Designing a Cruise Control System in RIMS

Owen Webb, Yuzhe Tao, Kelvin Adjei

Background/Motivation

The goal of this project is to implement a cruise control system using the RIMS software package from UC Riverside-Irvine. Cruise control is a common example of a control system because users interact with it often and it is easy to visualize. Students are able to picture a car speeding up or slowing down based on how fast the car is currently going and how fast the driver desires it to go.

The motivation from this project stems from the fact that millions of people drive cars in the US every day, most of which with the ability to use cruise control. Even though it is widely underused for most people, the cruise control system is the closest many people come to a driverless car. We wanted to show how simple it is to simulate a control system such as cruise control. Using RIMS, we would be able to display the output of this simple system and allow the user to interact with it.

Control Systems

A cruise control system is a closed loop system meaning it has a feedback loop that calculates a difference between the current output and the desired output. The system is able to automatically adjust proportionally to how large the difference is. Figure 1 shows a general closed loop control system.

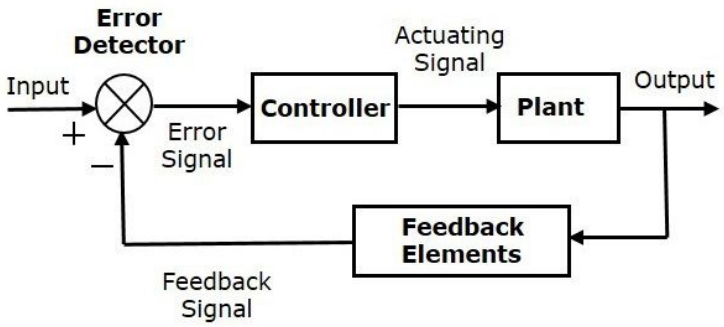

Figure 1. Closed loop control system

The cruise control system is implemented using a Proportional, integral, derivative (PID) controller. The proportional section allows the system to react directly with the error that is calculated and will be the main driver of the system to the desired value. The integral component gives the system the ability to control the duration of the error as it is an integral over time. The derivative term makes the system consider how quickly it is adjusting to the error as it takes into account the instantaneous derivative. These are shown in equation 1 below.

$$u = K_p e + K_i \int_0^t e dt + K_d \frac{d}{dt} e \quad \textit{eq(1)}$$

Implementation

The implementation of our Cruise Control System consisted of using RIMS and a synchronous state machine. The state machine, as shown in figure 2, consisted of 4 states.

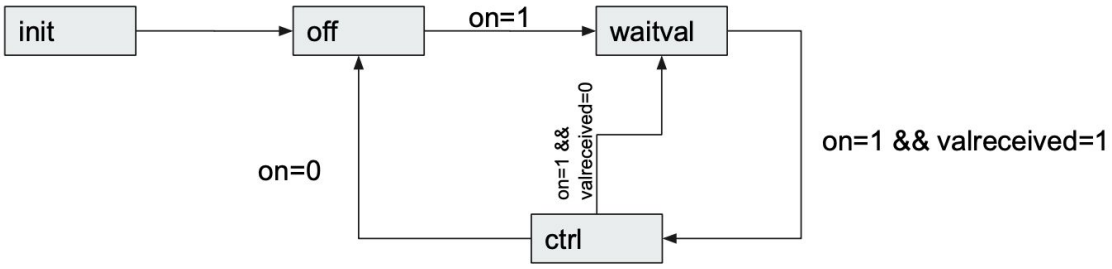

Figure 2. The simple cruise control state machine diagram

These 4 states are init, off, waitval, and ctrl. Init was the initial state when the program was started up. This state only set starting values to 0 and moved directly to the off state. In the off state, the state machine waited for the user to input a UART value. If the user entered a ‘g’ for go, the control would then set on to be 1 and move to the waitval state. In the waitval state, the state machine would again wait for the user to input a desired value. This state converts the ASCII characters to decimal values and

then determines if they are in between 0 and 15, the only values that we could use (more about this in discussion). If the waitval state received a value between 0 and 15, the valreceived variable would then be set to 1. This then transitions the state machine to the ctrl state.

The cntrl state is the most complicated and is the actual implementation of the PID controller. In the control state, it is first checked to see if the user has entered a UART value. If the user has, the state checks to see if it was a ‘c’ to change the value, or a ‘s’ to stop the controller. If it is a ‘c’, the state machine transitions back to the waitval state where it waits for another value to be entered by the user. If the ‘s’ is entered, the state is set to off and therefore brings the value back down to 0. Once the UART has been checked, if neither of these conditions are true, the state machine will start the control system process. Once the PID has a desired value, it calculates the difference between the desired and actual as an error. This error is fed through the PID equation that is displayed in equation 1. We determined that with $K_p = 9$, $K_i = 6$, and $K_d = 2$, we could get a reasonable response for the system as shown in the results section. The rest of the implementation stems from the actuator. The actuator, B in our case, is what the output of the system is and is used to calculate the actual value of the output. This output is then pushed as the next value, the next time the timer runs. This update cycle is the basis of a controller and allows us to continually check the difference between the desired and actual values. This feedback loop is the bench for what we built our cruise control system on.

Results

We achieved our goal of implementing our cruise control system with slight modifications from the original plan. Using a PID controller in RIMS we were able to simulate a cruise control system including the interaction with the user. Figure 3 shows an implementation of the response of the system to a desired value of 5.

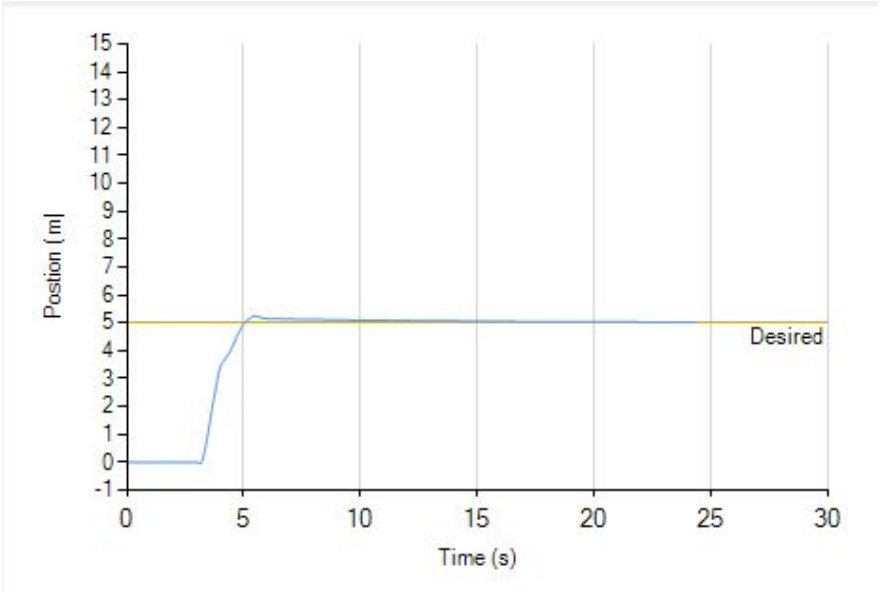

Figure 3. The response of the system to a desired value of 5

This output was exactly what we wanted when reacting to a desired value. An increase in speed without too much overshoot and a leveling out at the desired output. The odd result that we got occurred at the extremes of the limits. When the desired values were between 4 and 10, we got mainly graphs that looked as figure 3 does. When the desired output was outside of this value, we got outputs that looked as figure 4 does.

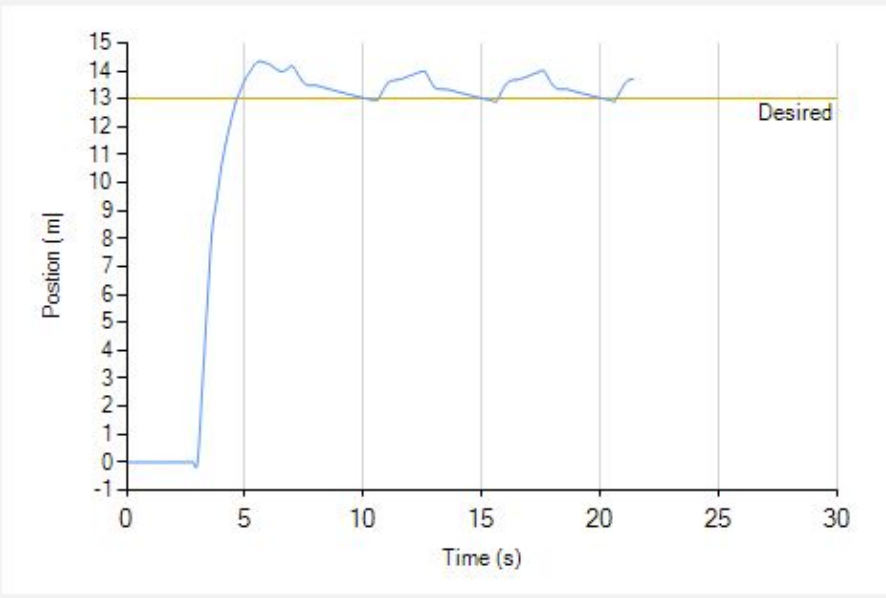

Figure 4. Output of control system with desired value at 13

To fix this issue, we decreased the integMax and increased integMin value. In doing this, we discovered that these values determined the peaks around 13 as shown in figure 4. With this adjustment, we got graphs as displayed in figure 5.

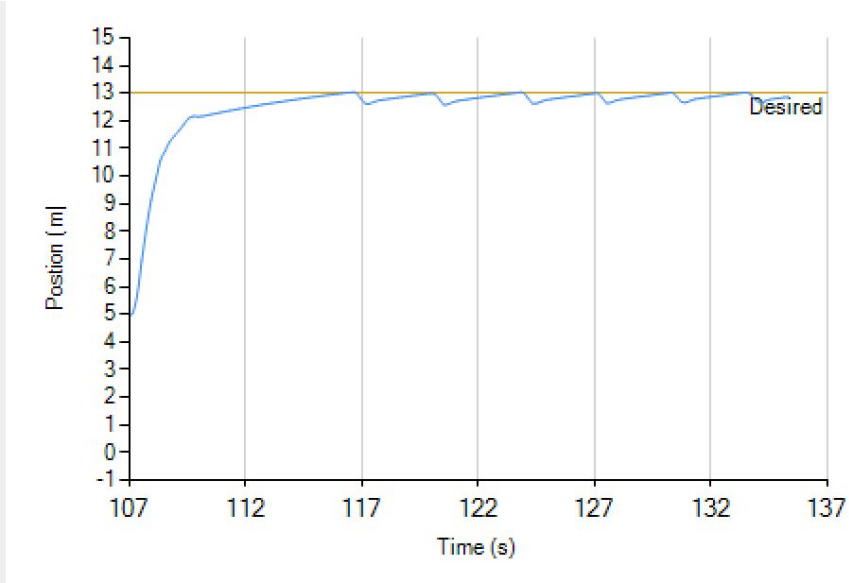

Figure 5. Setting to 13 with integMax and integMin implemented

In order to achieve the optimal output with reasonable edges this idea was required. Using our constant values and this slight adjustment, we were able to create a small

Discussion

In this project, we were able to create a functioning and interactive PID controller to simulate a cruise control system that interacts with its user. The interaction with the user allows the user to set a control value which will control the cruising speed.

In terms of problems encountered, we were unable to adjust for a plot reflecting the realistic speed values. However, we do believe that the graph we received in RIMS PID simulator will scale well. We were also unable to complete the goal of plotting the

graph using MATLAB which would resolve the issue of being unable to plot realistic speed values and the issue of scaling graph.

Conclusion

In conclusion, in this project, we were able to create a simple PID controller using RIMS to simulate a cruise control system with user interaction. In terms of future research, we should try to discover the implementation of the project in MATLAB or Simulink to help generate a more realistic graph of the cruise control system. We look forward to building a more realistic non simple PID controller that would allow us to include the accelerator as part of the system. Thus, enabling the user to learn how fast they are moving and accelerating once they switch on the cruise control system.

Appendices

A. Resources

Fig 1: [Control Systems - Quick Guide]

B. Code:

```
#include "rims.h"

/*Define user variables for this state machine here. No functions; make them
global.*/
#define Actuator B
short integMax = 1;
short integMin = -1;
short Desired;
short Actual;
short Error;
short Deriv;
short Integ;
short calcAct;
short ActualPrev;

short Kp = 9;
short Ki = 6;
short Kd = 2;

int on = 0;
```

```
int val_recieved = 0;
int num = 0;
int num_1 = 0;
int num_2 = 0;

unsigned char SM1_Clk;
void TimerISR() {
   SM1_Clk = 1;
}
volatile unsigned char RxFlag = 0;
void RxISR(){
    RxFlag = 1;
}

enum SM1_States { Init, Ctrl, Off, WaitVal} SM1_State;

TickFct_OnOff_Ctrl() {
   switch(SM1_State) { // Transition
      case Init:
         SM1_State = Off;
         break;

      case Off:
         if (!on){
            SM1_State = Off;
         }
         else if (on){
             printf("Cruise Control On. Enter a Speed.\n");
             SM1_State = WaitVal;
         }
         break;

      case WaitVal:
         if(!val_recieved){
             SM1_State = WaitVal;
         }
         else if (val_recieved){
             SM1_State = Ctrl;
         }
         break;

      case Ctrl:
         if(on ==1 && val_recieved == 1){
            SM1_State = Ctrl;
         }
         else if(on ==1 && val_recieved == 0){
             SM1_State = WaitVal;
             printf("Enter a new desired speed.\n");
         }
         else{
```

```
            SM1_State = Off;
        }
        break;
    default:
        SM1_State = Off;
} // Transitions

switch(SM1_State) { // State actions
    case Init:
        Actuator = 0;
        ActualPrev = 0;
        Integ = 0;
        break;

    case Off:
        printf("Cruise Control Off\n");
        num = 0;
        Actuator = 0;
        ActualPrev = 0;
        Integ = 0;
        while(!RxFlag);
        RxFlag = 0;
        if(R == 0x67){
            on = 1;
        }
        break;

    case WaitVal:
        while(!RxFlag);
        RxFlag = 0;
        num_1 = R-0x30;
        while(!RxFlag);
        RxFlag = 0;
        num_2 = R-0x30;
        num = num_1*10 + num_2;

        if(num < 15 && num > 0){
          num = (0x0F & num);
          val_recieved = 1;
          printf("Setting Speed to %dmph\n\n",num);
        }
        else{
            printf("Please Enter Speed In Range\n");
        }
        break;


    case Ctrl:
        if(RxFlag == 1){
           if(R == 0x73){
```

```
                on = 0;
            }
            else if(R == 0x63){
                val_recieved = 0;
            }
            RxFlag = 0;
        }
        else{
            Desired = (num);
            Actual = ((A & 0xF0)>>4);

            // Calculate proportional error
            Error = Desired - Actual;

            // Calculate integral
            Integ += Error;
            if(Integ > integMax)Integ=integMax;
            if(Integ < integMin)Integ=integMin;

            // Calculate derivative
            Deriv = Actual - ActualPrev;

            // Calculate actuator output
            calcAct = Kp*Error + Ki*Integ - Kd*Deriv;
            if(calcAct < 0){
                calcAct = 0;
            }

            Actuator = calcAct;
            ActualPrev = Actual;
        }
        break;

      default: // ADD default behaviour below
      break;
   } // State actions

}

int main() {
   const unsigned int periodOnOff_Ctrl = 500;
   TimerSet(periodOnOff_Ctrl);
   TimerOn();
   UARTOn();

   SM1_State = Init; // Initial state
   B = 0; // Init outputs

   while(1){
      TickFct_OnOff_Ctrl();
```

```
        while(!SM1_Clk);
        SM1_Clk = 0;
    }
}
```