RBE 2002: Unified Robotics II
Lab 4 Background

Last modification: November 12, 2022

1 Introduction

An underlying theme of this course is that sensors are imperfect. Whether it's due to noise, non-linearity, drift, or many other phenomena, no sensor is ever going to register the exact "truth" of the property that it is measuring. Even something as simple as reading a button is fraught with pitfalls, thanks to bouncing.

In this lab, you'll explore the basics of sensor fusion: fundamentally, the notion that two different sensors are telling you different things about the *same* property. Redundancy is good, but figuring out how to deal with the conflicting information is non-trivial.

For this lab, we'll concentrate on the IMU that is built-in to your Romi. The IMU has an accelerometer and a gyroscope (but no magnetometer). The chip communicates with the microcontroller using the I2C protocol, a two-wire, synchronized, serial interface, and you'll hear more about how I2C works in lecture and explore other interfaces.

1.1 Objectives

The successful student will be able to:

- Interface an IMU with a microcontroller,
- Create a complementary filter and explain how the filter performs better than the individual sensors that are used in the filter, and
- Use the filter results to inform their robot of its state (namely, detect a ramp).

2 Sensor interfaces

In previous labs, you explored the difference between analog and digital interfaces. In particular, you read analog voltages from a sensor with an analog-to-digital converter. You also explored the nature of simple digital signals, for example the ultrasonic echo. Many sensors, however, have a lot more complexity than the simple designs seen so far. A GPS receives high-frequency radio signals from several satellites and does complex computations to determine its location. A number of temperature sensors perform filtering and analog-to-digital conversion on-board. Inertial measurement units use MEMS sensors to determine acceleration, rates of rotation, or measure a magnetic field. The wealth of sensors is practically unbounded, and communication with such devices requires more complex protocols than can be provided by a simple digital or analog interface. Not

only is it necessary to transmit the results of measurements, but the sensors often have a number of configuration parameters that can be changed to suit the user's needs.

In such cases, digital interfaces are still used, but instead of just a single `HIGH` or `LOW` as you've seen with, say, a button, the microcontroller and peripheral communicate using strings of `1`'s and `0`'s. These `1`'s and `0`'s can be sent using one of many *protocols*, each of which defines how to package, transmit, and interpret the data as it's sent between devices. Typically, data is packaged into bytes, and data transfer can be unidirectional, where only one device sends data, or bidirectional, where both devices send information.

The vast majority of sensors (all that we'll encounter in this class) send data using serial interfaces, meaning that the data is sent bit-by-bit, one after the other. Conversely, with parallel interfaces multiple bits are sent simultaneously over multiple wires. Parallel interfaces are common with very high-speed devices, such as disk drives, memory cards, or cameras.

The foremost distinction between different protocols is whether they are *synchronous* or *asynchronous*:

- With **synchronous** communication, a clock is shared between both the microcontroller and the peripheral. A separate wire is used to transmit pulses, usually, but not always, from the microcontroller to the peripheral. By monitoring to clock pulses, the peripheral knows when to send each bit of information or how to interpret each bit that it receives. The rate of communication is determined by the device that controls the clock, but because there is a single clock, the exact speed doesn't matter so long as it is within the computational ability of each device. Common synchronous interfaces include:

  - **SPI**. The *Serial Peripheral Interface* is a very common protocol for communicating with sensors and other microcontrollers. Typically, each peripheral device requires four connections, plus a ground. Three lines belong to the SPI *bus*: a line for the clock, and two lines for data (one in each direction). The fourth line is for the *chip select*, which tells the peripheral when it is active. This allows multiple devices to share the same SPI bus, so long as they each have their own chip select connection. The SD card interface that you've been using is an SPI interface.
  - **I2C**. The *Inter-Integrated Circuit* protocol is a two wire (plus ground) interface. One line is the clock line and the other is the data line. Each peripheral has a unique, 7-bit address, which allows multiple devices to be on the same bus: each one listens for its address to be called, in which case it continues the communication with the microcontroller. The entire protocol can get complicated – it has very specific rules for calls and responses – but Arduino has a useful library for managing it all.

  SPI and I2C interfaces are so common that many microcontrollers contain special hardware to manage the low-level communication, which lessens the burden on the main processor.

- With **asynchronous** communication, each device uses its *own* clock to time the data pulses. The two devices must agree on a data transfer rate so that the bits are interpreted properly. As you can imagine, such communication requires both devices to have reasonably accurate clocks to be sure that no information is lost or garbled. Common asynchronous interfaces include:

- **UART**. The *Universal Asynchronous Receive Transmit* interface is an extremely common asynchronous protocol. So common, in fact, that it is often referred to (at least in Arduino-speak) as a "serial port", despite the fact that all of the protocols here are serial protocols. It is the protocol behind the communication between the Arduino and your computer (though there is a USB translator in between). This is why you get garbled data if the baud rate in the Serial Monitor is not the same as the setting in your Arduino code. A UART typically consists of two connections (plus ground), with one wire being for receiving data and one for transmitting. Data to be transmitted as bytes, with start, stop, and parity bits used to delineate each byte so that the receiver doesn't lose track of the bits. Because it is so common, most microcontrollers have special hardware for at least one UART.
- **RS-232/485**. The RS-232 and RS-485 interfaces are similar to the UART except that they are optimized for longer transmission distances. The bit patterns are generally the same as with a UART (start bits, stop bits, parity bits), but the voltages range from -5 V to 12 V to reduce the impact of noise on the communication. The end result is that the microcontroller logic is similar to that for the UART (in fact, one typically uses the UART hardware), but *level shifters* must be used to convert the voltages to ones the microcontroller can handle. You won't be using either of these protocols in this lab.
- **1-wire bus**. There are a number of 1-wire protocols, which are the asynchronous analogues to I2C. Communication consists of requests and responses, and a number of peripherals can be connected to the same bus, so long as they have different addresses.
- **IR and radio**. Though wireless, infrared and radio transmissions (including WiFi, Bluetooth, and other wireless protocols) are serial connections. Data is sent bit-by-bit using a high frequency *carrier wave* to encode the relatively low frequency (compared to the carrier wave) data. There are no physical connections, of course, so it is impractical to share a clock between two devices.

Again, many microcontrollers will have special hardware for implementing these protocols, especially the UART.

2.1 More on I2C

As noted, I2C is a two-wire interface where the controller and the peripherals all share one data line (`SDA`) and one clock line (`SCK`). To initiate communication, the controller first sends a `START` command and then the address of the desired peripheral on the data line. Every peripheral will hear the address, but only the peripheral with the correct address will respond, and the microcontroller and peripheral will "own" the bus until a `STOP` signal is sent, at which point a new session with another device can be started.

When initiating a session with a peripheral, the seven bit address of the device is augmented by an eighth bit, which is the `READ/WRITE` bit – it tells the peripheral if it should expect to send or receive data. If the microcontroller is writing to the peripheral, after sending the peripheral address, the microcontroller will typically send an address of a *register* that it is writing to, followed by the value to put in the designated register. To read from a peripheral, the microcontroller first *writes* the address of the register it wishes to read from and then restarts the communication in read mode, after which the peripheral will send the data contained in that register. Because the protocol is uses a request/response format, data reception is typically acknowledged with an `ACK` signal. More details and examples will be presented in lecture.

3 The IMU

Your Romi comes equipped with an *inertial measurement unit*, or IMU. As noted, the IMU consists of an accelerometer and a gyroscope. We ask you to explore some of the specifics in the lab exercises.

3.1 The accelerometer

The accelerometer measures not only "true" accelerations, but also the apparent acceleration due to gravity. This latter quality is useful because if we can measure gravity, we can figure out which direction is down (though we don't know the orientation w.r.t. rotations around the gravity vector). There are two problems with reading the gravity vector:

1. Accelerations of the robot will "mask" the gravity vector.
2. Vibrations of the robot will lead to noise in the sensor signal.

To understand how to use the accelerometer to determine which way is down, let's write the response of the accelerometer as a combination of the gravity vector and acceleration. To simplify things, we'll consider only rotations about the y-axis, which we will assume is horizontal. Because the robot is rotating, we have two coordinate systems, as shown in Figure 1:

1. A local coordinate system attached to the robot. We denote the local coordinates with lowercase subscripts and lowercase unit vectors for each axis, (e.g., $\hat{\mathbf{i}}$ and $\hat{\mathbf{k}}$).
2. A global coordinate system attached to the earth (often called the "lab frame"). We denote the global coordinates with uppercase subscripts and uppercase unit vectors (e.g., $\hat{\mathbf{I}}$ and $\hat{\mathbf{K}}$).

The orientation of the local system in the global is described by the angle, θ_y, which has a positive value for positive rotations about the θ_y axis.

To determine the accelerometer readings, which we'll designate by α_x, α_y, α_z, we must also express gravity in the local coordinates (since the accelerometer is attached to the robot, it's in the local coordinate system). Doing so results in,

$$\alpha_x = a_x - g \sin \theta_y \tag{1}$$
$$\alpha_z = a_z + g \cos \theta_y \tag{2}$$

where a is the physical acceleration and g is the apparent acceleration due to gravity and is a positive number in this derivation. More details of the derivation were presented in class.

When the acceleration of the robot is zero, then it's straightforward to use trigonometry to calculate θ_y from the accelerometer readings in the x- and z-axes.

Unfortunately, any acceleration (that is not parallel to gravity) will affect the ratio, which will lead to errors in the estimation of the true direction of gravity. Of course, one can try to remove the acceleration terms by estimating the acceleration of the system using other factors (for example, motor torque or change in velocity). In practice, however, it is still difficult to get a good estimate of the gravity vector.

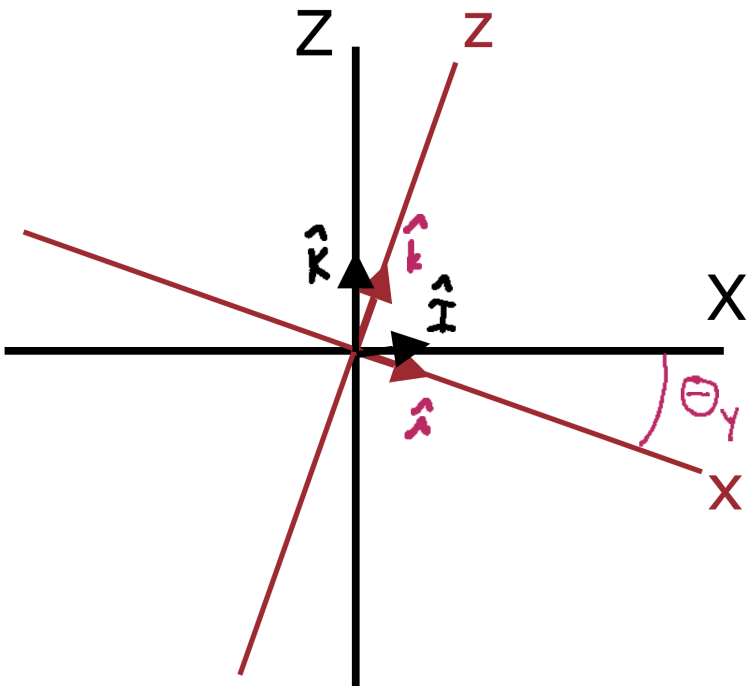

Figure 1: Coordinate system for a single rotation. Is the y-axis into or out of the page?

3.2 The gyroscope

The gyroscope measures angular velocity – the *rate* of spinning or turning. While gyroscopes are less susceptible to noise (especially from vibrations), gyroscopes are prone to offset errors, typically referred to as *bias*. If you set a gyroscope on a stable surface such that it is motionless, it will still register a small, but non-zero rotation, equivalent to slowly rolling the robot over. It is possible to reduce the bias through calibration, but the bias will change slowly over time due to a variety of factors, such as fluctuations in temperature or supply voltage. Without a correction, the errors will compound.

4 Sensor fusion

Ultimately, we want to determine the orientation of our robot. But while the accelerometer is good at measuring gravity over the long term, it is susceptible to noise. The gyroscope is less susceptible to noise, but prone to drift. In short, two different sensors are giving conflicting information about the orientation of our robot. What to do?

The solution is to implement *sensor fusion*, where information from two (or more) sources are blended to produce a "best estimate" of the true state of the system. In this instance, we'll build an algorithm that relies on the gyroscope for short-term motion tracking, but also uses the accelerometer to remove the gyroscope drift over the long-term.

4.1 Estimators

In the end, we will never be able to determine the exact value of a system variable – the best we can do is make *estimates* and use *system models* and *observations* to improve those estimates. Though we can't know the error between the estimate and the true value at any time,[1] we can construct our estimator to guarantee that the estimate will *approach* the true state over time. We can even draw statistical or probabilistic conclusions about the error to help us make decisions. We will avoid all the details here[2] and simply demonstrate that the complementary filter is capable of producing a better estimate of the system than either the accelerometer or gyroscope alone.

4.2 The complimentary filter

The complimentary filter is a basic fusion algorithm that uses a fixed weighting to combine information from two (or more) sensors, where the weight can be chosen experimentally or through theoretical considerations. For our problem here, we want a filter that reacts quickly to motion, but removes the noise that plagues the accelerometer.

The basic procedure is to make a *prediction* using kinematic updates from the gyroscope and *correct* the prediction based on observations from the accelerometer.

Let the gyroscope reading at time, k, be given by γ^k, which is a measurement of the true angular velocity, ω^k, plus some slowly varying bias, b,[3]

Using the sensitivity, s, of the gyroscope as defined in the datasheet, we can write,

$$\gamma_y = \sigma \frac{d\theta_y}{dt} + b_\gamma \tag{3}$$

where b in this case has the same units as ω (typically radians/second, but you can also use degrees, so long as you're consistent).

We can discretize the equation and rearrange it to solve for an estimate of θ. This will be a preliminary estimate for θ, so we'll call it θ',

$$\theta'_y = \hat{\theta}_y^{k-1} + \Delta t \cdot s \left(\gamma_y^k - \hat{b}_y^{k-1} \right) \tag{4}$$

Note that we started with the *estimate* of the angle at the previous time-step, $\hat{\theta}^{k-1}$. Also, because the bias at timestep k is unknown (we'll update it below), we use our *current best guess* for the bias, which is b^{k-1}.

Next, let's assume that you've processed the (noisy) accelerometer readings to calculate the orientation angle. We'll call the resulting *observation* of the orientation angle, $\tilde{\theta}_y^k$,

$$\tilde{\theta}_y^k = \tan^{-1} \left(-\frac{\alpha_x^k}{\alpha_z^k} \right)$$

[1] Otherwise, we could just adjust the estimate and the error would be zero.

[2] But not in RBE 3002!

[3] There is also a noise term, but since the noise can never be determined, we leave it out of the equations for simplicity.

To correct the prediction, average the preliminary estimate with the observation from the accelerometer,

$$\hat{\theta}_y^k = (1-\kappa)\theta'_y + \kappa\tilde{\theta}_y^k \tag{5}$$

where κ is a weighting factor that allows us to put more weight on one sensor or the other.

To see why this is useful, note that γ is measuring a rate (i.e., a derivative), so Equation 4 has the same form as a high-pass filter. Further, Equation 5 can be re-written,

$$\hat{\theta}_y^k = \theta'_y + \kappa\left(\tilde{\theta}_y^k - \theta'_y\right) \tag{6}$$

which has the form of a low-pass filter for the accelerometer. In the end, we've combined a high-pass filter for the gyroscope and the low-pass filter for the accelerometer – exactly what we want!

We call the last term in parentheses the *innovation*.

The final step is to update the bias. For that, we use another low-pass filter: the new bias is just the old bias, updated using innovation,

$$\hat{b}^k = \hat{b}^{k-1} - \epsilon\frac{1}{\Delta t \cdot s}\left(\tilde{\theta}_y^k - \theta'_y\right) \tag{7}$$

where $\epsilon << 1$ so that the bias is slow to change and we divide by Δt to keep the units correct. In the activities below, you'll ignore the bias for most of the development, and then add it in at the end.