Real-Time Systems

Laboratory 1: Control of the Ball and Beam Process

2022 Version

Preparatory Exercise

It is **required** to have finished Exercise 3 before coming to the lab and to bring the corresponding code with you. You should also read through the lab manual before coming to the exercise session. **Additionally, note** that almost all of the exercises in this manual can be completed prior to the lab session.

Code Skeleton

To simplify the coding process, we have provided a code skeleton, including all the java files you will need. It can be downloaded on Canvas.

Introduction

Your task is to write a Java program that controls the ball and beam process. The Java program should consist of two parts:

- `OpCom`, a version of the Swing-based GUI from Exercise 4. Provided to you.
- `Regul` from Exercise 3, structured as `BallAndBeamRegul`, but with the public interface provided to you.

The main difference between the new Regul class and the one used in Exercise 3 is that the GUIs for the inner PI controller and the outer PID controller are now implemented by `OpCom` rather than by internal GUI classes. Thus, it is **important** that you **do not** include the `PIGUI` and `PIDGUI` classes from Excerise 3 and that you use the `OpCom` class we are providing you here.

A predefined `ReferenceGenerator` class is used to provide the reference signal. This class has its own GUI. From the GUI you can decide to either use a squarewave signal as the reference or to manually set the reference using a `JSlider`.

ModeMonitor

The public interface of `ModeMonitor` is the following:

```
  public class ModeMonitor {
  private Mode mode = Mode.OFF; // Off mode to start with

  // Sets new mode
  public synchronized void setMode(Mode newMode) {
    mode = newMode;
  }

  // Returns the current mode
  public synchronized Mode getMode() {
    return mode;
  }

  // Existing modes
  public enum Mode {
    OFF, BEAM, BALL;
  }
}
```

This class exists to make the shared resource `Mode` mutually exclusive.

OpCom

The public interface of `OpCom` is the following:

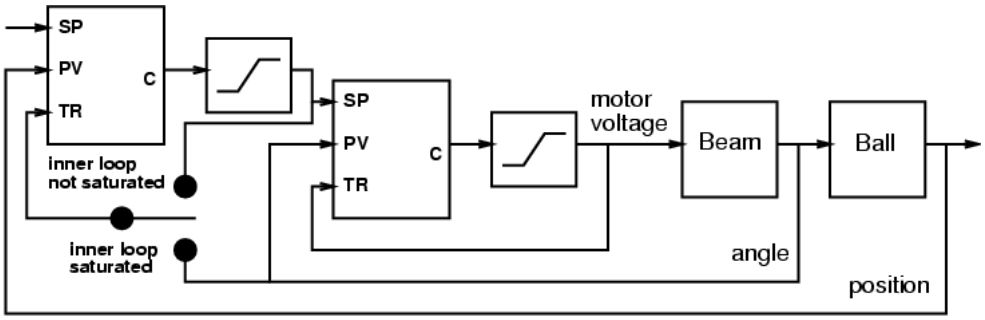
```
  // Constructor. Note: Different from Exercise 4.
  public OpCom(int plotterPriority, ModeMonitor modeMon);

  // Passes in a reference to Regul. Called from Main.
  public void setRegul(Regul r);

  // Build up the GUI. Called from Main.
  public void initializeGUI();

  // Starts the plotting within OpCom. Called from Main.
  public void start();

  // Plots a new control signal data point
  public void putControlData(double t, double u);

  // Plots a new measurement data point
  public void putMeasurementDataPoint(double t, double yRef, double y);
```

The control modes are acquired through the `ModeMonitor` class.

Regul

The `Regul` class should contain one PI controller for the inner loop and one PID controller for the outer loop. Use a single thread for the execution of both the controllers in the same way as in the exercise. `Regul` receives its reference signal by calling the `getRef()` method of the `ReferenceGenerator`.

The controller should use the cascade structure according to the figure below. The code should be written so that the **delay** between the sampling of the measurement signals and the generation of the control signal is **minimized**.



`Regul` should run in one out of three different modes: OFF, BEAM, and BALL (the mode is kept by the `ModeMonitor`). In each loop, the regul thread should call `sendDataToOpCom()` to send the control and measurement data point to the OpCom. If the mode is OFF the signals sent to `OpCom` (y, yref and u) should all have the value 0.

A major difference compared with the exercise is that you now use real analog IO classes rather than the classes that communicate with the virtual process. The following code example shows how to use these classes:

```
import se.lth.control.realtime.*;

class IODemo {
  public static void main(String[] args) {
    AnalogIn yChan;
    AnalogOut uChan;
    double y;
    try {
      uChan = new AnalogOut(1);
      yChan = new AnalogIn(1);
    } catch (Exception e) {
      System.out.println(e);
      return;
    }

    try {
      uChan.set(0.0);
    } catch (Exception e) {
      System.out.println(e);
    }

    try {
      y = yChan.get();
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

Another difference compared to Exercise 3 is that the reference signal now is plotted by the GUI rather than by the virtual simulator. Hence, the reference signal should no longer be output using an analog output. Instead, the reference signal should be sent to `OpCom` by calling `sendDataToOpCom()`.

Regul Public Interface

The Regul class should have the following public methods:

```
// Constructor
public Regul(int priority, ModeMonitor modeMon);

// Passes in a reference to OpCom. Called from Main.
public void setOpCom(OpCom o);

// Passes in a reference to ReferenceGenerator. Called from Main.
public void setRefGen(ReferenceGenerator r);

// Sends the data to OpCom
public void sendDataToOpCom(double yRef, double y, double u);

// Set the parameters of the inner PI controller (inner controller).
// The PIParameters class is the same as in Exercise 3 (without
PIGUI).
// Not synchronized, the synchronization is handled by PI.
public void setInnerParameters(PIParameters p);

// Returns the current parameters in the inner loop.
// Only called during OpCom GUI initialization.
// Not synchronized, the synchronization is handled by PI.
public PIParameters getInnerParameters();

// Set the parameters of the outer PID controller (outer controller).
// The PIDParameters class is the same as in Exercise 3.
// Not synchronized, the synchronization is handled by PID.
public void setOuterParameters(PIDParameters p);

// Returns the current parameters in the outer loop.
// Only called during OpCom GUI initialization.
// Not synchronized, the synchronization is handled by PID.
public PIDParameters getOuterParameters();

// Called from OpCom when the Stop button has been pressed,
```

```
  // before the system is shut down.
  public void shutDown();
```

PI and PID

Use your PI and PID classes (removing the `PIGUI` and `PIDGUI` dependence) from Exercise 3 with the following extensions:

```
  // PI

  // Sets the I-part of the controller to 0.
  // For example needed when changing controller mode.
  public synchronized void reset();

  // Returns the current PIParameters.
  public synchronized PIParameters getParameters();



  // PID

  // Sets the I-part and D-part of the controller to 0.
  // For example needed when changing controller mode.
  public synchronized void reset();

  // Returns the current PIDParameters.
  public synchronized PIDParameters getParameters();
```

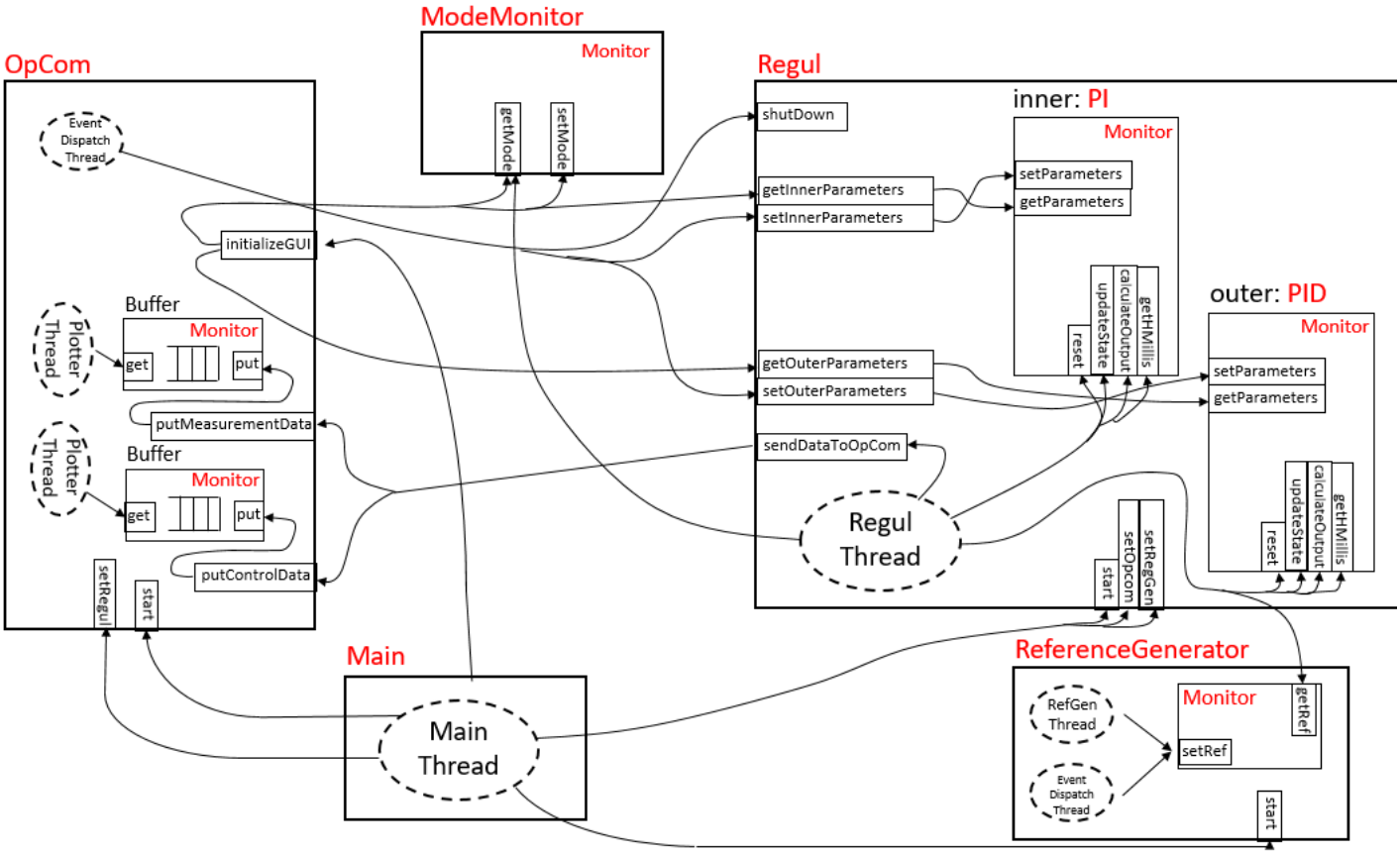
Main

The following Main class should be used.

```
  import javax.swing.*;

  public class Main {
    public static void main(String[] argv) {
      // Initialise ModeMonitor
      ModeMonitor modeMon = new ModeMonitor();

      // Thread priorities
      final int regulPriority   = 8;
      final int refGenPriority  = 7;
      final int plotterPriority = 6;

      // Initialise Control system objects
      ReferenceGenerator refGen = new ReferenceGenerator(refGenPriority);
      Regul regul = new Regul(regulPriority, modeMon);
      final OpCom opCom = new OpCom(plotterPriority, modeMon);

      // Setting dependencies
      regul.setOpCom(opCom);
      regul.setRefGen(refGen);
      opcom.setRegul(regul);

      // Start threads
      Runnable initializeGUI = new Runnable() {
        public void run() {
          opCom.initializeGUI();
          opCom.start();
        }
      };
      try {
        SwingUtilities.invokeAndWait(initializeGUI);
      } catch(Exception e) {
        return;
      }

      refGen.start();
      regul.start();
    }
  }
```

Structure Diagram

The design can be summarized in the figure below. The notation introduced in the Buttons exercise is used.



NOTE: In the diagram, the PI and PID classes are drawn as if they are inner classes of Regul but they should be kept as ordinary, "top-level", classes.

Process Interface

The output signals from the process (angle and position) are voltages in the interval [-10, 10]. The angle measurement should be connected to AnalogIn 0 and the position signal to AnalogIn 1.

The input signal to the process (the control signal) is a voltage in the interval [-10, 10]. The control signal should be connected to AnalogOut 0.

---

1. Update your PI and PID classes using the modified interface.
2. Implement `Regul` based on your `BeamRegul` and `BeamAndBallRegul` classes from Exercise 3. It should use the **external IO** instead of the simulator's IO. Make sure that it sends plot data to `OpCom` in the correct way. We have provided a code skeleton as a starting point.

Make sure that:

1. The IO delay is minimized (i.e., the delay between sampling of the measurement signals and the generation of the control signal).
2. The lock times aren't longer than necessary.
3. The outer loop tracking is in accordance with the structure figure above.

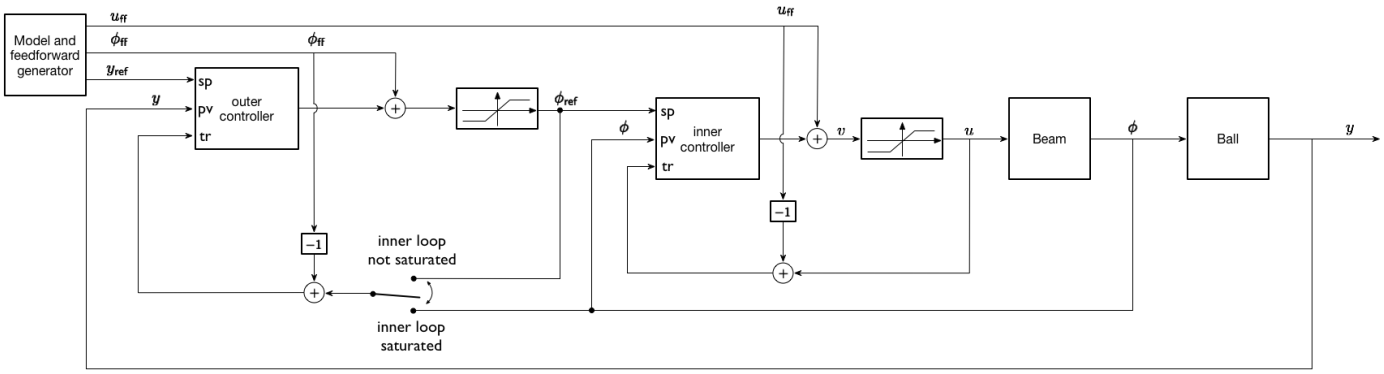
3. Test your program against the real ball and beam process.

    - Start by ensuring that your program work in the BEAM mode. Start with the controller parameter values that you used for Excercise 3.
    - Once your program works in BEAM mode, go on to BALL mode. Start with the controller parameter values that you used for Exercise 3.

Time-Optimal Feedforward Generator

In the next part you will extend the previous control-structure with a time-optimal feedforward generator. You have already been provided all the necessary classes for this, and will only have to do some minor changes in order to make it work. Before moving on, you should recall the lecture slides from *Lecture 10*.

Below you can see a block diagram for the new control structure. To change your current control-structure into this, you will need to call two new methods from the reference generator in order to get the feedforward terms: `referenceGenerator.getPhiff()` and `referenceGenerator.getUff()`. You should also recall from the lecture that when using the feedforward generator you will need to use gamma=1 when computing the D-term in the PID-controller. Therefore, in `PID.java` you will have to change `D = ad*D - bd*(y - yOld)` to `D = ad*D + bd*(e - eOld)` (**NOTE** the plus-sign!)



1. Update the `Regul.java` to use the new feedforward signals
   - The outer loop can use the method `getPhiff()` to retrieve the feedforward term
   - The inner loop can use `getUff()`
   - When implementing the anti-windup tracking, you should use `updateState(u-uff)` for the inner loop, and `updateState(angRef-phiff)` for the outer loop (or `updateState(ang-phiff)` if the inner loop is saturated)
2. Update `PID.java` to use gamma = 1:
   `D = ad*D - bd*(y - yOld)` should be changed into
   `D = ad*D + bd*(e - eOld)`
3. Compile and test the new system using time-optimal feedforward generator. Remember to switch the reference generator into "Time-optimal mode". Do you see any difference?

Analysis

Make sure you understand (and can answer) the following questions. The lab supervisor can ask you to motivate one or more of the questions.

1. Give *at least* three shared resources that we have to handle with extra care.
2. Why do we put the `synchronized` keyword on every method in the `PI` and `PID` classes? Motivate.
3. Why would it be poor design to add an integrator to the inner loop controller?
4. In the course so far we usually sleep threads using the following code snippet:

```
t += h; // t was the previous release time and h is sample time
duration = t - system.currentTimeMillis();
if (duration > 0) {
  try {
    sleep(duration);
  } catch (InterruptedException x) {
    // Do something
  }
}
```

   Explain what it means for the controller that `duration` is less than or equal to zero.
5. Briefly describe why we are using feedforward.