial portion of the transition and measures how quickly the output ramps up to the near-final value.
- **Responce Time**: the time it takes for the sensor to initially react to a change in the input signal and reach a significant portion (90%) of the final output value. It includes both the rise phase and potentially part of the stabilization phase, focusing on the overall responsiveness of the sensor to the change.
- **Settling Time**: the time it takes for the sensor to stabilize within a certain range after a step change in input, tipycally within 2% of the final value. It is about how long it takes for the sensor to stop fluctuating and produce a stable output.
- **Overshoot**: the extent to which the sensor' output exceeds the final value during a transient response before stabilizing



Tipically, we characterize the sensor from this point of view using the **frequency response**, which shows the magnitude and phase of the sensor output as a function of the frequency of the input signal. We can define the **bandwidth** of the sensor as the range of frequencies over which the sensor can accurately measure the input signal:



1.1.8 Reliability

The **reliability** of a sensor refers to its ability to consistently perform accurate measurements under specified conditions over a defined period. It is typically expressed as the **probability** that the sensor will operate without failure or degradation in performance for a given duration or number of usage cycles. Tipically it is espressed using the **mean time between failures (MTBF)**, which is the average time that a sensor can operate without failing.

$$MTBF = \frac{\text{Total Operating Time}}{\text{Number of Failures}}$$

The reliability of a sensor can be influenced by various factors, such as the quality of the sensor components, the manufacturing process, the operating conditions, and the maintenance practices. A reliable sensor is essential for ensuring the accuracy and consistency of the measurements it provides, especially in critical applications where sensor failure can have serious consequences.

1.2 Calibration

Calibration is the process of **adjusting and verifying the accuracy** of a sensor by comparing its output to a known reference. It is a critical step in ensuring that the sensor provides accurate and reliable measurements. Calibration is necessary because sensors can exhibit **drift** over time, where their output changes due to factors like aging, temperature variations, or mechanical stress. By calibrating a sensor, we can correct for these changes and ensure that its output remains accurate and consistent. In order to understand the calibration process, let's consider a simple example. Suppose we have a linear sensor designed to measure a physical quantity, like a force F and produce an output voltage V. The relationship between the input force and the output voltage is assumed to be linear:

$$V = m \cdot F + b$$

And suppose that from the datasheet we know that the sensor has a sensitivity of m=0.2V/N and an

offset of b=0.4V. However, we don't know if in practice the sensor behaves exactly as expected and we want to calibrate it. The calibration process typically involves the following steps:

1.2.1 Establishing a reference

The first step is to identify a known reference value for the physical quantity being measured. In the case of our linear sensor we can apply two known forces F1 and F2:

$F_1 = 10N, F_2 = 20N$

The reference can be obtained from another calibrated sensor or using a physical standard with a known value. The reference should be traceable to a recognized standard to ensure its accuracy.

1.2.2 Exposure to the reference value

The sensor is then exposed to the reference value, and its output is measured. In our case, we can apply the forces F1 and F2 and measure the output voltages V1 and V2:

$V_1 = 2.5V, V_2 = 4.7V$

Normally, it is not suffient to measure the output only once, but it is necessary to repeat the measurement several times to account for variability and noise in the sensor output.

Then, we can calucalate the difference between the real sensor output and the value of the sensor output expected from its transefer function when exposed to the reference value. From the datasheet, the expected output voltages for the forces F1 and F2 should be:

$eV_1 = 0.2 \cdot 10 + 0.4 = 2.4V \; eV_2 = 0.2 \cdot 20 + 0.4 = 4.4V$

The difference is called calibration error.

1.2.3 Adjusting the sensor

If the calibration error is significant, the sensor may need to be adjusted to correct for the error. This adjustment can involve changing the sensor settings, recalibrating its internal components, or applying correction factors to its output. In our case we can recalculate the parameter of its linear transfer function by solving the system of equations:

$m \cdot 10 + b = 2.5$

$m \cdot 20 + b = 4.7$

That provides the new values of the sensitivity and the offset:

$m = 0.21V/N, b = 0.3V$

1.2.4 Validation

After adjusting the sensor, the calibration process is repeated to verify that the sensor output now matches the reference value within an acceptable tolerance. The sensor is considered calibrated if the calibration error falls within the specified limits. However, this is a simple example becouse we take only to points supposing that the sensor is still linear. In practice, calibration is often more complex, involving multiple reference points, non-linear corrections, and detailed analysis of the sensor's behavior. The calibration process is essential for ensuring the accuracy and reliability of sensor measurements, especially in critical applications where precise measurements are required. By calibrating sensors regularly and following best practices, we can maintain the quality and performance of sensor systems over time.

In general, the theory of measurement is a complex field that involves a deep understanding of physics, electronics, and signal processing. The design and implementation of sensor systems require careful consideration of the sensor characteristics, the measurement environment, and the intended application. By understanding the key concepts of sensors, specifications, and calibration, we can develop effective sensor systems that provide accurate, reliable, and consistent measurements for a wide range of applications.

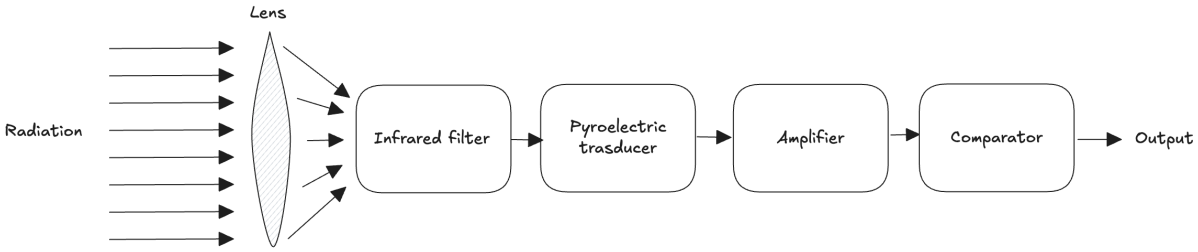
1.3 Context Awareness

Sensors are essential for detecting and quantifying various **environmental parameters**, enabling embedded systems to make informed decisions. For example, sensors can regulate room temperature, adjust display brightness, or detect the presence of objects. This task is inherently complex because the environment is a dynamic, multifaceted system with numerous interacting components. By measuring a wide range of environmental parameters, sensors play a key role in achieving what is often referred to as **context awareness**, allowing systems to adapt to and interact intelligently with their surroundings.

1.3.1 Detecting presence

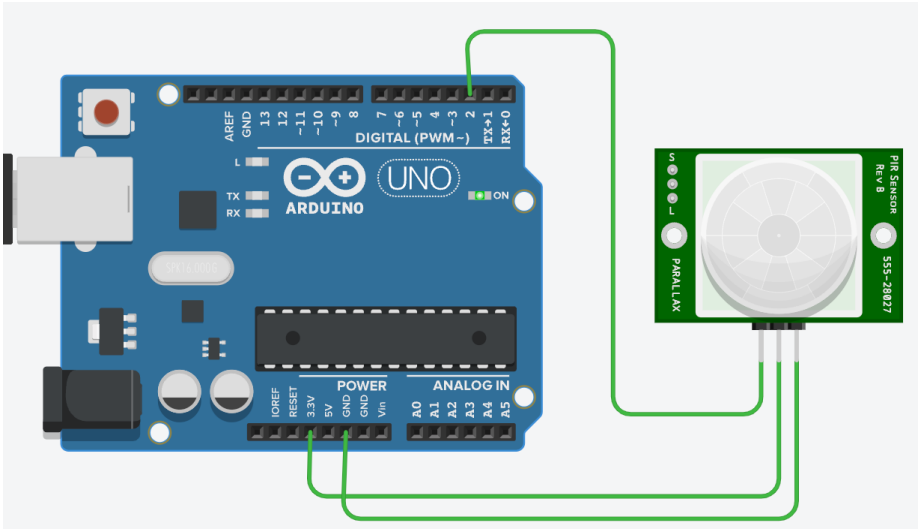
One of the most common applications of sensors is detecting the presence or absence of an person in a room. This is a fundamental task in many systems, from industrial automation to consumer electronics. Sensors used for presence detection are often designed to respond to specific stimuli, such as light, sound, pressure, or motion. The choice of sensor depends on the environmental conditions in which the detection is performed. In this example, we exploit a **Passive Infrared PIR sensor**, which is a device that **captures changes in infrared radiation emitted by objects** in its field of view. When a person enters the sensor range, their body heat causes a change in the infrared radiation detected by

the sensor, triggering an output signal. The PIR sensor is sensitive to changes in temperature, making it ideal for detecting the presence of warm-blooded animals, such as humans. In particular we will use the AM312 PIR sensor (see *01.DetectingMotion* folder). The AM312 sensor consists of the following main components:



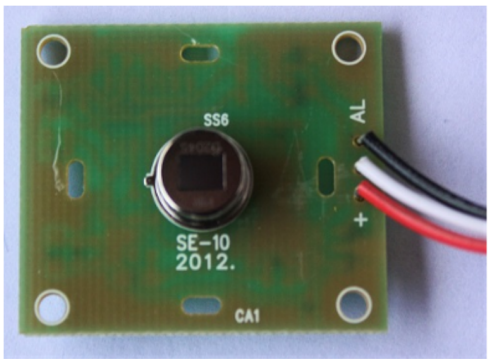
The **lens** on the outside of the sensor expands the detection area of the sensor by focusing infrared radiation from multiple directions onto the sensor. Then the **infrared filter** allows only IR radiation in the 8–14 µm wavelength range (typical for human body heat) to reach the **pyroelectric transucer**, which exploit the properties of some crystals to generate an electric charge when they are heated or cooled. This **sensitive element** converts the infrared radiation into an electrical signal. The raw signal generated is weak and requires **amplification** and **filtering**. The processed signal is sent to the output pin, which provides a digital output: HIGH (e.g., 3.3V or 5V, depending on the supply voltage) when motion is detected or LOW (0V) when no motion is detected.

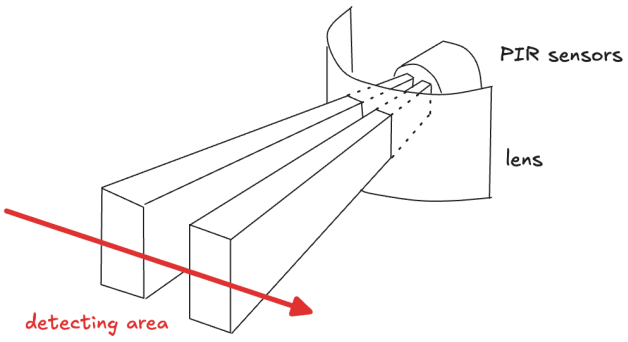
In the Makers Kit we have a PIR sensor that we can connect to the Arduino board to detect presence:



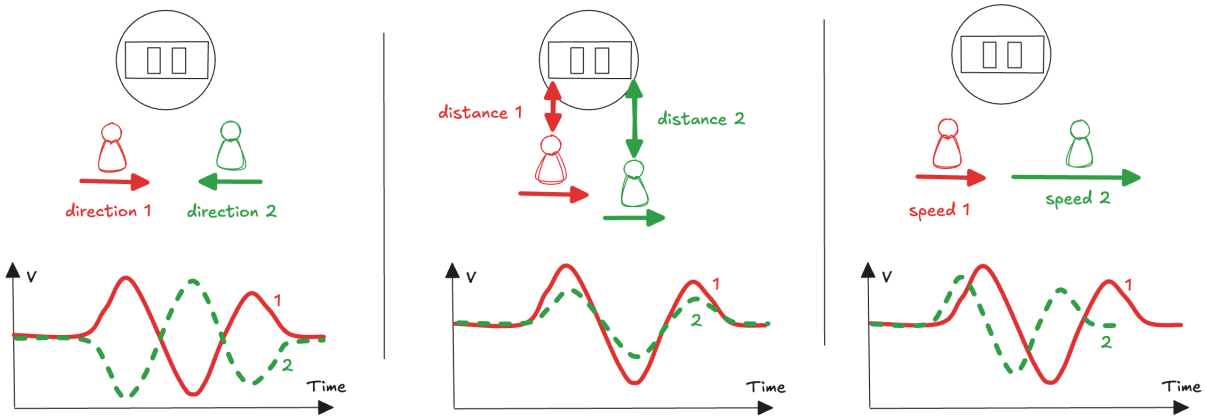
```
// define the pin that the sensor is attached to
int pir_pin = 2;

// a variable to count the number of times the sensor has been triggered
int count = 0;

void setup() {
  Serial.begin(9600);
  pinMode(pir_pin, INPUT);
}

void loop() {

  // detect the output of the sensor
  int pir_val = digitalRead(pir_pin);

  // if the sensor is triggered, increment the count
  if(pir_val == HIGH) {
    count++;

    // send the count to the serial port
    Serial.print("Motion Detected: ");
    Serial.println(count);

    // The sensor needs 2 seconds before it can detect motion again
    delay(2000);
  }
}
```

We can also leverage the analog output of the PIR sensor to gain more detailed insights, not only about presence but also about movement. As an example, consider the following scientific paper "Human Movement Detection and Idengification Using Pyroelectric Infrared Sensors" investigates human movement identification using a pair of orthogonally aligned PIR sensors with modified lenses:





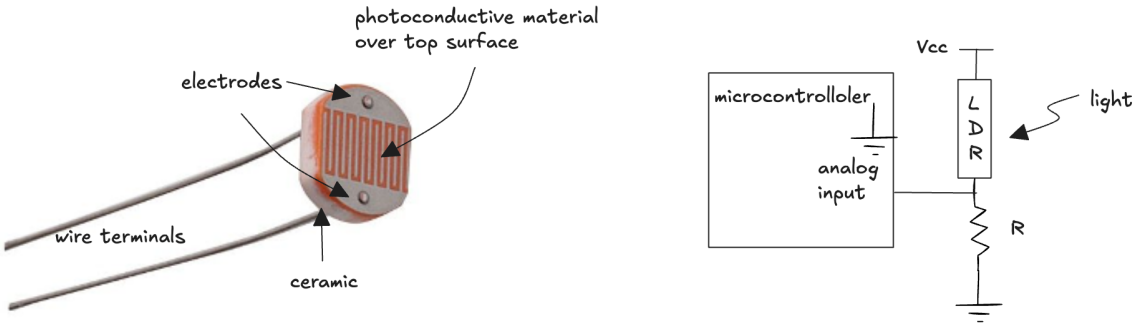
The researcher collect raw data of walking in different directions, at different distances and at different speed levels:



Classification analysis was conducted using machine learning algorithms demonstrated the possibility of classify movement direction, speed, and distance intervals, as well as identifying individuals with high accuracy.

1.3.2 Detecting Light

A simple and effective method for detecting light involves using a **Light Dependent Resistor (LDR)**, also known as a **photoresistor**. This sensor operates using semiconductor materials, such as Cadmium Sulfide (CdS), whose electrical resistance changes with varying light intensity. The underlying principle is **photoconductivity**: when light strikes the material, its energy excites electrons from the valence band to the conduction band. This process increases the material's conductivity as light intensity rises, thereby decreasing its resistance. By integrating the LDR into **a voltage divider circuit**, it produces a variable voltage output that can be read by the analog input of a microcontroller. In the Makers Kit we have a LDR sensor that we can connect to the Arduino board to detect light:

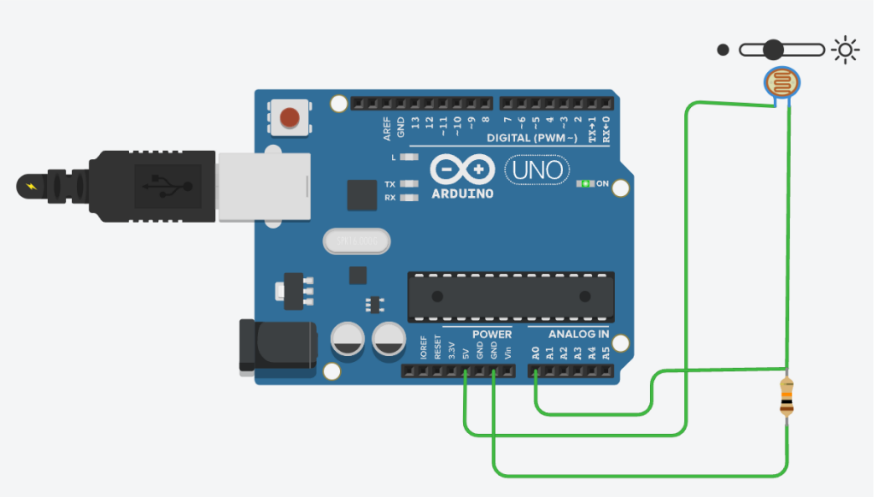


The voltage across either the LDR or the resistor can be determined using the following formula:

$$V_{out} = Vdd \cdot \frac{R}{LDR + R}$$

The value of R has to be defined by the user, and it is typically chosen to be in the same order of magnitude as the LDR resistance in the dark. The LDR resistance can vary significantly depending on the light intensity, ranging from a few KOhms in bright light to several MOhms in darkness.

he code for reading the LDR sensor reading is simple as reading from an analog pin (see *02.Detecting-Light* sketch):

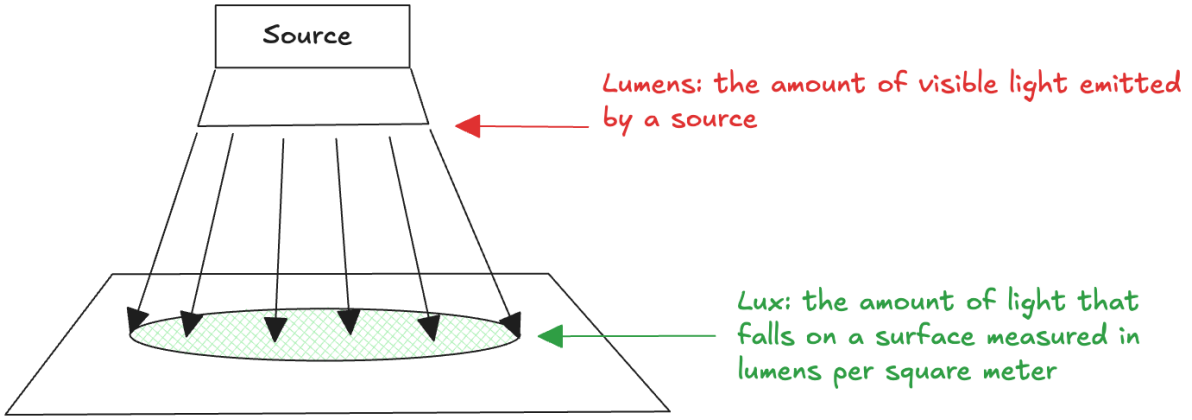



```
// select the input pin for the photoresistos
const int pin = 0;

// variable to store the value coming from the sensor
int val = 0;

void setup() {
  Serial.begin(9600);
}

void loop()  {
  // read the voltage on the photoresistor
  val = analogRead(pin);

  // print the value to the serial port
  Serial.println(val);
}
```

The LDR is a simple and cost-effective sensor for detecting light levels, making it suitable for applications like automatic lighting control, light intensity measurement, and light-sensitive alarms. However, it has several limitations, since the relationship between light intensity and resistance is **highly non-linear**, which means it can be difficult to directly correlate resistance values with light intensity without additional calibration or compensation. Another limitation is the **spectral sensitivity**. Photoresistors are typically responsive only to a specific range of wavelengths, usually within the visible spectrum. This makes them unsuitable for measuring light outside this range, such as ultraviolet or infrared light, which is important in some applications that require broader spectrum measurements. Additionally, photoresistors have **slower response times**, which can be problematic in situa-

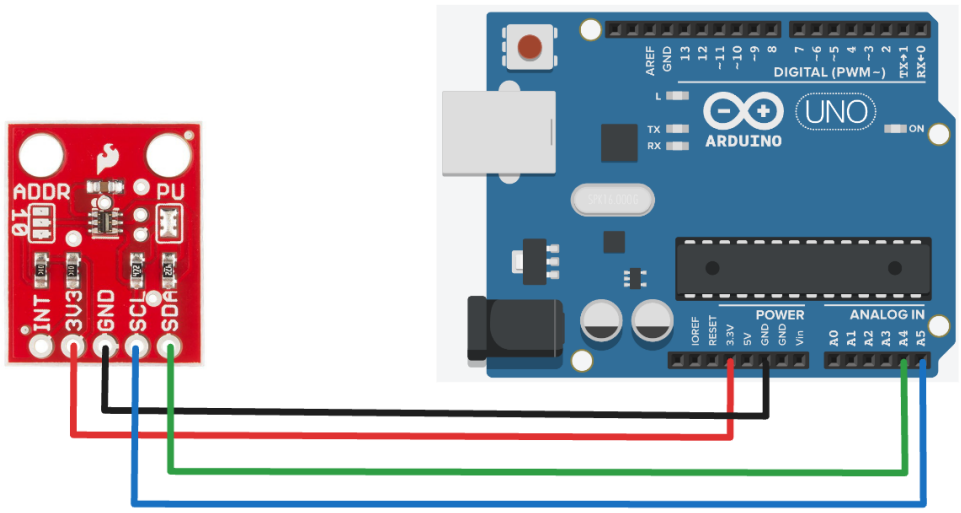
tions where light intensity changes rapidly, as it may fail to track such fluctuations accurately. Finally, another issue is the **sensitivity to temperature**. The resistance of a photoresistor can change with temperature, introducing errors when measuring light in environments where temperatures vary significantly.

For accurate light measurement in terms of **lumens**, which quantify the total visible light emitted by a source, more specialized sensors, are required. Lumen is often quantified in terms of **lux** when considering its intensity per square meter:



Lux is the unit that describes how bright a light source will appear to the human eye, factoring in both the total light output and the area over which it is distributed. The human eye is capable of perceiving a wide range of light intensities, from as low as 0.0001 lux in starlight, up to over 100,000 lux in direct sunlight. This is a vast range, and it is difficult to build a sensor with the same sensitivity across the entire spectrum, which can be very useful in applications like photography.

In the Makers Kit, we have a TSL2561 sensor. Although it is not a real luxmeter (manufacturer has characterized its output against professional equipment to produce lux approximation equations), it is designed to handle the vast differences in light levels by incorporating features that mimic the characteristics of a camera. For example, the sensor has a **sensitivity setting** that is similar to the ASA film rating, allowing it to adjust its response to light levels. Additionally, it includes an **integration time**, which is akin to a camera's shutter speed, helping to control how long the sensor ”exposes” itself to light. By adjusting the sensitivity and integration time, the TSL2561 can effectively measure light in both very dim environments, such as starlight, and extremely bright conditions, like direct sunlight. It is a digital sensor that communicates with the microcontroller using the **I2C protocol**:



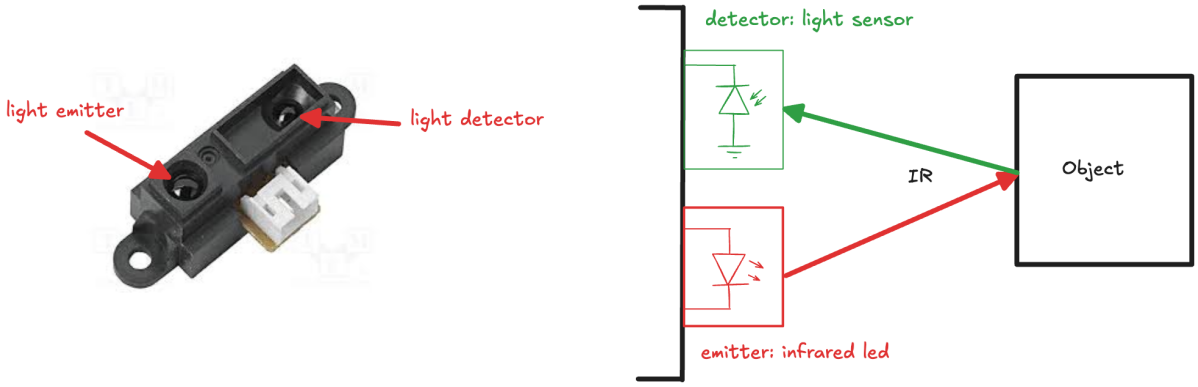
The code to setup and interact with the sensor is provided by the manufacturer with the SparkFunTSL2561 library that can be easily integrated into the Arduino IDE(see the *03.DetectingLight* sketch):

```
// Include the TSL2561 sensor library
#include <SparkFunTSL2561.h>

// Include the I2C library for communication
#include <Wire.h>

// Create an SFE_TSL2561 object to represent the light sensor
SFE_TSL2561 light;

// Global variables to store settings
// Gain setting: 0 = X1 (low gain), 1 = X16 (high gain)
// Integration ("shutter") time in milliseconds
boolean gain;
unsigned int ms;

void setup() {
  Serial.begin(9600);

    // Initialize the TSL2561 sensor.
    // If no I2C address is provided, it uses the default (0x39).
    if (!light.begin()) {
        Serial.println("Error: Sensor initialization failed!");
        while (1);
    }

    // Configure gain
    // gain = 0, device is set to low gain (1X)
    // gain = 1, device is set to high gain (16X)
    gain = 0;

    // Configure integration time
    // intgration_time = 0, integration will be 13.7ms (short integration time)
    // intgration_time = 1, integration will be 101ms (medium integration time)
    // intgration_time = 2, integration will be 402ms (long integration time)
    // intgration_time = 3, use manual start / stop to perform your own integration
    intgration_time = 2;

```

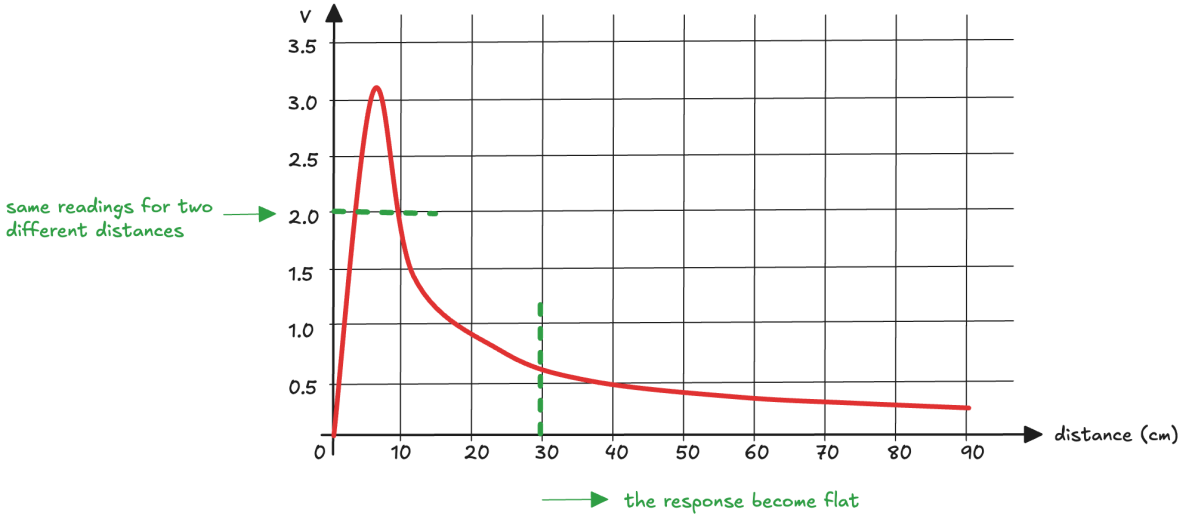
```
    // Set the gain and integration time, the library will provide
    // the timing for the integration time in ms
    light.setTiming(gain, time, ms);

    // Power up the sensor to begin measurements
    Serial.println("Powering up the sensor...");
    light.setPowerUp();
}

void loop() {

  // Read the light sensor data (visible and infrared readings)
  unsigned int data0, data1;
  if (light.getData(data0, data1)) {

    // Print raw sensor data
    Serial.print("Data0 (Visible Light): ");
    Serial.print(data0);
    Serial.print("  Data1 (Infrared Light): ");
    Serial.println(data1);

    // Calculate the lux (light intensity) based on the sensor readings
    double lux;
    boolean good = light.getLux(gain, ms, data0, data1, lux);

    // Print the lux value and indicate if the calculation was successful
    Serial.print("Lux: ");
    Serial.print(lux);
    if (good)
      Serial.println(" (Good measurement)");
    else
      Serial.println(" (Bad measurement: Sensor may be saturated)");

  }

  // If data retrieval fails, print the error
  else {
    byte error = light.getError()č
    switch (error) {
        case 0:
            Serial.println("Success");
            break;
        case 1:
            Serial.println("Data too long for transmit buffer");
            break;
        case 2:
            Serial.println("Received NACK on address (disconnected?)");
            break;
        case 3:
            Serial.println("Received NACK on data");
            break;
        case 4:
            Serial.println("Other error");
            break;
        default:
            Serial.println("Unknown error");
    }
  }

  // Delay before the next measurement cycle
  delay(1000);
}
```

1.3.3 Measuring distance

In order to measure the distance of our embedded system from objects in the sorrounding environment, we can use a **infrared (IR) sensor**. It works by using a specific light sensor to detect a select light wavelength in the infrared spectrum. By using a led which produces light at the same wavelength as what the sensor is looking for, it can look at the intensity of the received light. When an object is close to the sensor, the light from the led bounces off the object and into the light sensor. This results in a large jump in the intensity, which can be used to determine the distance of the object from the sensor. Another possibility is to measure the time it takes for the light to bounce back to the sensor, which is the principle behind **Time-of-Flight (ToF)** sensors. These sensors emit a light pulse and measure the time it takes for the pulse to return after reflecting off an object. By knowing the speed of light, the sensor can calculate the distance to the object. In the Makers Kit we have the Sharp GP2Y0A41SK0F sensor:

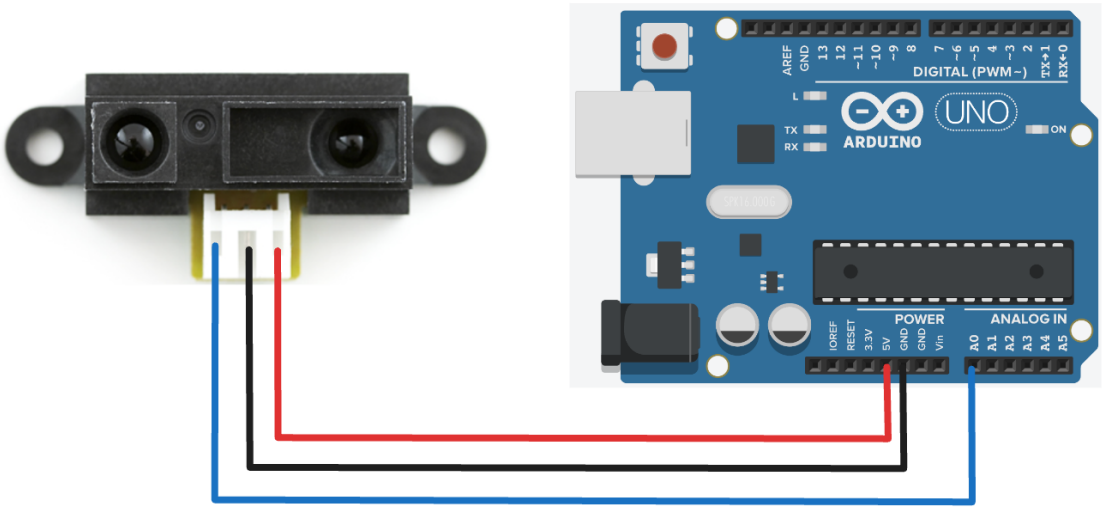


However, the response is non-linear and even worse, it is not monotonic, which means that the output voltage can decrease as the distance increases:



This limitation poses a challenge when using the sensor to measure distance accurately. To address the non-linearity, a **lookup table** can be employed to map the sensor's output to actual distances. The lookup table is created by recording the sensor's output at various known distances and interpolating the values between these points. However, certain distances produce the same output voltage. This overlap means the sensor cannot differentiate between these distances, limiting its effectiveness for objects closer than 4 cm. Additionally, beyond 30 cm, the sensor's response becomes flat, rendering it unable to detect objects farther than this distance.

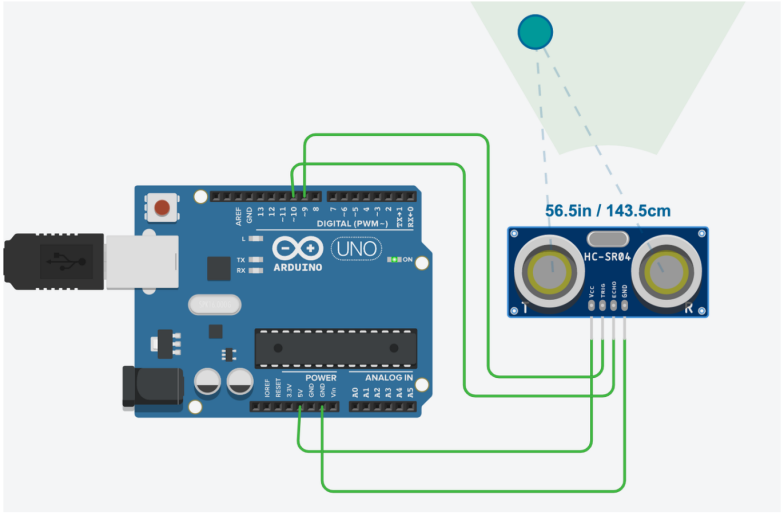
We can consider the code to read the output from the analog IR sensor and convert it into a corresponding distance in centimeters, uding a lookup table with interpolation (see *04.MeasuringDistance* sketch):




```
// Analog pin connected to the sensor
const int SENSOR_PIN = A0;

// Reference voltage in millivolts (5V for most Arduino boards)
const long REFERENCE_MV = 5000;

// Distance table configuration
const int TABLE_ENTRIES = 12;  // Number of entries in the distance table
const int INTERVAL = 250;      // Millivolts separating each table entry

// Distance in cm corresponding to each 250 mV interval
static int distanceTable[TABLE_ENTRIES] = {150, 140, 130, 100, 60, 50, 40, 35, 30, 25, 20,
     15};

// Function to calculate the distance based on the sensor's output voltage
int getDistance(int millivolts) {
    // If the millivolts exceed the last table entry, return the last value
    if (millivolts >= INTERVAL * TABLE_ENTRIES) {
        return distanceTable[TABLE_ENTRIES - 1];
    }

    // Calculate the index of the table entry
    int index = millivolts / INTERVAL;

```

```
    // Calculate the fractional part for interpolation
    float fraction = (millivolts % INTERVAL) / (float)INTERVAL;

    // Perform linear interpolation between the current and next table entries
    return distanceTable[index] - ((distanceTable[index] - distanceTable[index + 1]) *
        fraction);
}

void setup() {
    Serial.begin(9600);
}

void loop() {
    // Read the analog value from the sensor
    int sensorValue = analogRead(SENSOR_PIN);

    // Convert the analog value to millivolts
    int millivolts = (sensorValue * REFERENCE_MV) / 1023;

    // Calculate and print the corresponding distance
    int distance = getDistance(millivolts);
    Serial.print(distance);
    Serial.println(" cm");

    // Wait before the next measurement
    delay(100);
}
```

An alternative approach is to use an **ultrasonic sensor**, which determines distance by emitting ultrasonic sound waves and measuring the time it takes for the waves to bounce back after hitting an object. By knowing the speed of sound, the sensor calculates the distance based on the round-trip time of the sound waves. Ultrasonic sensors offer greater accuracy and reliability across a wide range of distances, typically from a few centimeters to several meters. They are particularly suited for applications requiring precise distance measurements, as they perform well in challenging environments such as dusty, foggy, or misty conditions where optical sensors might face difficulties. However, their accuracy can be influenced by factors such as the texture and angle of the object's surface, which may result in measurement errors. Additionally, their response time is slower compared to infrared sensors, making them less ideal for high-speed applications. In the Makers Kit we have the HC-SR04 sensor, it has two pins for triggering the ultrasonic pulse and receiving the echo:



The microcontroller sends a 10 µs HIGH signal to the Trigger pin to activate the ultrasonic transmitter, which emits 8 cycles of a 40 kHz ultrasonic wave. The sensor waits for the echo to return, to calculate the time we can use the **pulseIn()** function. Finally, the time it takes for the echo to return is proportional to the distance between the sensor and the object, khowing the speed of sound in air, we can calculate the distance:

$$d = \frac{t \cdot v}{2}$$

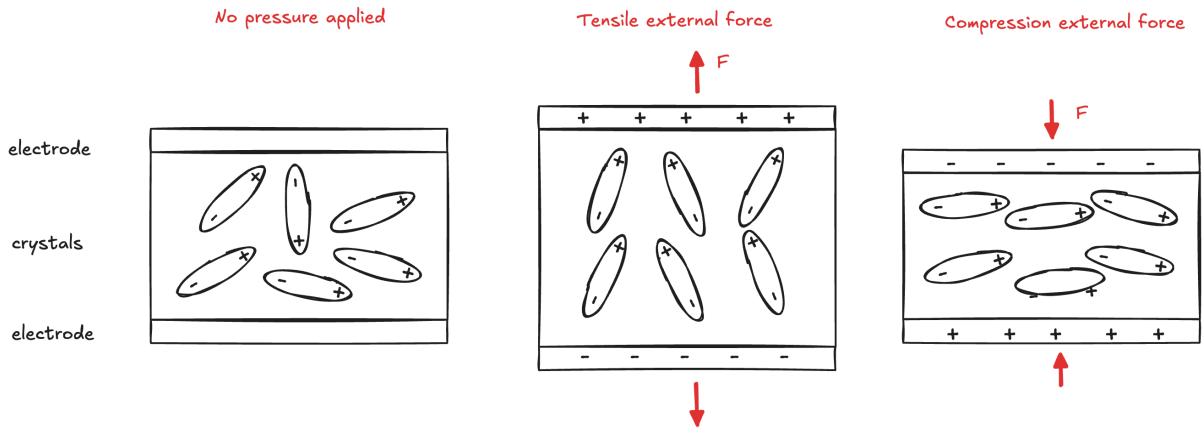
The division by 2 is necessary because the measured time is for the round trip (to the object and back), so the actual distance is half of that time. This code demonstrates how to read the distance (see *05.MeasuringDistance_sound* sketch):

```
// Define the pins for the ultrasonic sensor
const int trigPin = 9;
const int echoPin = 10;

// Define variables for the duration and distance
float duration, distance;

void setup() {
    pinMode(trigPin, OUTPUT);
    pinMode(echoPin, INPUT);
    Serial.begin(9600);
}

void loop() {
    // Send a 10 µs pulse to the trigger pin
    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);

    // Measure the duration of the echo pulse
    duration = pulseIn(echoPin, HIGH);

    // Calculate the distance based on the speed of sound
```

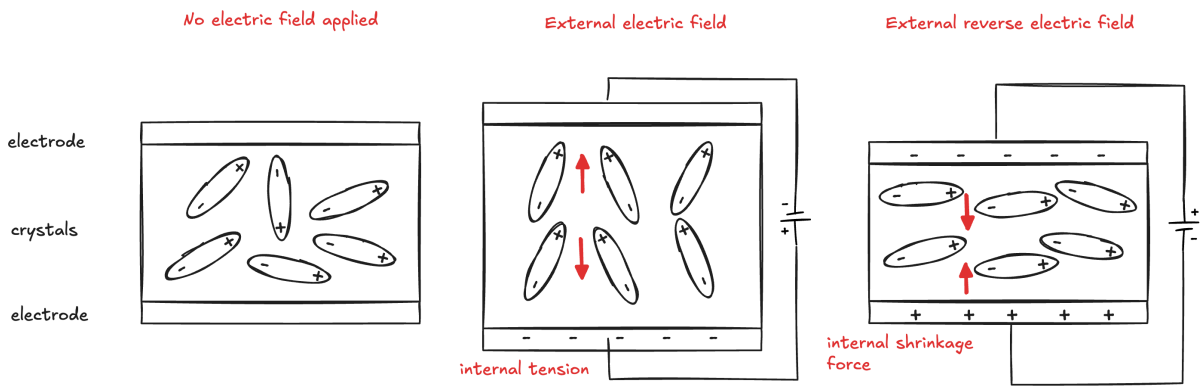
```
26      distance = (duration*.0343)/2;
27
28      // Print the distance to the serial monitor
29      Serial.print("Distance: ");
30      Serial.println(distance);
31
32      // Delay before the next measurement
33      delay(100);
34  }
```

1.3.4 Detecting vibration

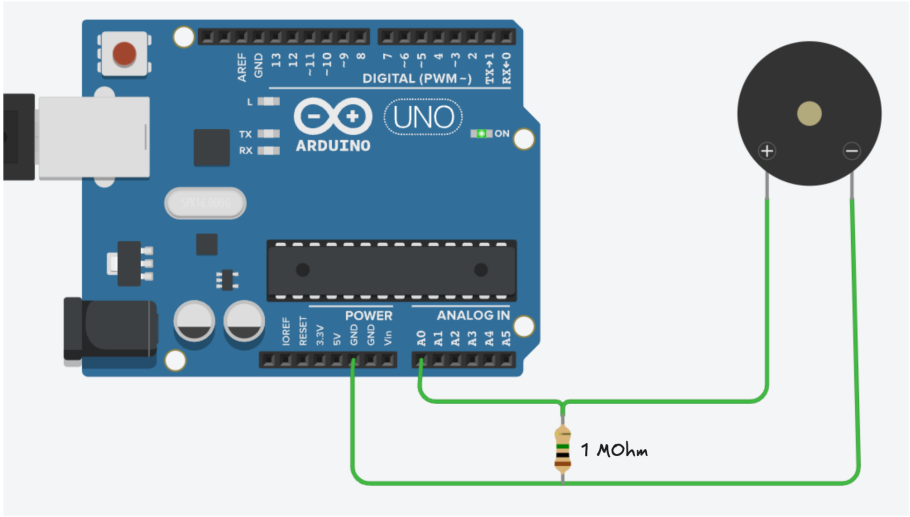
The **piezoelectric effect** is a phenomenon in which certain materials, such as quartz, ceramics, or specific polymers, generate an electric charge when subjected to mechanical stress. When an external force is applied in a specific direction, it causes a redistribution of electrical charges within the material, creating electric polarization. This results in an accumulation of charge on the material surfaces. Once the external force is removed, the material returns to its uncharged state. The polarity of the generated charge is aligned with the direction of the applied force, and the amount of charge produced is directly proportional to the magnitude of the applied stress.



The inverse piezoelectric effect refers to the phenomenon in which the application of an alternating electric field to a crystal induces mechanical deformation and vibration, also known as the electrostrictive effect. When the frequency of the applied alternating voltage matches the crystal natural frequency (which typically depends on the crystal's size), the amplitude of the mechanical vibrations increases significantly. This phenomenon is referred to as piezoelectric resonance:



This effect can be used in various types of sensors, including pressure sensors, accelerometers, and microphones. They can detect mechanical changes (e.g., pressure, vibrations) and convert them into an electrical signal. As an example, on the Makers kit we have a PKM22EPP-40, a simple buzzer (a piezoelectric device that generates sound by vibrating in response to an electrical signal). We can use it to detect vibrations by reversing the process: instead of applying an electrical signal to generate vibrations, we can detect vibrations by reading the electrical signal generated by the buzzer:



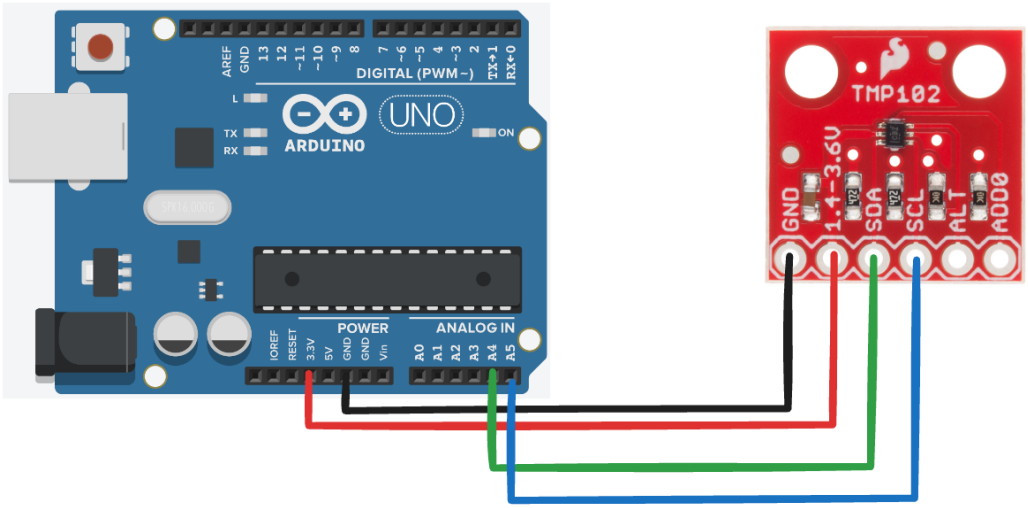
The buzzer generates an electrical signal when it is subjected to mechanical vibrations. However, when the buzzer is not vibrating, the analog pin might be left floating (no connection to either power or ground) and this can lead to unpredictable readings due to noise or interference. The resistor connects the analog pin to ground to ensure that when the buzzer is not generating a signal, the analog pin reads a low voltage. The code to read the output from the buzzer and detect vibrations when the signal exceeds a certain threshold (see *06.DetectingVibration* sketch):

```
1  // The analog pin connected to the sensor
2  const int sensorPin = 0;
3
4  // Variable to count the number of vibrations detected
```

```
int count = 0;

// Threshold value for vibration detection
const int THRESHOLD = 10;

void setup() {
    Serial.begin(9600);
}

void loop() {

    // Read the sensor value
    int val = analogRead(sensorPin);

    // If the sensor value exceeds the threshold, increment the count
    if (val >= THRESHOLD){
        count++;

        // Print the number of vibrations detected
        Serial.print("Vibration detected: ");
        Serial.println(count);

        // Delay to prevent multiple detections
        delay(200);
    }
}
```

1.3.5 Measuring temperature

A **thermal sensor** operates based on the principle that the properties of certain materials change with temperature. Specifically, the bandgap of a semiconductor varies with temperature, and this variation can be harnessed to produce a corresponding voltage or resistance signal. In the Makers kit we have the low-power digital TMP102 sensor. It includes a temperature sensor diode as the primary sensing element, which detects temperature changes. The analog signal from the diode is converted into a digital signal by a 12-bit analog-to-digital converter, providing a resolution of 0.0625°C and an accuracy of ±0.5°C, operating within a temperature range of -25°C to +85°C. Control logic manages the sensor's operation, including power modes and communication protocols. The sensor supports I2C, SMBus, and two-wire interfaces for easy integration with microcontrollers and other digital systems (see *07.MeasuringTemperature* sketch):

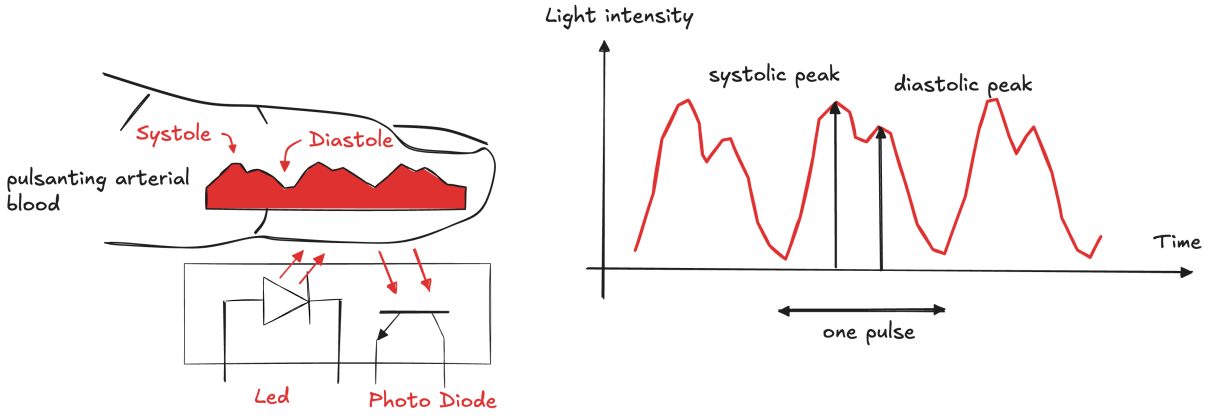



```c
#include <Wire.h>

// Define the I2C address of the TMP102 senso
int tmp102Address = 0x48;


// A faunction to read the temperature from the TMP102 sensor
float getTemperature() {

    // Request 2 bytes of data
    Wire.requestFrom(tmp102Address, 2);

    // Read the most significant and the least significant bytes
    byte MSB = Wire.read();
    byte LSB = Wire.read();

    // Combine the two bytes into a 12-bit integer
    // The temperature is a 12-bit value, using two's complement
    // for negative values
    int TemperatureSum = ((MSB << 8) | LSB) >> 4;
    float celsius = TemperatureSum * 0.0625;

    // Return the temperature in Celsius
    return celsius;
}

void setup() {
    Serial.begin(9600);
    Wire.begin();
}

void loop() {

    // Get the temperature in Celsius
    float celsius = getTemperature();

    // Print the temperature to the serial monitor
    Serial.print("Celsius: ");
    Serial.println(celsius);

    // Delay to slow down the output
    delay(200);
```

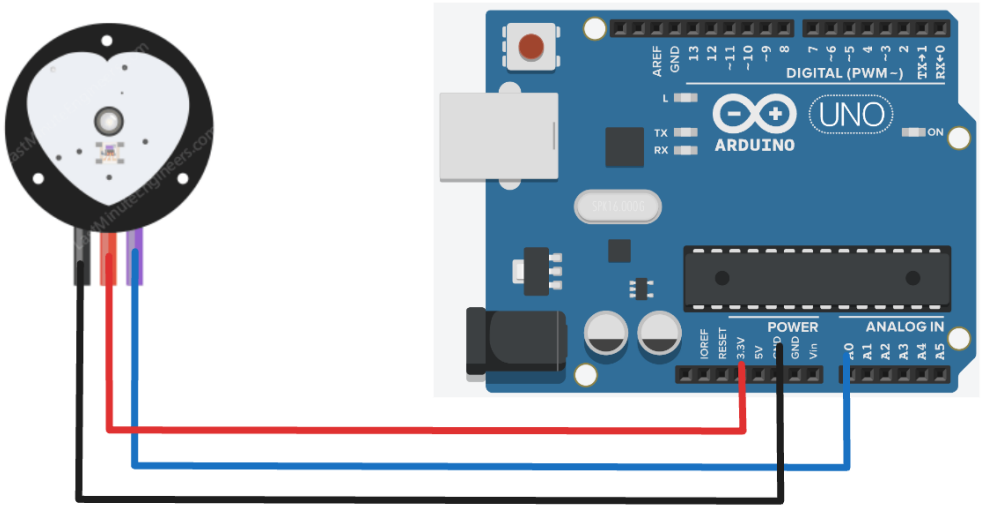
```
43  }
```

1.3.6 Measuring physiological signals

A **photoplethysmograph** (PPG) is a non-invasive optical sensor designed to detect changes in blood volume within the microvascular tissue. It works by emitting light from a source, typically a red or infrared led, onto the skin. As blood absorbs light differently depending on its volume and oxygenation level, the light that is either reflected or transmitted through the tissue varies with each heartbeat. This variation is captured by a photodetector, such as a photodiode, positioned to measure the changes in light intensity:



The signal obtained consists of two main components: a pulsatile signal, which corresponds to the rhythmic blood flow caused by the heartbeats, and a baseline component representing the steady-state absorption of the surrounding tissue. By processing this signal through amplification and filtering, key physiological parameters such as heart rate, oxygen saturation, and blood pressure trends can be derived. Commonly placed on areas like the fingertip, wrist, or earlobe where blood perfusion is strong, the PPG sensor combines simplicity, efficiency, and reliability, making it a widely used tool in health monitoring and wearable technology. In the Makers kit we have a PulseSensor Kit, a PPG sensor designed to be used wiht Arduino. We can use it to demonstrate a live visualization of human heartbeat in Arduino Serial Plotter.(see *08.PulseSensor* sketch):




```arduino
// Pin Definitions
const int PulseSensorPin = A0;
const int LEDPin = LED_BUILTIN;

// Variables to store the sensor data and threshold
// to detect a heartbeat (adjust based on your setup)
int Signal = 0;
const int Threshold = 580;

void setup() {
  pinMode(LEDPin, OUTPUT);
  Serial.begin(115200);
}

void loop() {
  // Read the raw signal from the Pulse Sensor
  Signal = analogRead(PulseSensorPin);

  // Print the signal value to the Serial Plotter for visualization
  Serial.println("Signal: " + String(Signal));

  // Check if the signal is above the threshold
  if (Signal > Threshold) {
    digitalWrite(LEDPin, HIGH); // Turn on the LED if a heartbeat is detected
  } else {
    digitalWrite(LEDPin, LOW);  // Turn off the LED otherwise
  }

  // Add a small delay to stabilize readings (20ms = 50Hz sampling rate)
  delay(20);
}
```

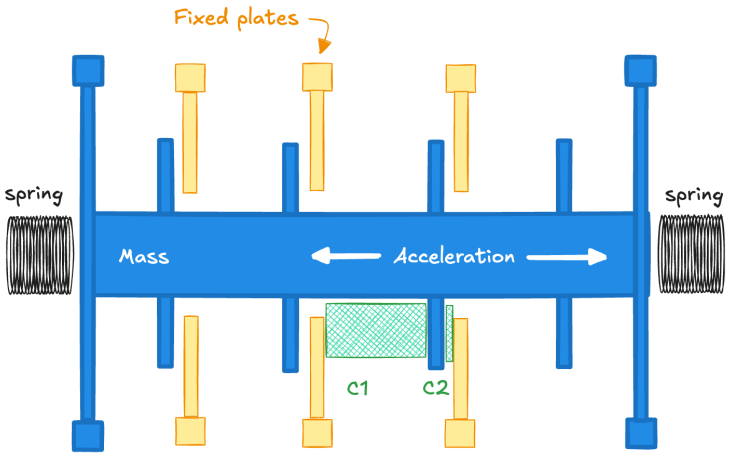
1.4 Self awareness

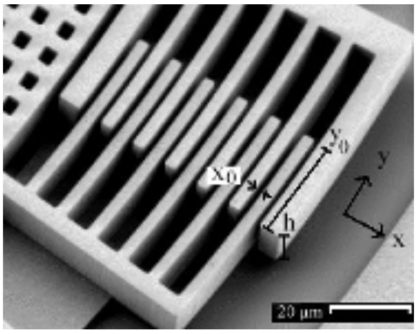
The concept of **self-awareness** is fundamental in embedded systems, enabling them to understand their state, monitor performance, and adapt to changing conditions. For instance, a device's **orientation** is critical in applications like navigation, gaming, and virtual reality. By determining its spatial

orientation, the device can deliver precise information to users, adjust its behavior dynamically, or seamlessly interact with its environment. In this section we concentrate on how we can supporto orientation using differnt source of information and how we can combine them to improve the accuracy of the estimation using **sensor fusion** techniques.

1.4.1 Accelerometers

An **accelerometer** measures the total acceleration acting on it, which includes both *dynamic forces from movemen*t and the **static acceleration caused by gravity**. This allows the device to detect motion as well as orientation relative to the Earth's gravitational field. Internally, an accelerometer consists of tiny capacitive plates. Some of these plates are fixed, while others are attached to microscopic springs, enabling them to move in response to external forces such as gravity, vibrations, or motion. When acceleration occurs, the movable plates shift their position relative to the fixed plates. This movement causes a change in the capacitance, which is the ability of the plates to store electrical charge. By measuring these changes in capacitance, the device can determine the magnitude and direction of the acceleration acting on it.

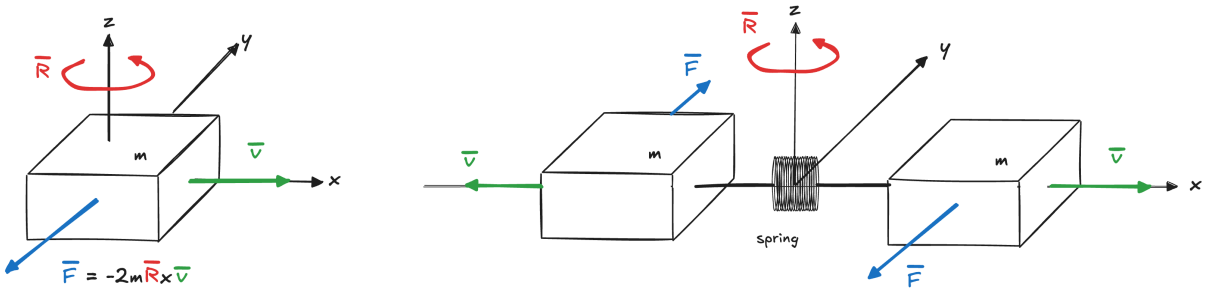




The measured acceleration is typically expressed in meters per second squared (m/s^2) or in G-forces, where one G-force is equivalent to approximately 9.81 m/s^2, the acceleration due to Earth's gravity. Notice the small size of the device, where mechanical elements such as tiny springs, capacitive plates, and moving masses are integrated with electronic circuitry on a single chip, typically fabricated using techniques similar to semiconductor manufacturing. These systems operate at the microscale and this type of device is called **MicroElectroMechanical Systems (MEMS)**. This technology is widely used in applications such as detecting orientation in smartphones, monitoring vibrations in machinery, and enabling motion-sensitive systems in automotive safety features like airbags.

1.4.2 Gyroscopes

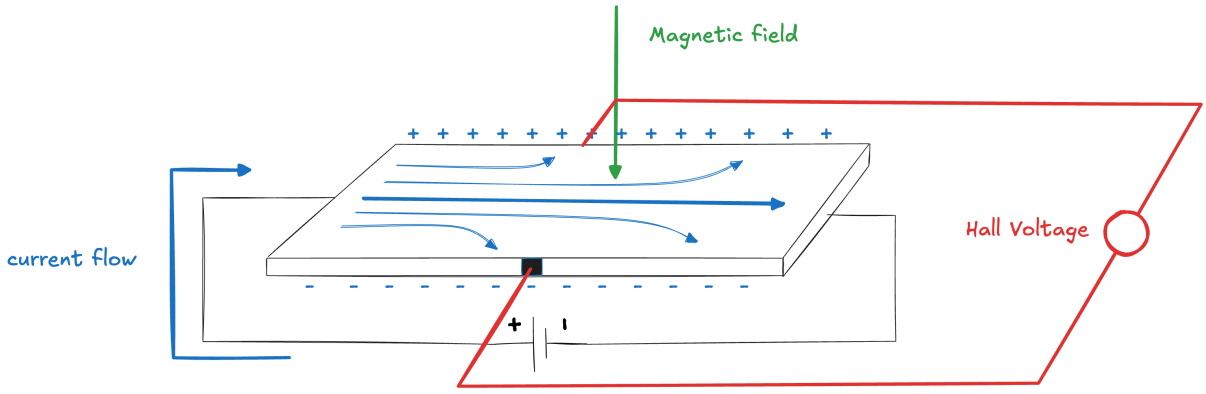
A **gyroscope** functions by exploiting the **Coriolis effect**, which causes a moving mass to experience a force that deviates its path when subjected to rotational motion. Inside a gyroscope, there is typically a suspended vibrating structure that responds to changes in angular velocity. When the device rotates, the vibrating element is affected by the Coriolis force, resulting in a shift perpendicular to its original vibration.



This displacement is minute, but it can be detected with high precision, often through capacitive or piezoelectric sensors. As the vibrating structure moves, the distance between it and fixed electrodes changes, which in turn alters the capacitance or generates a voltage. This variation is then measured and used to determine the angular velocity of the rotation.

1.4.3 Magnetometers

A **magnetometer** measures the strength and direction of a magnetic field using the **Hall Effect**.When a current flows through a thin, conductive plate, the charge carriers move in a straight line from one side of the plate to the other. However, when a magnetic field is applied perpendicular to the direction of the current, it causes the charge carriers to experience a force, known as the **Lorentz force**. This force causes the electrons to be deflected to one side of the plate, while the holes are pushed to the opposite side. As a result, a voltage develops across the width of the plate, perpendicular to both the current and the magnetic field. This voltage is known as the Hall voltage and is proportional to the strength of the magnetic field, allowing it to be used for magnetic field measurements:



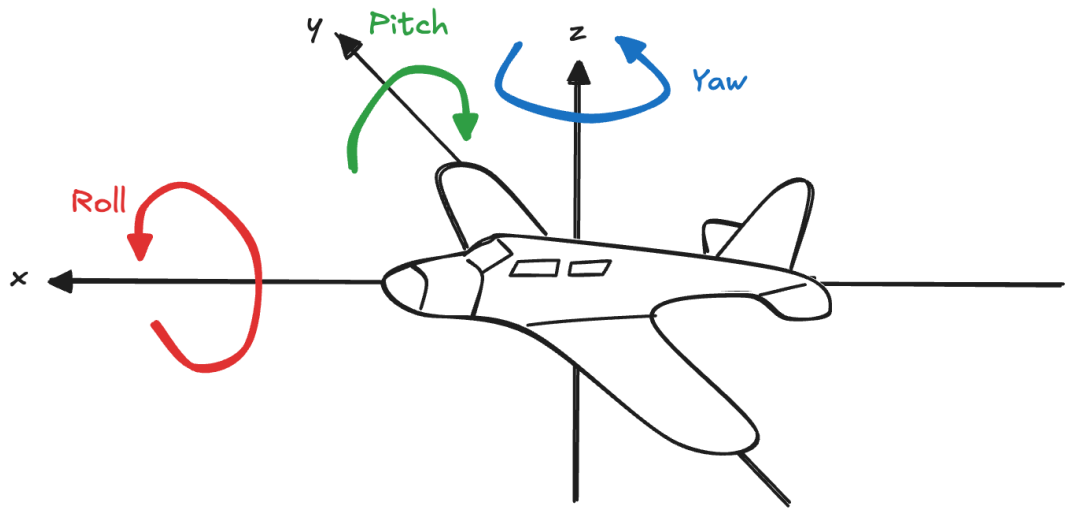
This device can be used to detect the Earth's magnetic field, which is relatively constant in direction and strength, providing a reference for orientation.

1.4.4 Orientation

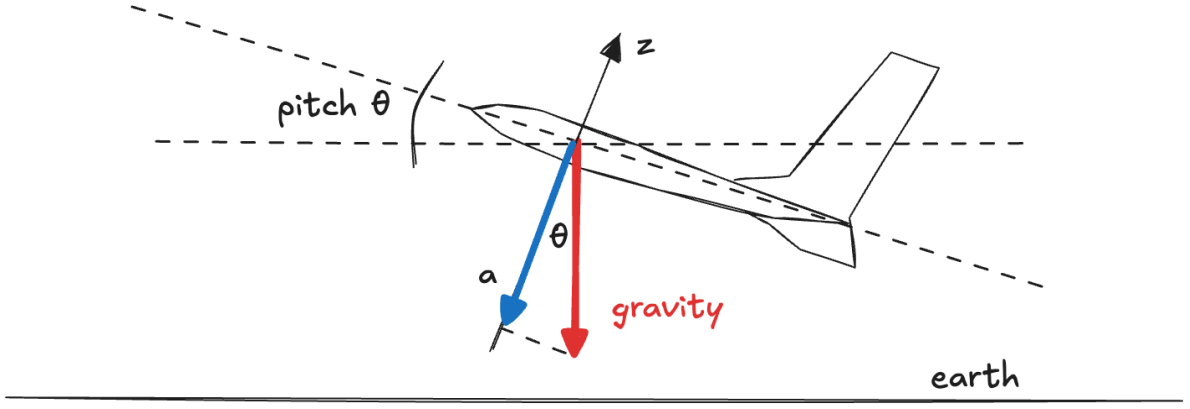
Orientation describes the position of an object in three-dimensional space relative to a reference frame, typically defined using three angles:

- **Yaw**: The rotation around the **vertical (z-axis)**. It describes how the object turns left or right, similar to a compass direction. For example, a car turning on a flat road is primarily changing its yaw.
- **Pitch**: The rotation around the **lateral (y-axis)**. It describes the up or down tilt of the object, like nodding your head or a plane adjusting its nose up or down during flight.
- **Roll**: The rotation around the **longitudinal (x-axis)**. It describes the tilting motion to the left or right, such as when an airplane banks to turn or a boat rocks sideways.

Together, these angles specify the orientation in 3D space. These angles are often used in applications like navigation, robotics, aerospace, and computer graphics to represent and control the position of an object.



As described, an accelerometer measures acceleration along predefined axes, and this includes both dynamic accelerations (from motion) and the Earth's gravity, which is a constant acceleration of approximately 9.8 m/s^2. By leveraging this measurement and in static condition (no motion), we can calculate the orientation of the body relative to the gravitational field. For example, considering only the pitch, with a body perfectly aligned with the horizontal plane, the gravity force is a constant downward force. When the body is tilted, the component of gravity along the z-axis changes according to the angle of tilt:



$$a = -g \cdot \sin(\theta)$$

By measuring the acceleration along the x-axis, we can calculate the pitch angle using the inverse sine function:

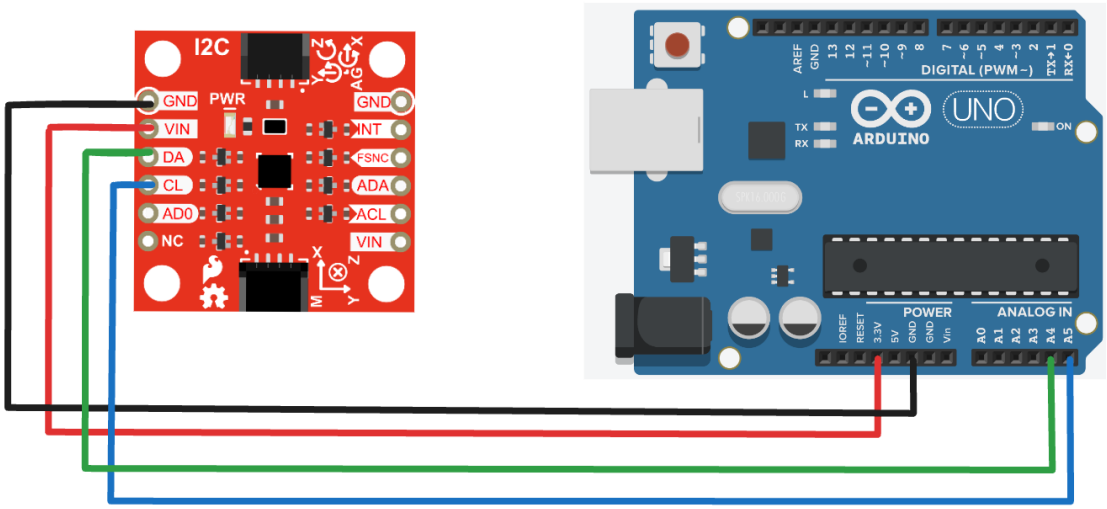
$$\theta = \arcsin\left(\frac{-a}{g}\right)$$

More generally, if the body is not perfectly aligned with the horizontal plane and considering the roll and yaw angles, the orientation can be calculated using the following equations:

$$pitch = \arctan\left(\frac{a_x}{\sqrt{a_y^2 + a_z^2}}\right)$$

$$roll = \arctan\left(\frac{a_y}{\sqrt{a_x^2 + a_z^2}}\right)$$

In the Makers Kit, we have a ICM-20948 sensor featuring a 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis magnetometer, all integrated into a single chip. The ICM-20948 sensor is a **9DOF** (degrees of freedom) IMU (Inertial Measurement Unit). We can use it to calculate the two angles (see *09.Pitch_Accelerometer* sketch):



We can conduct an experiment to analyze the behavior of the accelerometer when rotated around its y-axis. We held the IMU in one hand and gently rotated first in one direction and then in the opposite direction. The motion is performed carefully, aiming to maintain a smooth and consistent speed throughout the rotation. During the experiment, the pitch angle is recorded by reading the sensor data and sending it to the serial monitor (see *09.Pitch_Accelerometer*):

```
// Include the sensor library and the I2C library
#include <Wire.h>
#include "ICM_20948.h"

// Create an ICM_20948 object
ICM_20948_I2C myICM;

void setup() {
  Serial.begin(115200);
  Wire.begin();

  // Initialize the ICM-20948
  if (myICM.begin(Wire, 0x68) != ICM_20948_Stat_Ok) {
    Serial.println("IMU initialization failed!");
    while (1);
  }
  Serial.println("IMU initialized successfully!");
```

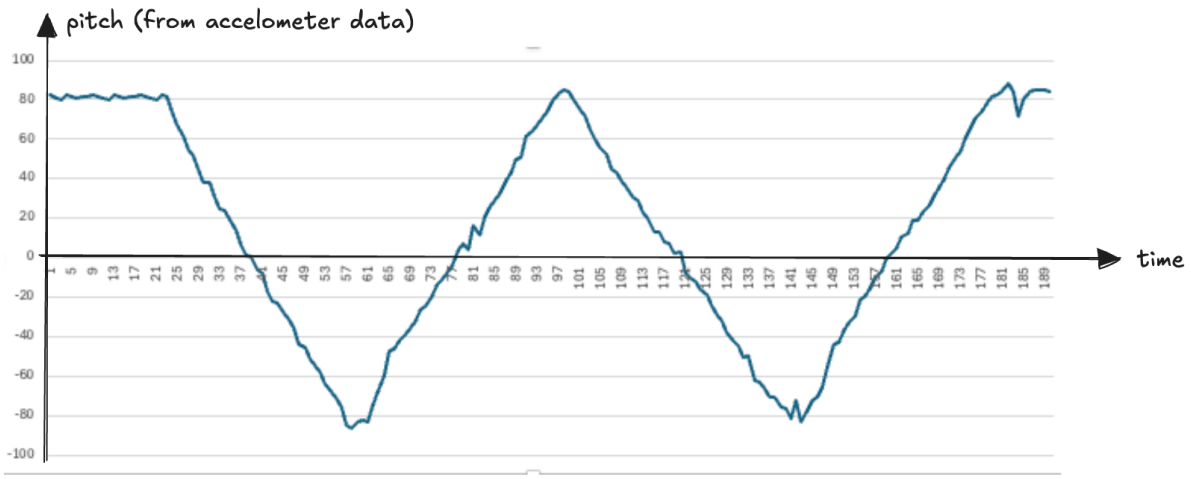
```
}

void loop() {

  // If sensor data ready are available
  if (myICM.dataReady()) {

    // Read accelerometer, gyroscope, and magnetometer data
    myICM.getAGMT();

    // Extract accelerometer data
    float ax = myICM.accX();
    float ay = myICM.accY();
    float az = myICM.accZ();

    // Calculate pitch angle (in degrees)
    float pitch = atan2(-ax, sqrt(ay * ay + az * az)) * 180.0 / PI;

    // Send pitch angle to Serial Monitor
    Serial.print("Pitch: ");
    Serial.println(pitch);
  }

  // Small delay to make output readable
  delay(200);
}
```

The data are collected by a Processing sketch that reads the serial data, appends them to a text file while visualizing the pitch in a 3D box on the screen.

```
import processing.serial.*;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.text.DecimalFormat;
import java.text.DecimalFormatSymbols;
import java.util.Locale;

// Output filename for storing pitch values
String outFilename = "out.csv";

// Serial object for communication
Serial myPort;

// Variable to store the pitch value
float pitch = 0;

void setup() {
    // Set up the canvas size and 3D renderer
    size(640, 360, P3D);

    // Initialize serial communication with the port
    myPort = new Serial(this, Serial.list()[2], 115200);
    myPort.bufferUntil('\n');
}

void draw() {
    // Clear the screen with a black background
    background(0);

    // Add lighting to the scene
    lights();

```

```
    // Apply 3D transformations and rotate the box based on pitch
    pushMatrix();
    translate(width / 2, height / 2, -30);  // Move the box to the center of the canvas
    rotateX(radians(pitch));  // Rotate the box on the X axis using the pitch value
    box(100);  // Draw the box with a size of 100 pixels
    popMatrix();  // Restore the previous transformation state
}

void serialEvent(Serial myPort) {
    // Read the incoming data until a newline character
    String rpstr = myPort.readStringUntil('\n');

    // Check if the data is not null
    if (rpstr != null) {
        // Split the data into an array using ':' as delimiter
        String[] list = split(rpstr, ": ");

        try {
            // Parse the pitch value from the received data
            pitch = float(list[1]);

            // Append the formatted output to the file
            appendTextToFile(outFilename, pitch);
        }
        // Catch and ignore any exceptions in case of malformed data
        catch (Exception ex) {}
    }
}

void appendTextToFile(String filename, float pitch) {
    // Set up the locale for countries that use ',' as a decimal separator
    DecimalFormatSymbols symbols = new DecimalFormatSymbols(Locale.ITALY);
    DecimalFormat df = new DecimalFormat("#0.00", symbols);

    // Try to open the file and write the text to it
    try (BufferedWriter out = new BufferedWriter(new FileWriter(dataPath(filename), true))
        ) {
        out.write(df.format(pitch));  // Write the pitch to the file
        out.newLine();     // Add a new line after each entry
    }
    catch (IOException e) {
        e.printStackTrace();  // Print any errors that occur while writing the file
    }
}
```

Finally, we can import in a spreadsheet the data collected and plot the pitch angle as a function of time. The plot shows the pitch angle increasing as the sensor is rotated in one direction and decreasing as it is rotated in the opposite direction. This demonstrate how the IMU detects and reports angular changes when subjected to controlled rotational motion:



The accelerometer is a sensor designed to measure all forces acting on an object, not just gravitational acceleration. This means it detects any external forces, such as vibrations, impacts, or motion-induced forces. As a result, its measurement is **highly sensitive and easily disrupted by even minor disturbances**. For instance, small forces like friction and vibrations from the environment can significantly alter the readings, making it challenging to isolate the specific force of interest (such as gravity or intentional motion).

However, we also have a gyroscope, which measures the **angular velocity** and we can also estimate angular position using gyroscope data. To achieve this, we must compute the **integral of angular velocity over time**. However, continuous integration is not possible in a digital system. Instead, we approximate it by summing a finite number of discrete samples taken at regular intervals:

$$\omega_y(t) = \frac{d}{dt} pitch$$

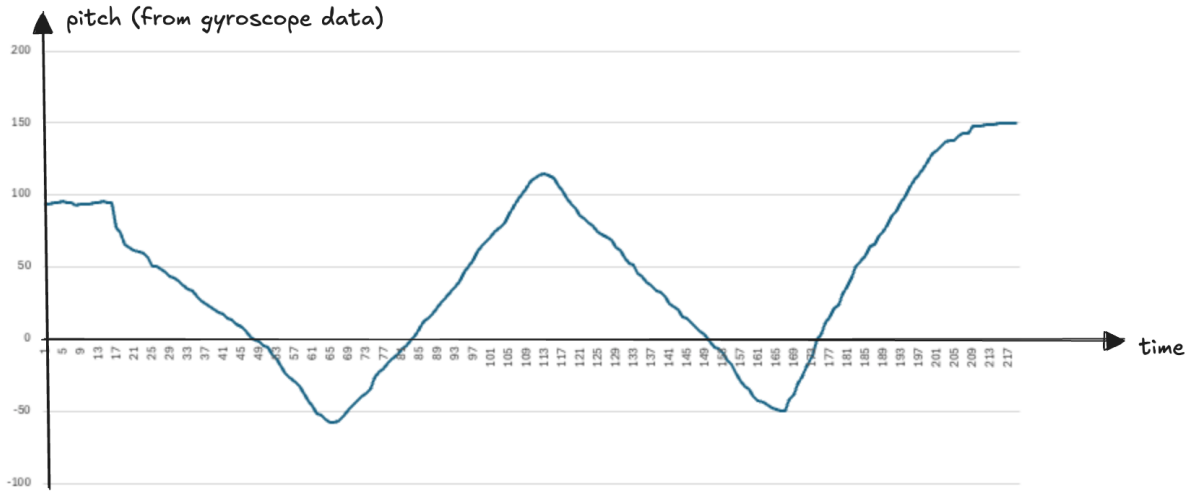
$\displaystyle pitch = \int_0^t \omega_y(t) , d t \approx \sum_{0}^t \omega_y(t) \cdot T_s $

We can repeat the experiment using the angular position estimated from gyroscope data (see *10.Pitch_Gyroscope*):

```
// Include the sensor library and the I2C library
#include <Wire.h>
#include "ICM_20948.h"

// Create an ICM_20948 object
ICM_20948_I2C myICM;

// Variable to store the timer for pitch calculation
// usgin integration of gyroscope data
unsigned long timer;

// Variable to store the pitch angle
// initialized to 90 degrees (the first position)
float pitch = 90;

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

```
  Wire.begin();

  // Initialize the ICM-20948
  if (myICM.begin(Wire, 0x68) != ICM_20948_Stat_Ok) {
    Serial.println("IMU initialization failed!");
    while (1);
  }
  Serial.println("IMU initialized successfully!");
}

void loop() {

  // If sensor data ready are available
  if (myICM.dataReady()) {

    // Read accelerometer, gyroscope, and magnetometer data
    myICM.getAGMT();

    // Extract gyroscope data
    float vx = myICM.gyrX();
    float vy = myICM.gyrY();
    float vz = myICM.gyrZ();

    // Calculate the pitch angle based on gyroscope Y-axis data
    // This is an integration of the velocity over time to get the angle
    float dt = (double)(millis() - timer) / 1000.0;
    pitch += vy * dt;

    // Update the timer for the next calculation
    timer = millis();

    // Send pitch angle to Serial Monitor
    Serial.print("Pitch: ");
    Serial.println(pitch);
  }

  // Small delay to make output readable
  delay(200);
}
```

We can repeat the experiment using the angular position estimated from gyroscope data:



The gyroscope data are much more stable and less sensitive to external disturbances compared to the accelerometer. However, if the angular velocity changes more rapidly than the sampling frequency allows, some variations will go undetected, resulting in an inaccurate integral approximation. Additionally, small inaccuracies in each measurement **accumulate over time**, gradually deviating from the true angular position. This phenomenon, known as **drift**, becomes more pronounced the longer the integration continues.

1.4.5 Sensor fusion

Using sensor fusion, accelerometers and gyroscopes complement each other to provide more reliable and accurate data. As mentioned, the accelerometer is sensitive to all forces, including external disturbances like vibrations, which can affect the measurement of orientation. Meanwhile, the gyroscope provides accurate rotational data but tends to drift over time due to the integration process. By **combining** both sensors, we can leverage the gyroscope short-term accuracy and the accelerometer ability to correct drift over time. This approach results in a more stable and precise estimation of orientation, addressing the individual limitations of each sensor. This can be done using a technique called **complementary filtering**, that combines the two angles, using a weight to control the blend between the accelerometer and gyroscope data. The equation for updating the pitch estimate is:

$$pitch(t) = \alpha \cdot (pitch(t-1) + \omega(t) \cdot T_s) + (1-\alpha) \cdot \arctan\left(\frac{a_x}{\sqrt{a_y^2 + a_z^2}}\right)$$

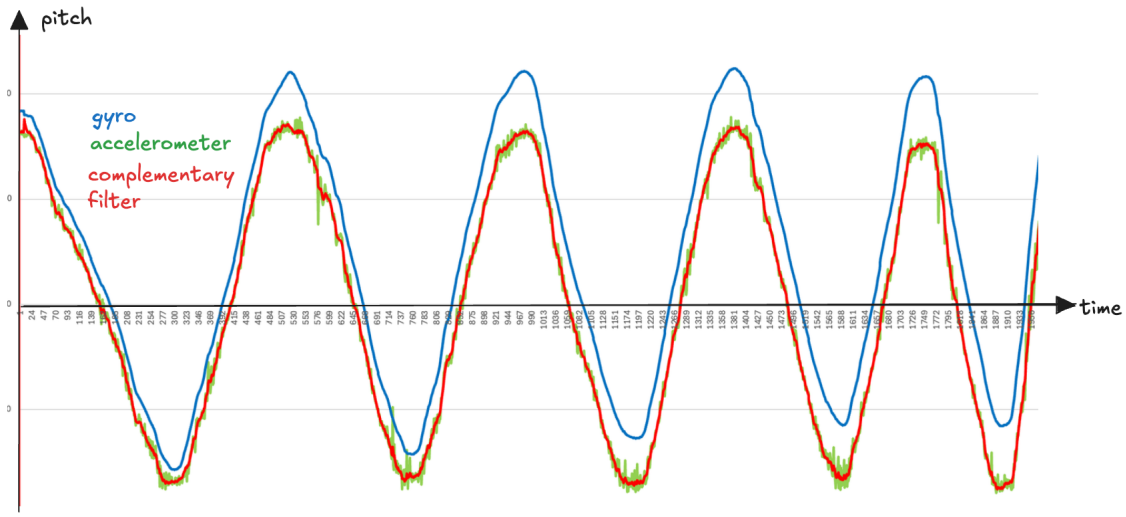
The first term represents the predicted angle from the gyroscope. It provides a fast response but can drift over time due to sensor noise. The second term represents the angle based on the accelerometer. It is stable over time but noisy in the short term (e.g., when there is a lot of movement or vibration). The weight determines the balance between the two sources, ant its value is usually set empirically, depending on the application. If the system is highly dynamic, a larger value of (e.g., 0.98 or 0.99) will give more weight to the gyroscope, making the system respond quickly to changes. If the system is relatively stable, we can decrease to give more weight to the accelerometer data for more accuracy (see *11.Pitch_SensorFusion*):

```
// Include the sensor library and the I2C library
#include <Wire.h>
#include "ICM_20948.h"

// Create an ICM_20948 object
ICM_20948_I2C myICM;

// Complementary filter parameter
const float alpha = 0.8;

// Estimated pitch angle (initialized to 90)
float pitch_acc = 90;
float pitch_gyro = 90;
float pitch = 90;
```

```

// Timer variables for delta time calculation
unsigned long timer = 0;

void setup() {
  Serial.begin(115200);
  Wire.begin();

  // Initialize the ICM-20948
  if (myICM.begin(Wire, 0x68) != ICM_20948_Stat_Ok) {
    Serial.println("IMU initialization failed!");
    while (1);
  }
  Serial.println("IMU initialized successfully!");

  // Initialize time tracking
  timer = millis();
}

void loop() {
  // If sensor data is available
  if (myICM.dataReady()) {

    // Read accelerometer, gyroscope, and magnetometer data
    myICM.getAGMT();

    // Extract accelerometer data
    float ax = myICM.accX();
    float ay = myICM.accY();
    float az = myICM.accZ();

    // Compute pitch angle from the accelerometer
    pitch_acc = atan2(-ax, sqrt(ay * ay + az * az)) * 180.0 / PI;

    // Extract gyroscope data
    float vy = myICM.gyrY();

    // Compute time difference Δ(t) in seconds
    float dt = (double)(millis() - timer) / 1000.0;

    // Compute pitch angle from the gyroscope
    pitch_gyro += vy * dt;

    // Compute pitch using the complementary filter
    pitch = alpha * (pitch + vy * dt) + (1 - alpha) * pitch_acc;

    // Update previous time
    timer = millis();

    // Print pitch angle to Serial Monitor
    Serial.print("Pitch: ");
    Serial.print(pitch_acc);
    Serial.print(":");
    Serial.print(pitch_gyro);
    Serial.print(":");
    Serial.println(pitch);
  }

  // Small delay to make output readable
  delay(10);  // Reduced delay for smoother angle tracking
}
```

We can repeat the experiment using the sensor fusion approach to estimate the pitch angle and com-

pare it with the accelerometer and gyroscope data:



The graph illustrates how the sensor fusion approach provides a more stable and accurate estimation of the pitch angle compared to using either the accelerometer or gyroscope alone. The accelerometer-based estimate is noisy, while the gyroscope-based estimate tends to drift over time. The complementary filter effectively combines the strengths of both sensors, leveraging the accelerometer's long-term stability and the gyroscope's responsiveness to produce a more reliable orientation estimate. This technique is widely used in applications such as drones, robotics, and virtual reality systems, where precise orientation tracking is essential. A more advanced method for sensor fusion is the **Kalman filter**, an algorithm that estimates the state of a system based on a series of noisy measurements. It recursively updates the state estimate by integrating new measurements with previous predictions, allowing it to handle multiple sensor inputs, compensate for noise and uncertainty, and adapt to changing conditions. However, a detailed discussion of the Kalman filter is beyond the scope of this course.

1.5 Hands-on Activity

1 - Adjust the brightness of an LED based on ambient light levels using Arduino **Pulse Width Modulation** (PWM). PWM controls brightness by rapidly switching the LED on and off, varying the proportion of time it stays on (duty cycle). This creates the effect of smooth dimming, as the human eye perceives the average light intensity rather than the rapid flickering. By adjusting the duty cycle in response to ambient light readings, the LED can automatically brighten in the dark and dim in well-lit conditions.

2 - Utilize the pitch and roll data to dynamically rotate a cube in 3D space along the corresponding axes. By continuously updating these values based on sensor readings, the cube will realistically respond

to changes in orientation, simulating real-world motion.

3 - Display a real-time chart of heart rate data collected from the heart-rate sensor. Continuously update the graph to visualize heart rate variations over time, allowing for easy monitoring of trends, anomalies, or sudden changes in pulse.