CSG2341 Intelligent Systems

Workshop 3: A fuzzy cruise controller

Related Objectives from the Unit outline:

- Identify appropriate intelligent system solutions for a range of computational intelligence tasks.
- Demonstrate the ability to apply computational intelligence techniques to a range of tasks normally considered to require computational intelligence.

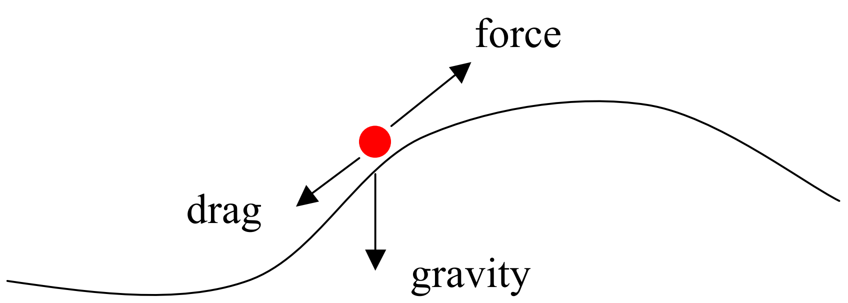
Learning Outcomes:

After completing this workshop, you should be able to demonstrate an understanding of the resources required to implement a problem solution based on fuzzy reasoning, to describe and analyse the steps in designing a fuzzy system to carry out approximate reasoning (in this case, for a control system) and use a computer package to develop such a fuzzy control system.

Background

One of the main applications for fuzzy systems is to control systems. In the lecture, we developed a controller for an inverted pendulum. In this workshop, you will implement a cruise control system. If you get stuck, you could take a look at the inverted pendulum controller solution.

On BlackBoard you will find a set of Java source files, in the zip file CruiseControl.zip, that simulate a vehicle travelling over hilly terrain, using a cruise control system to maintain constant speed. The diagram below shows how this scenario is modelled.



The vehicle (the red dot) is travelling from left to right. The forces acting on it are a drag force (air resistance, friction etc) that tends to slow the vehicle, gravity, which tends to slow down (uphill) or speed up the vehicle (downhill), and the force being applied by the engine and brakes of the vehicle.

The cruise control system tries to keep the speed of the car at a constant value of `30` m/s. While other choices are possible, these are the fuzzy variables that you will use:

Inputs (sensors):

- speedError: this is the actual speed minus the target speed. For example, if the actual speed is 35 m/s, then the speed error is 5 m/s. If the actual speed is 25 m/s, then the speed error is -5 m/s. Thus **speed error can be positive or negative!** An advantage to using speed error instead of speed, is that the target speed can be changed without having to change the fuzzy rules.
- acceleration: notice that this is a **measurement** - it measures how fast the car is speeding up or slowing down – and **not a control action** – i.e. it is not how hard the control system is pushing on the car's accelerator!

Outputs (controls):

- forceChange: the control system will already be applying some force, which could be either pressing on the accelerator, or pressing on the brake. This output variable determines **whether more force or less force needs to be applied**. This is not the same thing as how much force is applied. If it is positive, this means "apply more force". That could mean pressing harder on the accelerator, or easing off on the brake, or switching from the brake to the accelerator. If it is negative, this means "apply less force". An advantage to using force change instead of force is that the rules will be easier to construct – e.g. "If the car is going too fast, apply less force."

Task:

Write a Java class whose instances implement a fuzzy controller for the simulated vehicle.

Your fuzzy controller class should look something like this:

```
import au.edu.ecu.is.fuzzy.*;
/**
 * Your solution for a fuzzy cruise controller.
 *
 * @author (your name)
 * @version (2006/2)
 */
public class MyCruiseController implements CruiseController
{
    public MyCruiseController()
      throws FuzzyException
```

```
{
    // create fuzzy variable for speed error
    // INSERT YOUR CODE HERE

    // create fuzzy variable for acceleration
    // INSERT YOUR CODE HERE

    // create fuzzy variable for force change
    // INSERT YOUR CODE HERE
    // NOTE THAT THIS VARIABLE WILL NOT NEED ANY FUZZY SETS

    // now construct a matrix of fuzzy sugeno-type rules
    rules = new SugenoRuleSet();

    FuzzySet[] speedSets =
        {
        //INSERT YOUR FUZZY SETS FOR SPEED
        };
    FuzzySet[] accSets =
        {
        //INSERT YOUR FUZZY SETS FOR ACCELERATION
        };

    double[][] forceChanges =
        {
              // INSERT 2D ARRAY OF FORCE CHANGE VALUES
        };

   rules.addRuleMatrix(
        speedError, speedSets,
        acceleration, accSets,
        forceChange, forceChanges);
}

public FuzzyRuleSet getRuleSet()
{
    return rules;
}

public FuzzyVariable getSpeedError()
{
    return speedError;
}

public FuzzyVariable getAcceleration()
{
    return acceleration;
}

public FuzzyVariable getForceChange()
{
    return forceChange;
}

public double computeForceChange(
        double speedError,
        double acceleration)
    {
        try
        {
            this.speedError.setValue(speedError);
            this.acceleration.setValue(acceleration);

            rules.update();

            return forceChange.getValue();
        }
        catch (Exception e)
        {
            e.printStackTrace(System.out);
            return 0.0;
        }
    }

    private SugenoRuleSet rules;
    private FuzzyVariable speedError;
    private FuzzyVariable acceleration;
    private FuzzyVariable forceChange;
}
```

Step 1

Download CruiseControl.zip from BlackBoard. Create a folder for this workshop, extract the files, and create a NetBeans project as in last week's workshop.

Now create a new Java source file MyCruiseController.java in your default package and copy the code given above. Look over the code and make sure you understand what each of the methods is doing. Add comments describing each method.

Step 2

Create the fuzzy variables and their fuzzy sets.

A suitable range for `speedError` is -15 to 15, for `acceleration` is -20 to 20, for `forceChange` is -7000 to 7000.

Note that as you will be constructing a Sugeno-style controller, the fuzzy variable `forceChange` will not need any linguistic values.

Step 3

Create and add a matrix of Sugeno-style rules with one method call, in a manner similar to this (This is just an example to show you the syntax – the names of the sets, the number of entries in the arrays, and the values in the arrays will be different from these. For another example, see the inverted pendulum example from lectures. You will need to get proficient at reading and creating these matrices):

```
FuzzySet[] aSets = {a1, a2, a3};
FuzzySet[] bSets = {b1, b2, b3};
double[][] cMatrix =
      {
           // b1,  b2,  b3
            {1.0, 2.0, 3.0}, // a1
            {4.0, 5.0, 6.0}, // a2
            {7.0, 8.0, 9.0}  // a3
      };
ruleSet.addRuleMatrix(
            aVar, aSets,
            bVar, bSets,
            cVar, cMatrix
            );
```

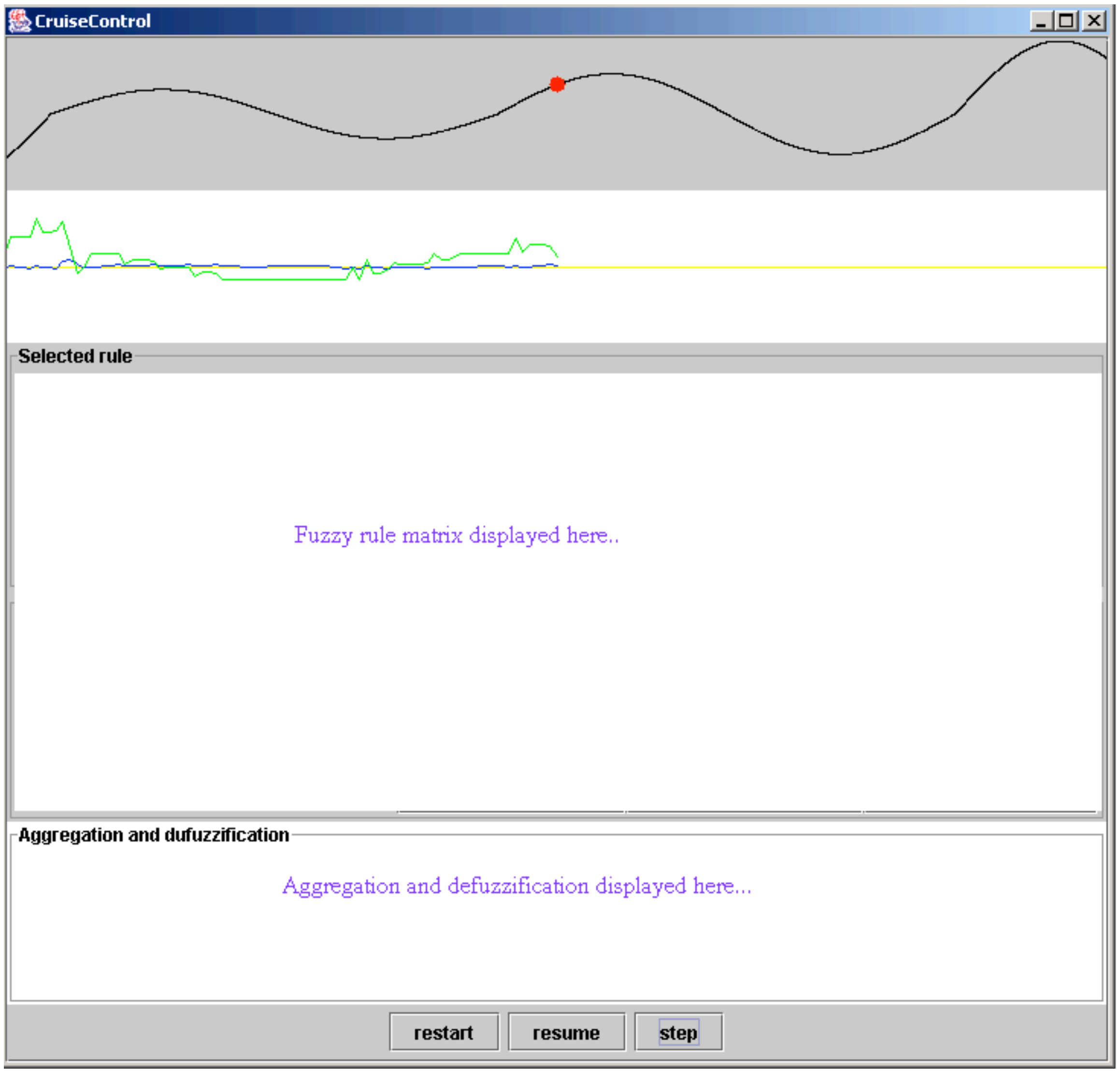
This creates and adds rules like:

> If (a is a1) and (b is b1) then c is 1.0
> If (a is a1) and (b is b2) then c is 2.0
> etc

Step 4

> Test your controller by running the class CruiseControlTest

This will bring up a graphical display that looks like this:


CruiseControl
Selected rule
Fuzzy rule matrix displayed here..
Aggregation and dufuzzification
Aggregation and defuzzification displayed here...
restart
resume
step

In this display, you will see the car (the red dot) moving to the right across the hills (the curved line). Just below that, the orange line represents the target speed, the blue line is the actual speed, and the green one is the force being applied (apologies to the colour blind). The blue line should stay very close to the orange line. Click on “restart” to start again, “pause” or “resume” to stop the simulation so that you can check what is happening, or start it again afterwards, or click on “step” to single-step through the simulation.

When you have completed this workshop, submit your Java source file for MyCruiseController to your tutor using the submission facility on BlackBoard. The workshop is due before midnight Sunday.