# Autonomous Mobile Robots

# Lecture 06: Feedback Control

Lecture is based on material from Robotic Explorations: A Hands-on Introduction to Engineering, Fred Martin, Prentice Hall, 2001.

## Outline

- Serial Communication and Data Collection
- Simple Feedback Control
- Proportional-Derivative Control
- PID Control from Matlab™ Tutorial

Matlab is trademark of MathWorks, Inc.

## Homework #6

- **Serial Communication and Data Collection:** Read Appendix C of Robotic Explorations (textbook)
- **Control:** Read Chapter 5 of Robotic Explorations (textbook)
- **Control History:** "A Brief History of Feedback Control" from Chapter 1: Introduction to Modern Control Theory, in: F.L. Lewis, Applied Optimal Control and Estimation, Prentice-Hall, 1992. Web page: http://www.theorem.net/lewis1.html
- **Matlab**™: Become familiar with Matlab for plotting and PID control: Read the whole Matlab Basics and PID Tutorial on the Control Tutorials for Matlab web site: http://www.engin.umich.edu/group/ctm/




## Serial Communication and Data Collection

- How to collect data on the Handy Board and upload it to a host computer for processing, using the IC environment.
- There are at least two ways upload sensor data from the Handy Board:

- **Sensor data is printed to serial line in real time**

  Ideal for when data does not need to be sampled too quickly, since sampling rate is limited by the serial transfer speed

  or when the amount of data being collected is so great as to exceed the Handy Board's memory

- **Sensor data is collected & stored in HB's memory and later uploaded to serial line**

  Best for capturing a short burst of rapidly changing data

  or when it is desired to leave the Handy Board in a remote location for the data collection process

# Serial Communication and Data Collection

## Serial Line Interaction

- IC interacts with HB via low-level protocol that allows it to do things like write to/from HB's memory while HB is executing IC programs
  - The runtime system resident on HB responds to this protocol at all times
  - Because of this, characters cannot simply be sent to the HB's serial line
- **Solution:** temporarily disable runtime system's responses to serial activity
  - From the host computer side, it will appear to IC that HB is not connected, because it will no longer respond to the built-in serial protocol
- Then, writing to the serial line is done by interacting with built-in 6811 serial port registers
  - *Serial Communications Data Register* (SCDR) is located at address 0x102f
    - If data is stored to this register, the 6811 transmits it as serial output; when a serial character is received, it is retrieved by reading from the same register
  - *Serial Communications Status Register* (SCSR) is located at address 0x102e
    - Bits in this register indicate when the serial port is busy (e.g., whether it is in the middle of receiving or transmitting a character)
- IC library file: **serialio.c**
  - Wrapper functions for interacting with serial port




# Serial Communication and Data Collection

## Connecting to a Terminal Program

- *Terminal emulator program:* used for receiving serially-transmitted data
- Test program for establishing a connection between the HB and a terminal emulator on the host computer
- **Load serxmit.c serialio.c**
- **`disable_pcode serial():`** so that HB does not interpret any accidental characters that might be sent from the host computer
- Infinite loop:
  - Prints a message to the LCD screen telling the user to press the Start button
  - Calls the library function **`start_press()`**, which waits for the Start button to be pressed
  - Transmits the 96 printable characters of the ASCII 1 character set, beginning with code 32, a space, and ending with code 127, a tilde
- To restart HB in normal mode: hold down Start button while turning it on

```
/* serxmit.c
Each time start button is
pressed,  transmits the 96-
character ASCII  set */
void main()
{
     int i;

disable_pcode_serial();
       while (1) {
        printf("Press Start
button to begin\n");
        start_press();

printf("Transmitting...\n");
        for (i= 32; i< 128;
i++)
        serial_putchar(i);
       }
}
```

# Serial Communication and Data Collection

## Printing to Serial Line

- IC library file **printdec.c** provides **printdec()**, which takes an integer as input and prints its value as a decimal number over the serial line
- Example: **analogpr.c** program to continuously print value of analog sensor 0 to serial line
  - After calling **printdec()** to print the sensor value, the program outputs the values 10 and 13 to the serial line
  - This is done using serial **putchar()** so that the data is sent as control characters. When interpreted by the terminal emulator, the 10 causes a line feed and the 13 causes a carriage return.
  - **msleep()** function in the inner loop of the display routine slows down the rate at which the HB broadcasts the sensor data to allow terminal emulator program to keep up on its screen display.
  - Sensor data is continuously displayed on the host computer screen

```
/* analogpr.c
requires printdec.c,
serialio.c */
void main()
{

disable_pcode_serial();
     while (1) {

printdec(analog(0));

serial_putchar(10);

serial_putchar(13);
/* wait 0.1 sec between
each print */
        msleep(100L);
     }
}
```



# Serial Communication and Data Collection

## Capturing Data

- For quickly changing data, the final piece of the puzzle is storing sensor data in the HB's memory for later printing to the serial line
- This allows a much faster capture rate since the speed is limited only by the speed of IC, rather than the relatively slow serial communications rate
- **datacoll.c:** IC program for capturing data
  - **data[]** array of 1000 elements
  - **main()** allows user to trigger data-collection and data-dump modes by pressing the Start button

```
/* datacoll.c
requires printdec.c,
serialio.c */
int SAMPLES=1000;
char data[1000];
void main()
{
     disable_pcode_serial();
     printf("press Start to
collect data\n");
     start_press();
     collect_data();
     beep();
     printf("press Start to
dump data\n");
     start_press();
    dump_data();
     beep();
     printf("done.\n");
}
```

## Serial Communication and Data Collection

### Capturing Data

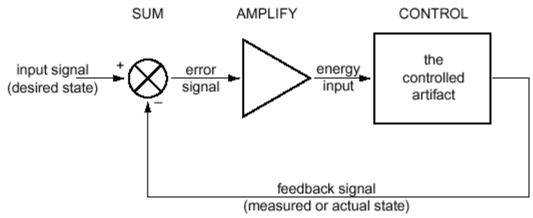
- **collect_data()** iterates through the elements of the array, storing a successive data sample in each one (takes 2 sec - 500 samples/sec - may slow down)
- **dump_data()** outputs the data stored in the array to the serial line, using the line-feed/ carriage-return technique
- Save data; load into spreadsheet program for graphing and analysis

```
void collect_data()
{
      int i;
      for (i= 0; i< SAMPLES;
i++) {
      data[i]= analog(0);
/* to slow down capture rate,
add msleep here */
      }
}
void dump_data()
{
      int i;
      for (i= 0; i< SAMPLES;
i++) {
      printdec(data[i]);
      serial_putchar(10); /*
line feed */
      serial_putchar(13); /*
carriage return */
      }
}
```




## Simple Feedback Control

- Goal: make error signal go to zero
- Example: Home Heating system
  - Thermostat measures room temperature, determines if air temperature is too cold or too hot, turns on/off furnace
  - Thermostat setting adjusted manually
  - Furnace is binary heat source: on/off
  - Delay in air heating up and cooling off
  - Air temperature oscillates around setpoint

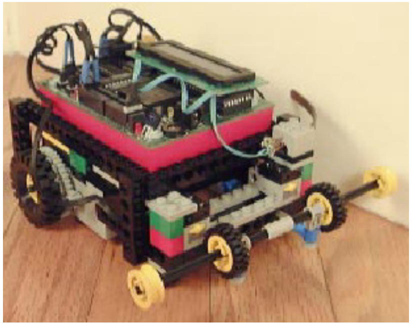



- Compare actual room temperature with desired temperature, which gives error signal
- Furnace is amplifier
- Artifact/Plant is house, which receives energy from the furnace
- Feedback is temperature measurement

# Simple Feedback Control

## Wall Following

- HandyBug with bend sensor or reflective IR sensor
- HandyBug turns towards wall if distance sensor indicates too far away; turns away from wall if too close
- Single threshold for "too far" and "too close" = **goal** variable
- Keep track of distance from wall during task = **data[]** vector
- Write data to serial line = **dump_data** routine
- Manually set wall-sensor threshold = **calibrate** routine
- Requires **serialio.c, printdec.c**



*HandyBug* design is equipped with a **bend sensor** "outrigger," enabling it to detect its distance from the wall. The closer to the wall, more the sensor is flexed. This increases its resistance, and a greater value is detected by *HandyBug*'s sensor circuitry.





# Simple Feedback Control

## Wall Following

```
/* wallfoll.c: threshold-based
wall follower
with data collection */
int LEFT_MOTOR= 0; /* motor and
sensor ports */
int RIGHT_MOTOR= 3;
int LEFT_WALL= 0;
persistent int goal;       /* wall
conditions */
```

```
void calibrate() {
while (1) {
     int wall= analog(LEFT_WALL);
     printf("goal is %d; wall is
%d\n", goal, wall);
     if (start_button()) {
          goal= wall; beep();
          }
     if (stop_button()) {
          printf("Set goal to %d\n",
goal);
          beep(); sleep(0.5); break;
     }
     msleep(50L); /* give a pause
for the display */
}
}
```

```
void main() {
     calibrate();
     ix= 0;
     while (1) {
       int wall= analog(LEFT_WALL);
       printf("goal is %d; wall is %d\n",
       if (wall < goal) left(); /* too far
       else right(); /* turn away from wall
       data[ix++]= wall; /* take data sampl
       msleep(100L); /* 10 iterations per s
     }
}
```

```
void left() {
motor(RIGHT_MOTOR, 100);
motor(LEFT_MOTOR, 0);
}

void right() {
motor(LEFT_MOTOR, 100);
motor(RIGHT_MOTOR, 0);
}
```

Hard turns

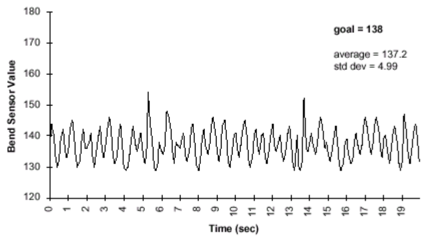
# Simple Feedback Control

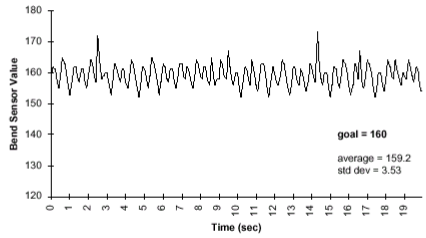
## Wall Following

```
void dump_data() {
int i;
disable_pcode_serial();
printf("Press START to send
data...\n");
start_press();
for (i=0; i< ix; i++) {
     printdec(data[i]);
     serial_putchar(10);
serial_putchar(13);
     }
printf("Done sending data.\n");
beep();
}
```

Results with bend sensor:

- HandyBug oscillates around setpoint goal value
- Never goes straight





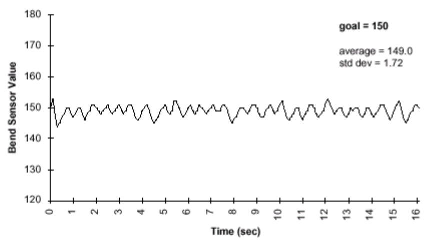

# Simple Feedback Control

## Wall Following

- Gentle Turning Algorithm:
  - Swings less abrupt
  - HandyBug completes run in 16 sec (vs. 19 sec in hard turn version) for same length course
- Exercise: try a 3-state algorithm: use two thresholds that will allow HandyBug to either turn left, go straight or turn right

```
void left() {
motor(RIGHT_MOTOR, 100);
motor(LEFT_MOTOR, 50);
}

void right() {
motor(LEFT_MOTOR, 100);
motor(RIGHT_MOTOR, 50);
}
```



# Proportional-Derivative Control

## Proportional Control

• Control algorithm generates a stronger response the farther away the system is from the goal state — response of control algorithm is **proportional to amount of error**

• **Test system:** experiment with proportional control and proportional-derivative control

– Control rotational position of LEGO wheel

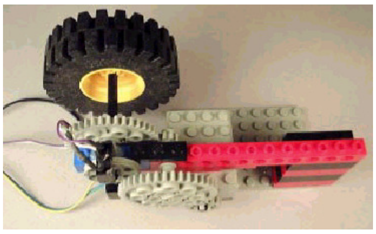
– Will vary power to motor, i.e., motor speed

• Load shaft encoder driver:

**load qencdr10.icb**

• Test shaft encoder:

```
encoder10_counts= 0;
while (1) { printf("%d\n",
encoder10_counts); msleep(50L);}
```



The proportional-derivative control test system includes a dc motor driving a two-stage gear reduction, and a large LEGO wheel which gives the system a fair bit of momentum (load on the system). At the middle stage of the gearing, a quadrature-based shaft encoder keeps track of the shaft position.



# Proportional-Derivative Control

## Proportional Control

• Turn on motor, wait, turn it off, and then print out the encoder reading (6-hole pulley wheel gives 24 counts/revolution on encoder):

```
{encoder10_counts=0; motor(0, 100); msleep(100L); off(0); msleep(500L);
printf("%d\n", encoder10_counts);}
```

• **Proportional Error Controller**: Set the encoder counter to 0, and then write an infinite loop to repeatedly set the **motor speed** to the difference between a desired position and actual position:

```
{encoder10_counts= 0; while (1) {motor(0, 100 - encoder10_counts);}}
```

• When the program starts to run, the difference between the desired position (**setpoint =100**) and the actual position (0) is 100, so the motor turns on full speed, driving the wheel toward the desired position. As it starts going, the error becomes progressively smaller. When it's halfway, at position 50, the error is only 50, so at that point the motor goes at 50% of full power. When it arrives at the intended position of 100, the error is zero, and the motor is off.

• **Proportional Gain (ratio between error and power):** Instead of a one-to-one ratio between error counts and motor power percentage, modify the controller so it multiplies the error value by 5:

```
{encoder10_counts= 0; while (1) {motor(0, 5 * (100 -encoder10_counts));}}
```

• Response should feel much "snappier." The wheel should reach the setpoint position faster, and it should resist being turned away from it much more aggressively.

# Proportional-Derivative Control

## Proportional Control

• **Overshoot:** when the system goes beyond its setpoint and has to change direction before stabilizing on it. For the test system of a LEGO wheel turning in space, overshoot doesn't seem to be much of a problem, but imagine if the system were a robot arm moving to a particular position. If it went beyond the position on its way getting there, it could have collided with some object just beyond the setpoint position.

• **Oscillations:** are related to overshoot. After the system goes beyond its setpoint, when it corrects and drives the other way it can "overshoot" in the other direction as well. Then the phenomenon repeats and one sees the system going back and forth around the setpoint, in a nervous or jittery manner.

• Serious concern to system designer: minimize both overshoot and oscillation, but provide adequate system response to changes in setpoints.

```
void collect_data()
{
     int i, power, counts;
     for (i= 0; i< SAMPLES;
        counts= encoder10_c
        power= pgain * (0 -
        motor(0, power);
        data[i++]= counts;
        data[i++]= power;
     }
}
```

• **pgain** is proportional error gain, units are conversion between error counts and percentage of full power, e.g., gain of 10 means 10-count error results in a full power action.
• **Power** represents % full power calculated by control function
• **Position** is actual encoder counts.

# Proportional-Derivative Control

## Proportional Control

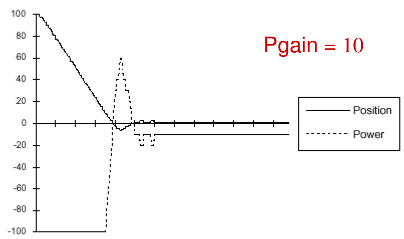
• Data taken every 0.2 seconds of real time. System is driven from 100 counts to 0 counts

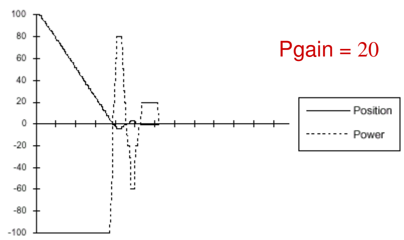
• **Pgain=10:** Full-power command as wheel heads toward setpoint.

  – When position within 10 counts of zero, power command began to fall off.
  – System overshot the zero point, and had to turn around
  – **Offset Error:** System did not stabilize at the goal. From around the 1.2 second mark onward, the position was fixed at a count of 1. This generated a power command of 10%, which was too small to activate the motor.

• **Pgain=20:** should ameliorate the offset problem, since the same static error will result in a higher power command

  – Offset error is solved
  – Another problem: the beginnings of an oscillation. Now the system overshoots three times—twice beyond the setpoint and once before it.



# Proportional-Derivative Control

## Proportional Control

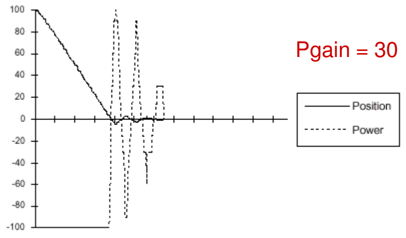
• **Pgain=30:** Oscillation problem is more pronounced; there are a total of five oscillatory swings

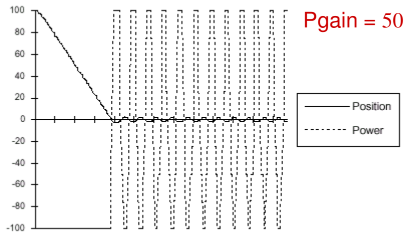
• **Pgain=50:** Oscillation behavior has taken over

– System moves to within a slight distance from the setpoint, but cannot stabilize at the setpoint

– Even a small error generates a power command that moves the system across the setpoint, resulting in a power command in the opposite direction

– While the position error is small on the graph, the power command swings are quite evident.






# Proportional-Derivative Control

## Proportional-Derivative Control

• **Problem:** Simply cranking up the pgain does not get the system to perform better

– Motor drives output wheel to its position faster, but still overshoots/oscillates at higher gains

– Driving at full power is appropriate when the system is far away from its setpoint

– When pgain is high, then even a slight deviation from the setpoint causes a large power command

• **Solution:** Correct for the momentum of the system as it moves toward the setpoint

– Momentum is mass times velocity; therefore, momentum is directly proportional to velocity

– Correct for the velocity when the system nears its setpoint by ***subtracting*** an amount from the power equation based on the velocity of the system

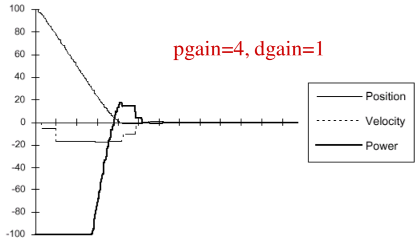
```
void collect_data()  {
int i, power, counts, velocity;
for (i= 0; i< SAMPLES;) {
     counts= encoder10_counts;
     velocity=
encoder10_velocity;
     power= pgain * (0 - counts)
- dgain * velocity;
     motor(0, power);
     data[i++]= counts;
     data[i++]= velocity;
     data[i++]= power;
     }
}
```

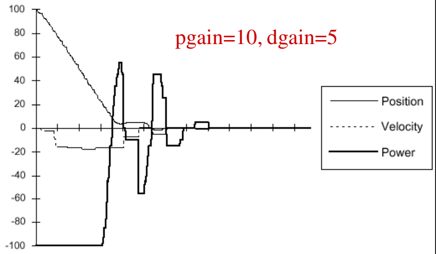
Power command is now combination between proportion of error and velocity of system

# Proportional-Derivative Control

## Proportional-Derivative Control

- **Pgain=4, Dgain=1:** Overshoot is minimized, and there is no oscillatory behavior at all.
- **Pgain=10, Dgain=5:** unstable; dgain is too large
  - Position graph: controller "puts on the brakes" too hard and the system stops moving before the destination setpoint (between the 0.8 and 1.0 second mark)
  - When the velocity hits zero, the proportional gain kicks in again and the system corrects
- **PD Control** is used extensively in industrial process control
  - Combination of varying the power input when the system is far away from the setpoint, and correcting for the momentum of the system as it approaches the setpoint is quite effective







# Proportional-Derivative Control

## Proportional-Derivative Control

- **Discrete Sampling Error:** Note discrete nature of the velocity graph, which is due to the fact that the encoder software calculates the velocity measure infrequently—every 0.15 seconds, as indicated by the steps in the velocity curve. (encoder driver takes difference between last encoder reading and current one to determine velocity) Since the testbed LEGO geartrain is overall a rather low-performance system, this is probably inconsequential, but this error is endemic to digital control systems and is a topic of much concern in control theory.
- Classical control theory is all about analyzing the response of systems. By modeling mass of the system being controlled, power flow into the system, load on the system, and other characteristics, it is possible to determine optimal values for the gain factors.
- **Self-tuning** or **adaptive controllers:** dynamically adjust gain parameters while system is in operation. This allows a controller to compensate for changing factors external to the system. Better than standard P-D controller.
- **PID control:** "I" stands for **integral**; an integral term can correct for steady-state errors like in the first example, where the system came to rest a few counts away from the setpoint. By integrating this error over time, the controller can deliver a "kick" to drive the system to the setpoint.