CM605
Embedded Systems Engineering

N-version Programming Simulation Exercise

This is an investigation of *N-version programming* using a java-based simulation of a *noisy* and possibly *unreliable* sensor.

Download and unzip Simulation.zip. This zip contains the following java classes:

1. **SensorSim** - simulates a *noisy* sensor: the value returned by its method **double getRdg()** is a *nominal* value plus or minus a a random amount of *noise*. (The noise is simulated using probability-theoretic techniques – noise values are random numbers with a *normal* probability distribution with a configurable *standard deviation*. The larger this is, the noisier the simulated sensor.)

To use the class, all you need to know about is

- an instance's two configurable attributes, the nominal value and the amount of noise (the mean and standard deviation of the normal probability distribution).
- The constructor takes as parameters an initial nominal value and amount of noise.
- Method **start()** starts the simulation running in its own thread.
- Method **setNominal(double)** adjusts the nominal value even when the simulation is running.
- Method **double getRdg()** retrieves a sensor reading (nominal val + noise).

The source code is available on request for your interest, but you do not need to see it, and should certainly not change it – use the class as a "black box".

2. The class **Simulation** runs a simulated sensor with nominal value initially 100 and noise (standard deviation) 5, and repeatedly at 500 millisecond intervals

- retrieves readings and displays them on a graph using a helper class, **DataDisplay,**
- outputs data to the console in the format *reading* (*nominal*): *difference.* By redirecting output to a text file you can log this output for experimental purposes.
- random-walks the nominal sensor value up and down.

Try it now: `java Simulation`.

You can run the simulation other other initial nominal values and noise values but we shall stick with 100, 5 for the investigation.

3. **Simulation.java** is the source file for the Simulation class. This is the only code you will need to read in detail or modify.

4. Class **DataDisplay** is a totally boring utility class that displays sensor readings on a graph. The blue trace is the time-series of sensor readings. The green line shows the current nominal value. The details of this class are not important: the only thing you need to know

about is its method **update(**double **reading,** double **snsNom)** which adds a sensor reading to the graph trace and updates the sensor nominal value display.

5. A *faulty sensor* is one which sporadically gives an erroneous reading. **FaultySensorSim** is a subclass of SensorSim which does this. Two extra attributes need to be supplied to the constructor (besides the initial nominal value and the noise value) – an integer specifying the fault frequency, and a double which is an erroneous value returned by getRdg() instead of the correct (though noisy) one, when a fault occurs. Fault occurrence is simulated by a "Poisson" process (like cars passing or rain drops landing!) and the frequency parameter is the mean interval between faults, in milliseconds.

Try using a FaultySensorSim instead of a SensorSim instance in the Simulation. Open the source file Simulation.java and find the line in the `runSimulation` method which constructs a SensorSim.

```
sensor = new SensorSim(sensorNom, sensorErr;
```

Change this to -

```
sensor = new FaultySensorSim(sensorNom, sensorErr, 10000, 200);
```

Re-compile and run this to see the effect of a faulty sensor with a fault interval of 10000 milliseconds and a fault value of 200. Try other fault frequencies and fault values.

Introducing the Investigation

Imagine a set-up where a sensor monitors some condition of the plant and the plant must be shut down immediately if the reading becomes dangerously high.

In reality, the sensor will be "noisy" like the simulated one, and the "noise" level may well be around 2.5 -- 5% of the full-scale reading of the sensor. In the simulation, imagine the full-scale reading is 200: so a noisy sensor with noise = 5 is realistic.

Suppose that we have to do an emergency shut-down if the value reaches 150. A noisy sensor might exceptionally trigger a shut-down when the nominal value is only 140 or even less, because "noise" boosts the reading to 150. We may decide we have to live with that. Alternatively we can create several sensors and take as our reading their *average.* Theory predicts that the "noise" would cancel out to some extent, and the noise in the average of n sensor readings is only $(1/\sqrt{n})^{th}$ of the noise in a single reading. Of course, it is probably uneconomic or impractical to have a very large number of sensors.

There is another potential problem, however. Our sensor might be faulty, and give us a high reading – a "hard-over" fault - from time to time and this could trigger a "false-alarm" shut-down. The shut-downs are required for safety reasons, but are expensive and we really need to avoid false-alarms. We might again employ several sensors, of which one might be faulty. Averaging the readings again will give some protection from a faulty high reading but the average could still be much higher than it should be. A better approach might be to

implement a *voter* which checks all the readings are within tolerance of each other and rejects "out-liers", returning the average of the remainder.

Warm up Task

Record the output of a couple of minutes' run of the basic Simulation application as provided (ie, using a SensorSim with nominal value initially = 100 and noise = 5). Redirect the output to a text file thus: `java SensorDataDisplay > log.txt`. Find out the maximum discrepancy of a sensor reading from its nominal value (ie maximum noise). Opening `log.txt` in a spreadsheet is a good way to do this. Also, using spreadsheet's AVERAGE and STDEV functions you can determine the mean and standard deviation of these discrepancies over the run.

Do not submit the detailed log – just very abridged listing (a few lines) and report the number of outputs, the maximum discrepancy and the mean and standard deviation of the sensor readings.

Do this again for the output of a couple of minutes' run of a Simulation application in which a FaultySensorSim is substituted, as explained above. How often are the faults occurring?

Main Task

Make a copy of Simulation.java and change the code in the `runSimulation()` method so that an array of SensorSim objects are employed rather than a single one. They should all be given the same initial nominal value and noise value and the random-walk should adjust them all in the same way. The reading should be an average of their readings.

Rather than hard-coding the size of this array, make it a parameter passed in through the constructor.

Make one of the simulated sensors is a FaultySensorSim and the remainder noisy but non-faulty are two other (non-faulty) \texttt{SensorSim} instances. Investigate the behaviour of this simulation with three (1+2) sensors: how high can the nominal value go before an emergency shut-down is triggered? Repeat with four (1 faulty + 3 non-faulty) sensors.

Can you devise a voter function that gives better protection against `false alarm' shut-down?

One idea to reject any reading that deviates too far (more than twice the noise level) from the average of the other readings, taking this as `the' average reading.

Investigate the ability of this voting approach to protect the system from false emergency shut-down due to a single faulty sensor and give a comparison with the simple averaging approach.

Listing - Simulation.java

```
/* Simulation
 * Sensor sample interval is 500 ms. Nominal sensor value random-walks up
 * and down from an initial value of 100; sensor "noise" = 5.
 *
 * Text output of readings and associated nominal values are output on
 * console and can be redirected to a log file for experimentation.
 */
import java.util.*;

public class Simulation {
  private DataDisplay display;
  private SensorSim sensor;
  private double sensorNom;       //nominal value -- random-walks
  private final double sensorErr; //fixed

  public Simulation(double n, double e) { //constructor
    sensorNom = n;
    sensorErr = e;
    display = new DataDisplay();
    runSimulation();
  }

  public void runSimulation()  {
    Random rng = new Random();
    sensor = new SensorSim(sensorNom, sensorErr);
    sensor.start();
    while(true) {
      double rdg = sensor.getRdg();
      System.out.printf("%7.2f (%5.1f): %4.1f\n", rdg,
         sensorNom, rdg-sensorNom);
        //sensor:output(nominal):difference on console; can be redirected to log
      display.update(rdg, sensorNom);

      if (rng.nextBoolean())  //nominal sensor output random-walks up & down
        sensorNom++;
      else
        sensorNom--;
      sensor.setNominal(sensorNom);

      try {  // 0.5-second sleep
        Thread.sleep(500);
      } catch (InterruptedException ix) {}
    }
  }

  public static void main(String[] args) {
    if (args.length < 2) {
      System.out.println("Using defaults initial nominal = 100.0, noise = 5.0");
      System.out.println(
          "For other settings use java Simulation <nom> <noise>");
      new Simulation(100, 5);
    }
    else {
      new Simulation(Double.parseDouble(args[0]), Double.parseDouble(args[1]));
    }
  }

} //end class Simulation
```