Module 1: Sensor Data Acquisition and Processing in Android

1 Summary

This module's goal is to familiarize students with acquiring data from sensors in Android, and processing it to filter noise and to gain insight. Students will model and design sensor-driven applications before implementing their designs into Android applications.

2 Hardware Requirements

Any Android compatible device with an accelerometer and a touch screen.

3 Time Needed

- 60 minutes for the basic application

4 Additional References

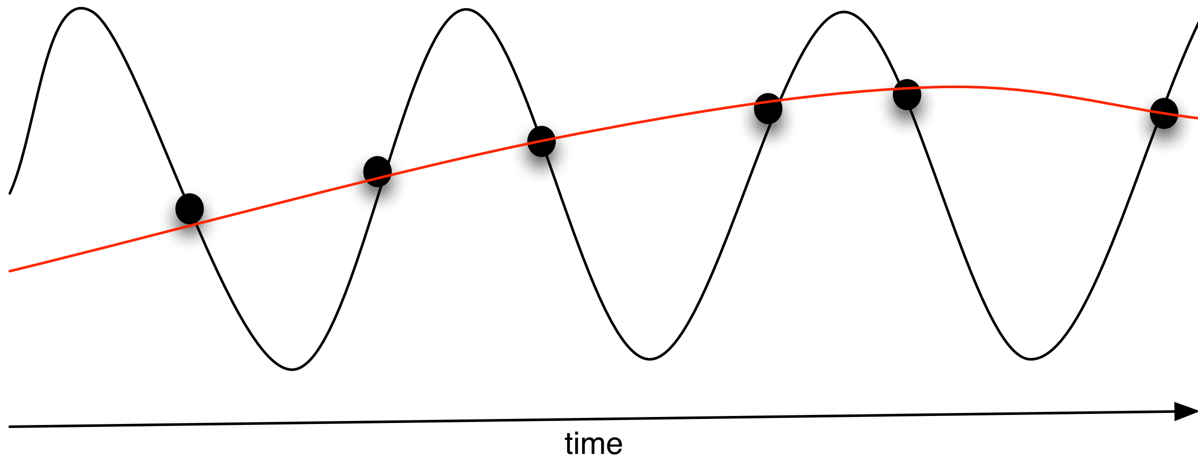
-
- "Finite State Machines and Modal Models in Ptolemy II" by Edward Lee.
  http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.pdf
- Android Developer's Guide. *http://developer.android.com/guide/index.html*

5 Signal Processing in Android

Most things in nature are continuous, e.g. the sound we hear, the images we see, etc. However, continuous functions cannot be directly represented on a computer. Instead, we usually gather samples of these functions at a moment in time. We store these samples and use them to reconstruct the original continuous function.

For instance if we are recording sound using a microphone we would set an Analog to Digital Converter (ADC) to sample the signal (changes in air pressure at the microphone) multiple times per millisecond and use a number to represent each sample. These samples can be converted back into voltage using a Digital to Analog Converter (DAC) and then into sound using some type of playback device (e.g. speakers).

The number of samples we need to fully reconstruct the original sound depends on how high-pitched the sound becomes. A sampling rate that can reconstruct a drum may differ from that one we need to reconstruct cymbals. Digitizing the cymbals with a lower sampling rate may result in an inaccurate recording, and a strange sound heard during playback. This is due to a phenomenonon known as *aliasing*. In the figure below we see a worse case, where based on the samples we reconstruct a signal (the red, slow sine wave) that is very different from the actual signal (the black, fast sine wave) due to insufficient samples (circles).



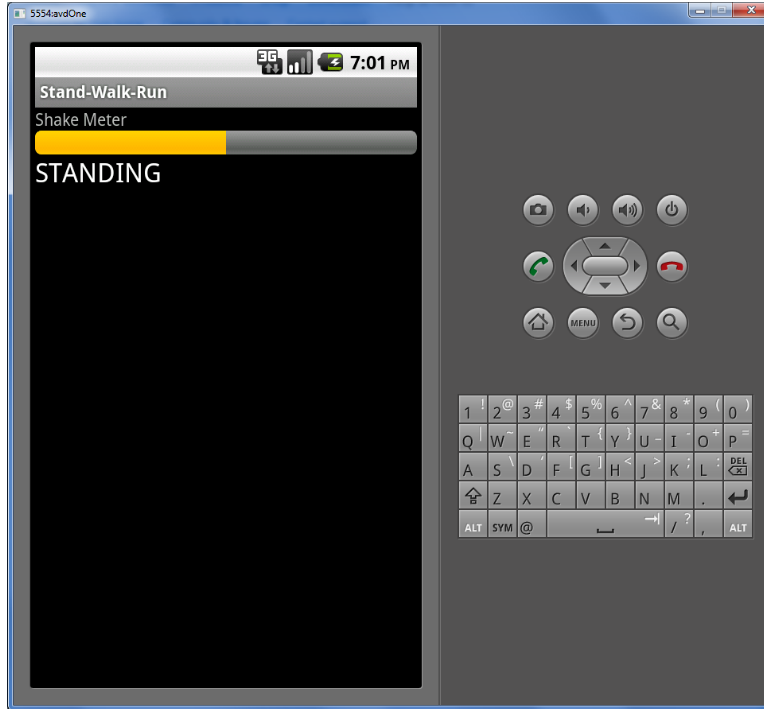
A couple of questions of interest in signal processing are:

- What sample rate do we need to capture a given type of signal?
- What kind of smoothing or filtering can we perform to reduce some of the aliasing effects?

In some cases (e.g. the cymbals) we may have recovered something that is close to the actual signal, and some small mathematical operations on the signal samples may help us to improve our signal. For instance, we can perform a moving average of the signal, which will smooth some of the adverse aliasing effects (a.k.a a box filter). Intuitively, this will reduce some of the spikes in the signal that may have randomly appeared in our samples, and help the resulting signal seem more reasonable (think of a smoother playback sound in the speakers). A small improvement to a simple moving average can be to add weights to it, so that, for instance, newer readings are weighed as more important than the past data. This is one idea, but in fact signal processing has a number of fairly sophisticated algorithms used to remove the effects of aliasing and noise, and determine a particular type of information encoded in the signal. In this course module, we will use some of the more basic approaches of signal processing on our Android device, which produces a number of signals through a variety of sensors (e.g. camera, microphone, accelerometer, gyroscope, GPS).

5.1 Application 1: Stand, Walk, Run

Most Android devices have an accelerometer sensor, whose main purpose is to detect the rotation of the phone in your hands. This is used by the OS to change the screen orientation between portrait and landscape. However, this sensor can perhaps be used for other things. In this application, we would like to use the phone's accelerometer to determine the amount of shaking that the phone is experiencing. This will allow us to determine whether the person holding the device is **standing** (none or almost no shaking), **walking** (some shaking) or **running** (a lot of shaking).



To clasify the state of the phone into one of these readings, we need a way to combine the acceleration measured by the accelerometer on 3-axes: X, Y and Z. These axes are relative to the Android device. In other words, if you are holding the phone (in portrait orientation) the X axis is the vertical, the Y axis is the horizontal, and the Z axis is perpendicular to the surface of the device. Since we are interested in shaking, which is irrespective of direction we just combine the per axis acceleration measurements A_x, A_y and A_z into one measurement of the acceleration magnitude:

$$A_{mag} = \sqrt{A_x^2 + A_y^2 + A_z^2}$$

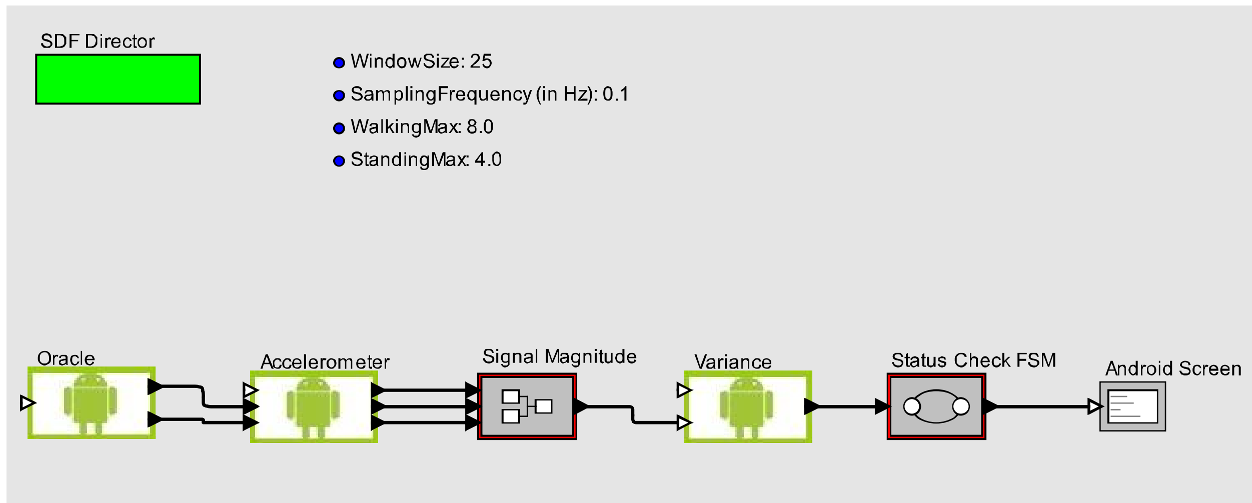
To compute the shaking we need a measurement of multiple moments of acceleration. A lot of shaking would be exhibited by high acceleration followed by a moment of no acceleration followed by more acceleration. This type of behavior can be easily calculated as the variance of a number of A_{mag} readings, computed as:

$$V = \frac{1}{n} * \sum_{i=1}^{n} (x_i - \mu)^2$$

We calculate the vairance using a sliding window of n readings of the accelerometer, taken at t Hz (or once every t seconds), to calculate the variance, and to classify the state of the device (stopped, walking, running). In Ptolemy II, You will get the opportunity to experiment with values for t and n, and while an exact correct value for these does not exist, there is definitely a range of values that will work and a range that will make the application work poorly.

To make the final distinction of the state of the device, we need to pick the borderline values for the variance. For instance, we could choose a variance of 2.5 as the borderline value between stopped and walking, so variances below 2.5 we would classify as stopped, and those above 2.5 as walking. To choose this magic borderline number we we divide the variance space in 3 equal parts using experimentation on our Ptolemy II model, which you will do next.

5.1.1 Starter PtolemyII Model



The starter model for this application simulates the data generated by an accelerometer sensor on an Android phone, using the `Oracle` and `Accelerometer` actors. They generate random, but constrained, accelerometer data, and we can control whether this data is mostly `Stopped`, `Walking` or `Running`, so that we can test the design of our application based on it. The `Oracle` actor creates a GUI allowing the user to choose which category should the simulated data belong. Then, this data is processed by a `SignalMagnitude` actor, calculating the magnitude of the acceleration, as detailed by the formula above. The statistical variance of this magnitude is calculated by the `Variance` actor, which contributes this value to a FSM that classifies the data as one of the three states, `Stopped`, `Walking`, and `Running`.

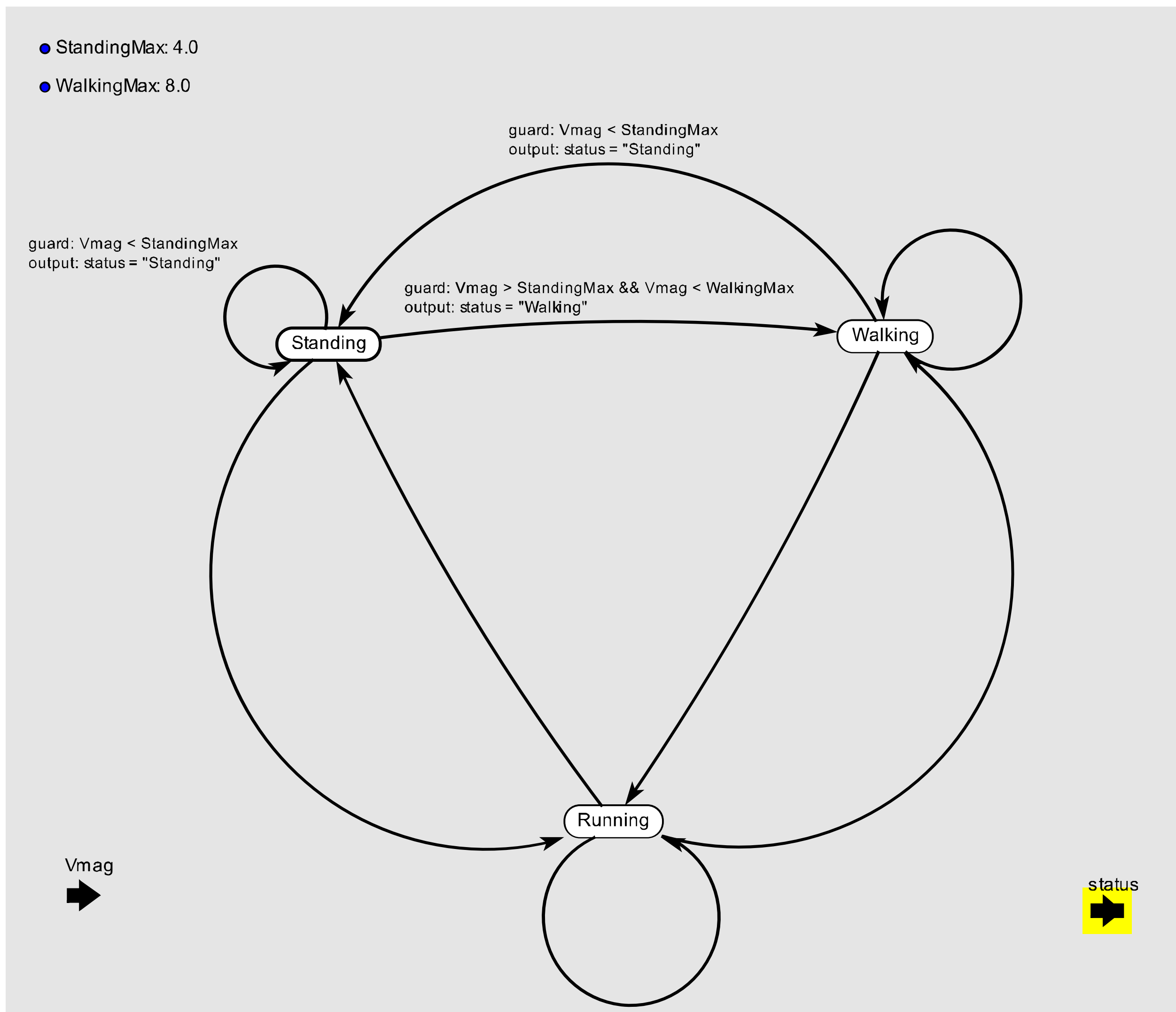
Both the `SignalMagnitude` and the `Status Check FSM` actors are incomplete and require your models work correctly. Once completed, the resulting simulation should produce the corresponding entries in the `Android Screen` to the generated data chosen in the `Oracle`

Variables on the top of the model allow for experimentaion with the values for window size, and the classification borders between stopped, walking, and running. Ideally, you can already figure out reasonable entries for these values during the modeling phase, and use those exact values during your implementation on the Android phone.

5.1.2 Modeling

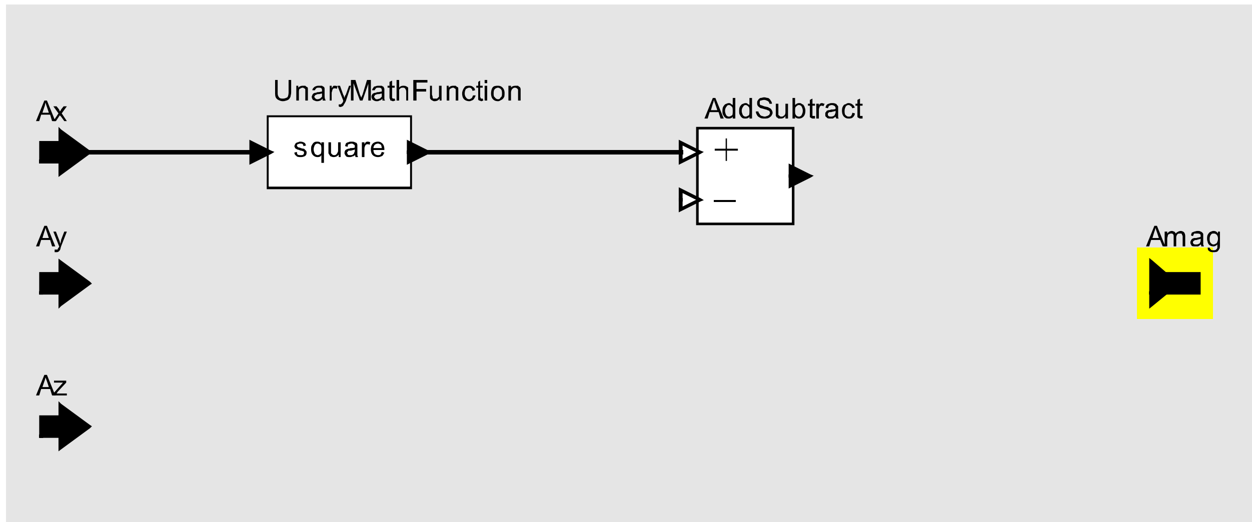
Once you have the starter model downloaded, you will notice that its missing an implementation of the `SignalMagnitude` and the `Status Check FSM` actors. Here, we describe how each of those can be modeled.

Status Check FSM. The Ptolemy model of this actor consists of two parts: the calculation of variance based on aggregated accelerometer readings and the process of determining the activity based on the variance. The latter part is depicted in this FSM, which consists of three states `Stopped`, `Walking` and `Running` determined based on the single input, the variance `V`. The FSM outputs a string as the output, via the `status` output port.



Notice that the above FSM provides direct transitions between each of the states, even between `Stopped` and `Running`. This means that the application should be able to classify two subseqent variances as belonging to each of these states, and does not require a pass through the `Walking` state.

Calculating the acceleration magnitude and variance based on simulated accelerometer readings is also modeled in Ptolemy. This process starts with the generation of accelerometer readings in the `Accelerometer` actor, which are A_x, A_y, and A_z. These three values are combined to form the magnitude of acceleration (according to the formula above in this text) in thie `SignalMagnitude` actor. Below is an incomplete version of that calculation - *it's up to you to complete it.* Finally, an actor that we provide, called `Variance`, does the final calculation of the statistical variance of a `WindowSize` number of acceleration magnitudes.



5.1.3 Implementation

Today's Android devices contain several hardware sensors, which collect data about a number of environmental conditions (e.g. acceleration, location, direction, etc.). In your application, the way to interface to these sensors is via the `SensorManager`, `Sensor`, `SensorEventListener` and `SensorEvent` interfaces, available in the `android.hardware` namespace. The sensor workflow in your application will consist of the following 2 steps: (1) determine whether a particular sensor is available; and (2) listen for events generated by the sensor that contain the latest data values.

To determine whether a sensor is available and get a handle to a particular sensor we use the `SensorManager` interface.

```
private SensorManager mSensorManager;
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
private Sensor mSensor;
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
if(mSensor == null)
{
  //no accelerometer exists on your device
  //so tell the user and then exit the app
}
```

The `Sensor` interface can also be used to determine more information about the particular hardware sensor (e.g. manufacturer name). Consult online Android information for more information on that interface.

One final step to get data from the sensor (accelerometer in our case) is to register a listener class. This listener class needs to inherit the `SensorEventListener` interface, providing a couple of mandatory methods: `onAccuracyChanged` and `onSensorChanged`. Note that this means that the user implements these two methods, and the Android OS will invoke them when each event occurs. Invocation to the `onAccuracyChanged` means that something changed in the type of data the sensor is able to produce, and is usually not very important to implement unless you care deeply for the accuracy of your data. Invocations to the `onSensorChanged` carry a new value for the sensor data, packed in the `SensorEvent` structure.

For this problem, we will provide you with a class that implements the `SensorEventListener` interface, called `AccelerometerReader`. You can register this class with the Android OS, using the following line:

```
private AccelerometerReader accelerometerReader = new AccelerometerReader();
mSensorManager.registerListener(accelerometerReader, mSensor, SensorManager.
    SENSOR_DELAY_NORMAL);
```

Reusing the Modeled Information. For this application, you will need to create a thread (using the `Handler` class, as we did in the previous module). The thread will do the following list of tasks: (1) get data from the accelerometer; (2) calculate *Amag* and save it's value into an array of n elements; (3) when the array becomes full (i.e. contains exactly n elements) calculate the variance and clear the array; (4) use the variance to determine (using the FSM logic) the movement category and change the UI.

The value for n will come from your model, as well as at what granularity to schedule the thread itself, corresponding to the sampling rate value from the model. Hopefully, also, you now know how to calculate A_{mag} from the model, and can translate this into Java code. The variance is calculated using the following formula:

$V = \frac{1}{n} * \sum_{i=1}^{n}(x_i - \mu)^2$

Also, remember how we encoded FSMs using a `enum` and a state variable in the previous module. Do the same here to translate the FSM you came up with which will make the final distinction of what state the user is using the `StandingMax` and `WalkingMax` constants from your model.

```
enum UserState {
     Standing,
     Walking,
     Runniing
}

UserState status = UserState.Standing;
...

if(status == UserState.Standing && variance > StandingMax) {
  status = UserState.Walking;
}
...
```

5.1.4 Application Extensions

- **Measure of Correctness.** Add buttons to the application so that the user can give feedback of whether he or she is standing, walking or running. Then compare the user feedback as the ground truth to the algorithm's decisions in order to determine the algorithm's accuracy. The final result should be a accuracy percentage of the algorithm (e.g. the algorithm is 95% accurate).
- **Biking.** Add riding a bike to the set of possible activities distingueshed by the algorithm. Biking should exhibit even more shaking than running, but it may need a larger window of measurements. Experiment with these parameters to get the algorithm to classify the activities with reasonable accuracy.