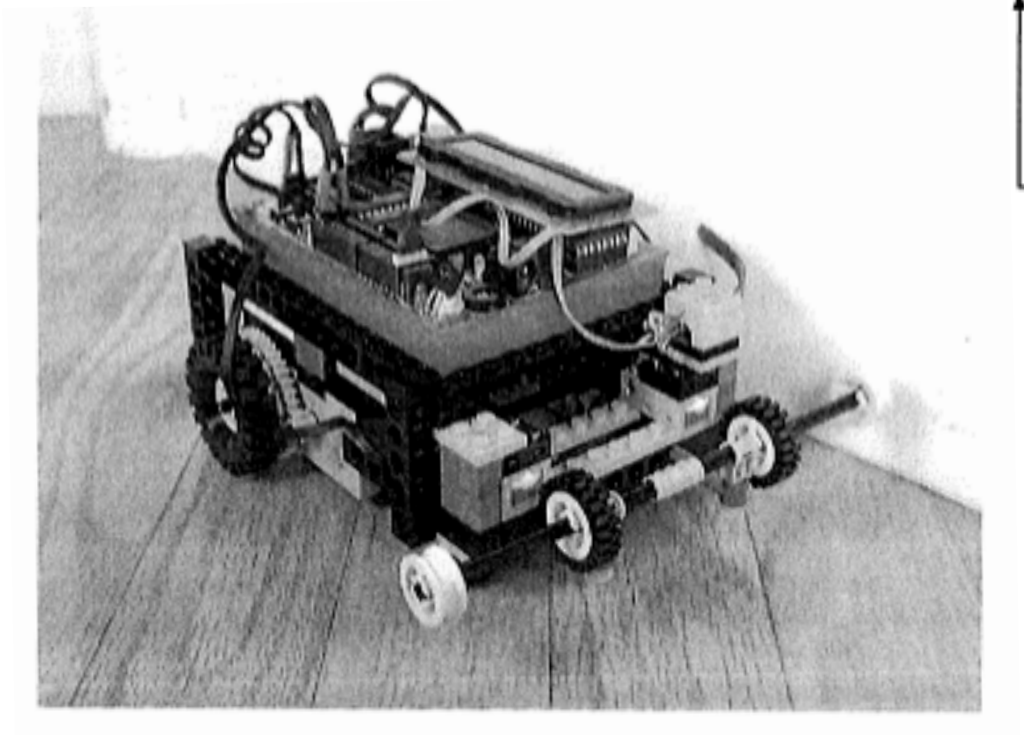
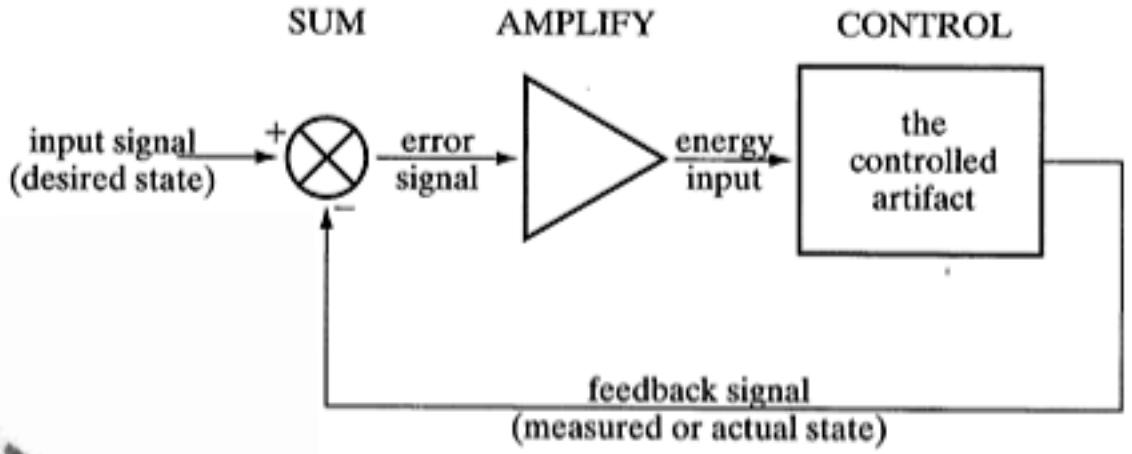
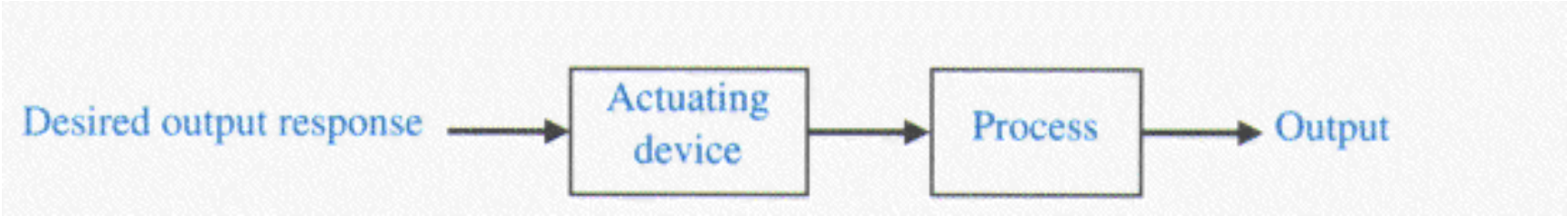
# Control Systems

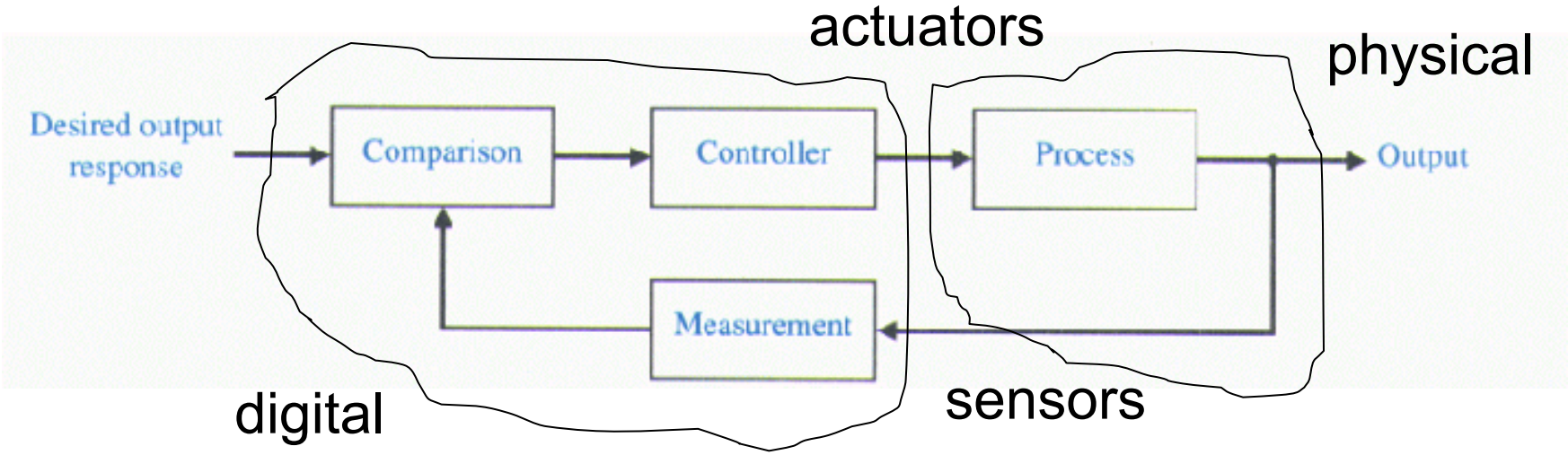


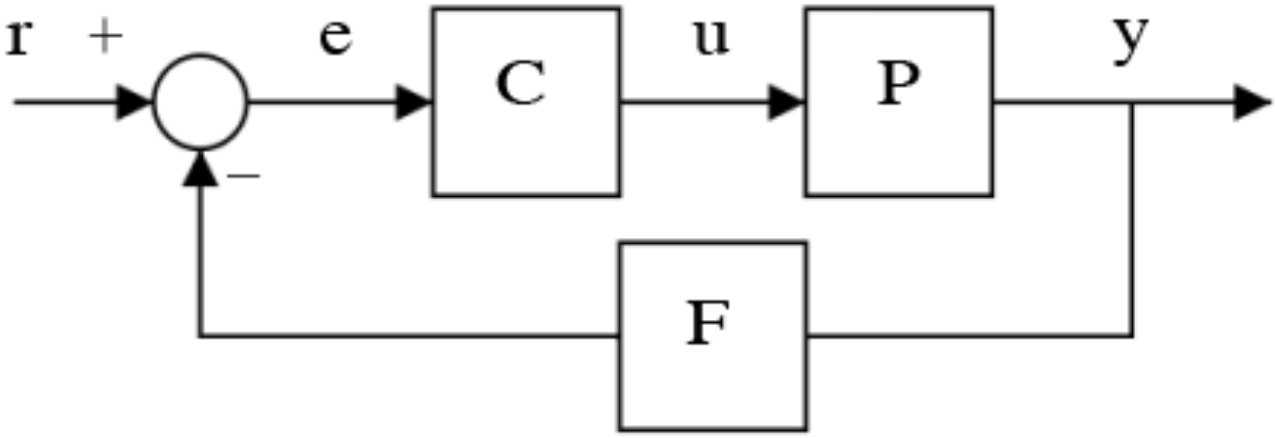


## Open loop control

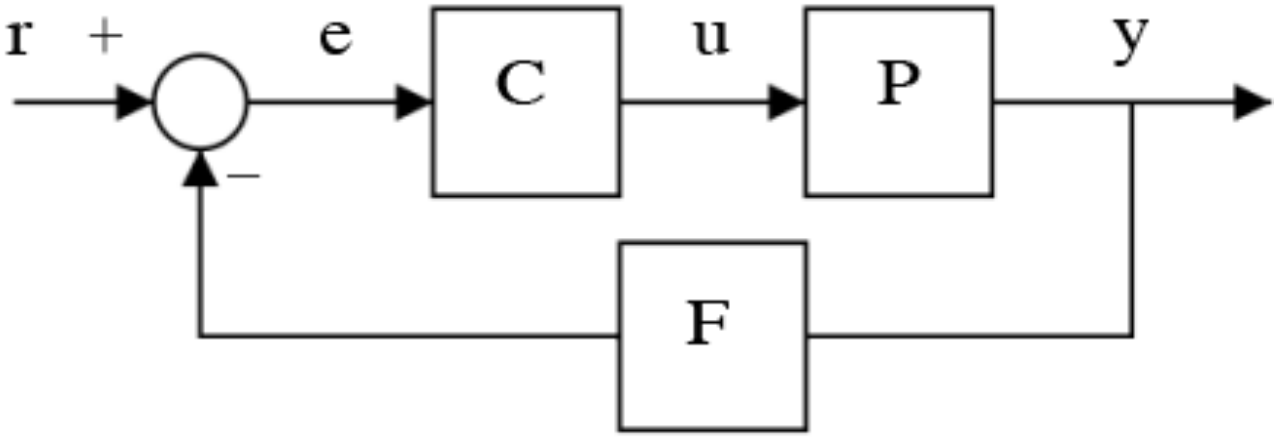



## Closed loop control





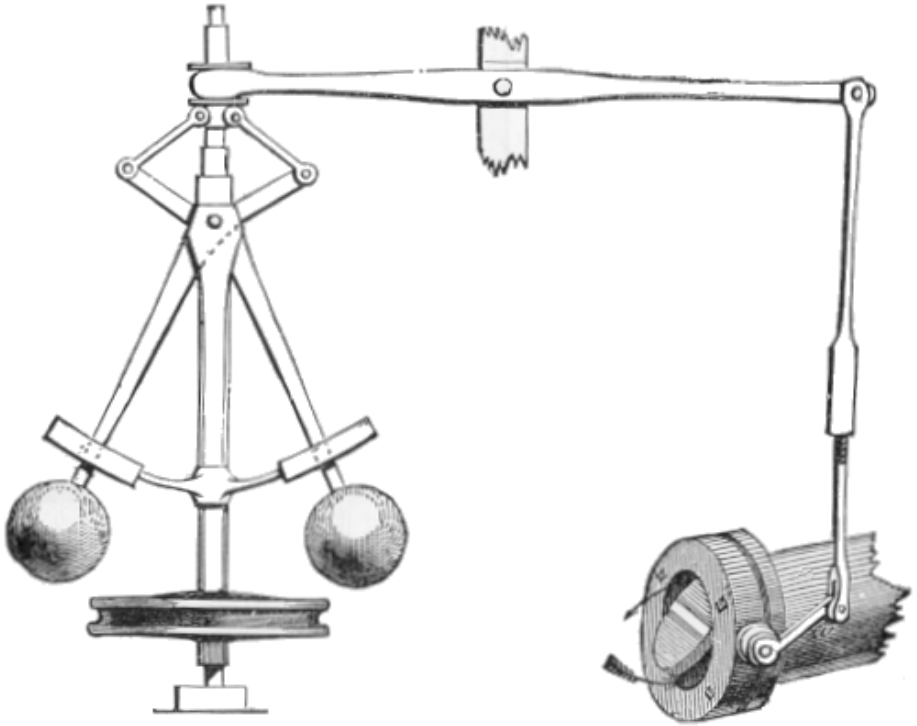

| | |
|---|---|
| r | reference value |
| e | error |
| C | controller |
| u | output from controller to control P |
| P | plant, controlled system |
| y | state of P |
| F | feedback to controller |

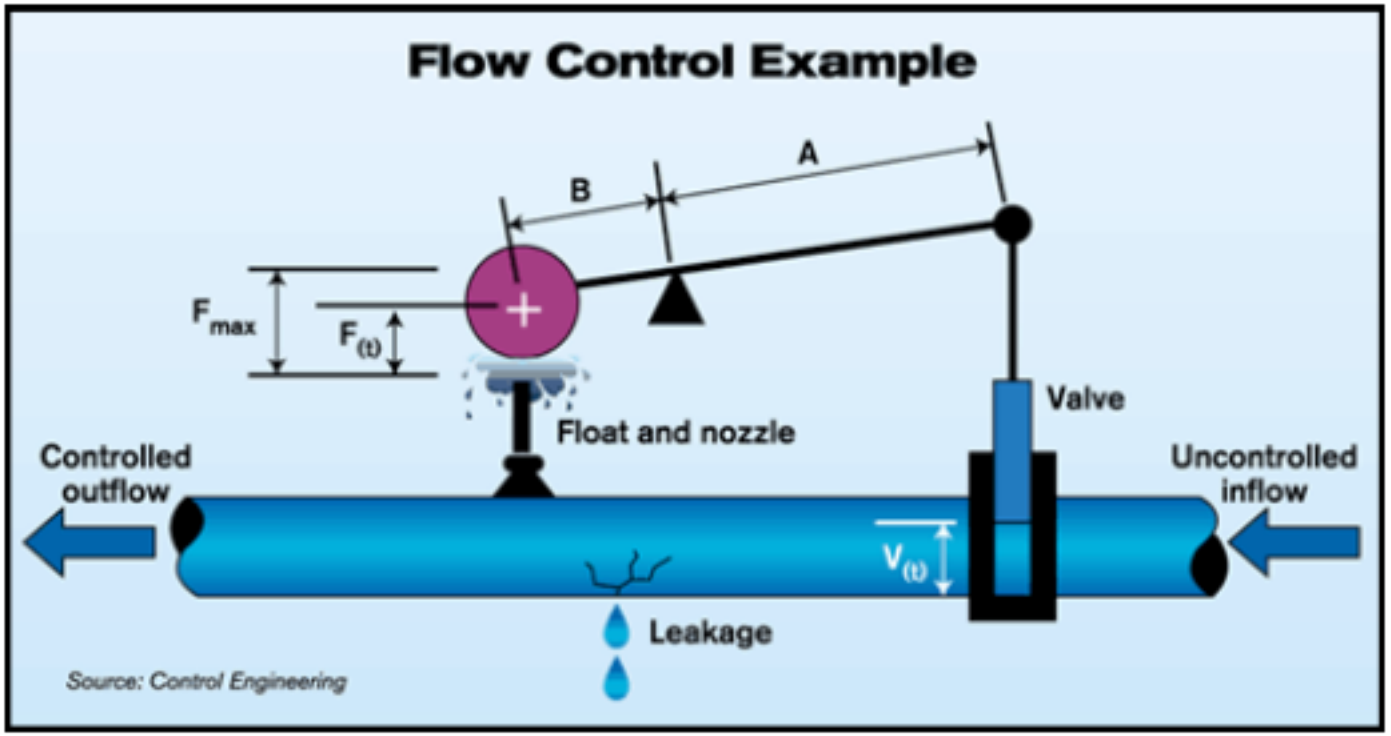


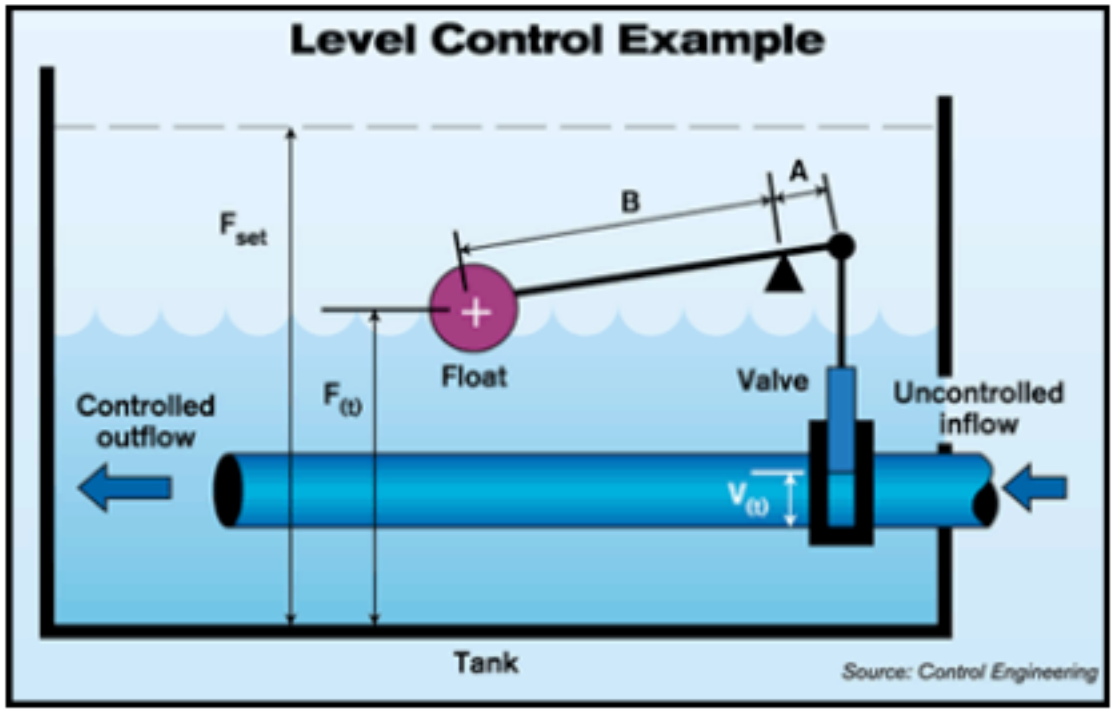

| | |
|---|---|
| r | reference value |
| e | error |
| C | controller |
| u | output from controller to control P |
| P | plant, controlled system |
| y | state of P |
| F | feedback to controller |

1. Mechanical
2. Electrical, analog
3. Digital

A centrifugal governor is a specific type of governor that controls the speed of an engine by regulating the amount of fuel (or working fluid) admitted, so as to maintain a near constant speed whatever the load or fuel supply conditions. It uses the principle of **proportional control**.




Flow Control Example
A
B
F max
F (t)
Valve
Float and nozzle
Controlled outflow
Uncontrolled inflow
V (t)
Leakage
Source: Control Engineering


Level Control Example
$F_{set}$
B
A
Float
$F_{(t)}$
Valve
Controlled outflow
Uncontrolled inflow
$V_{(t)}$
Tank
Source: Control Engineering

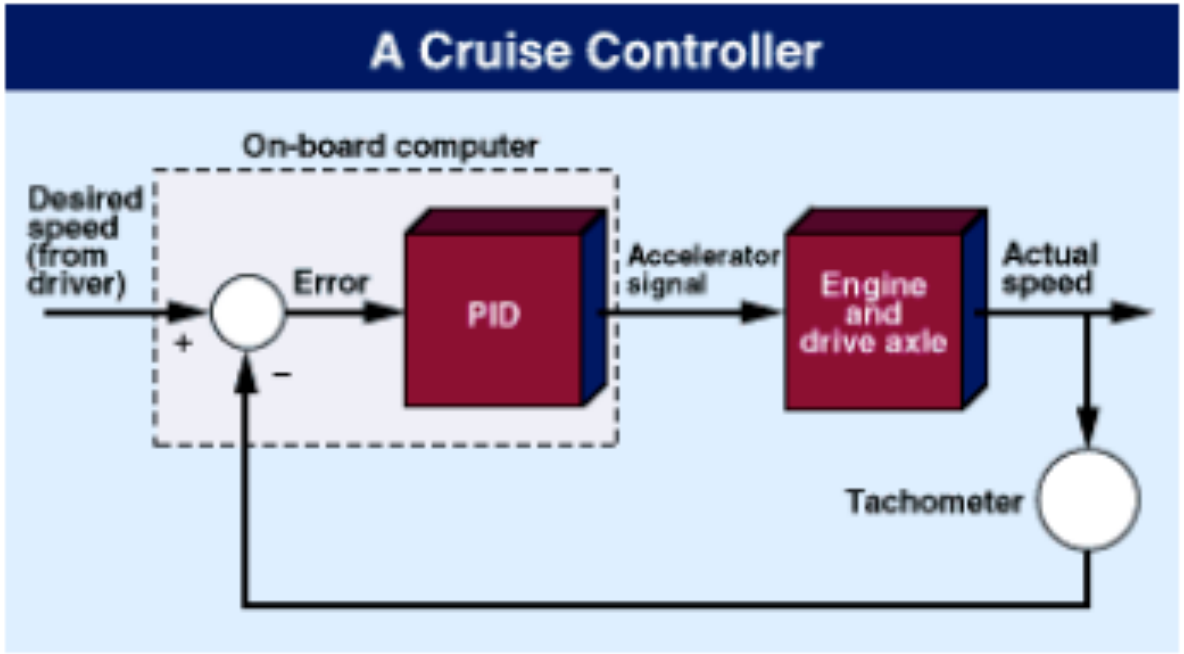
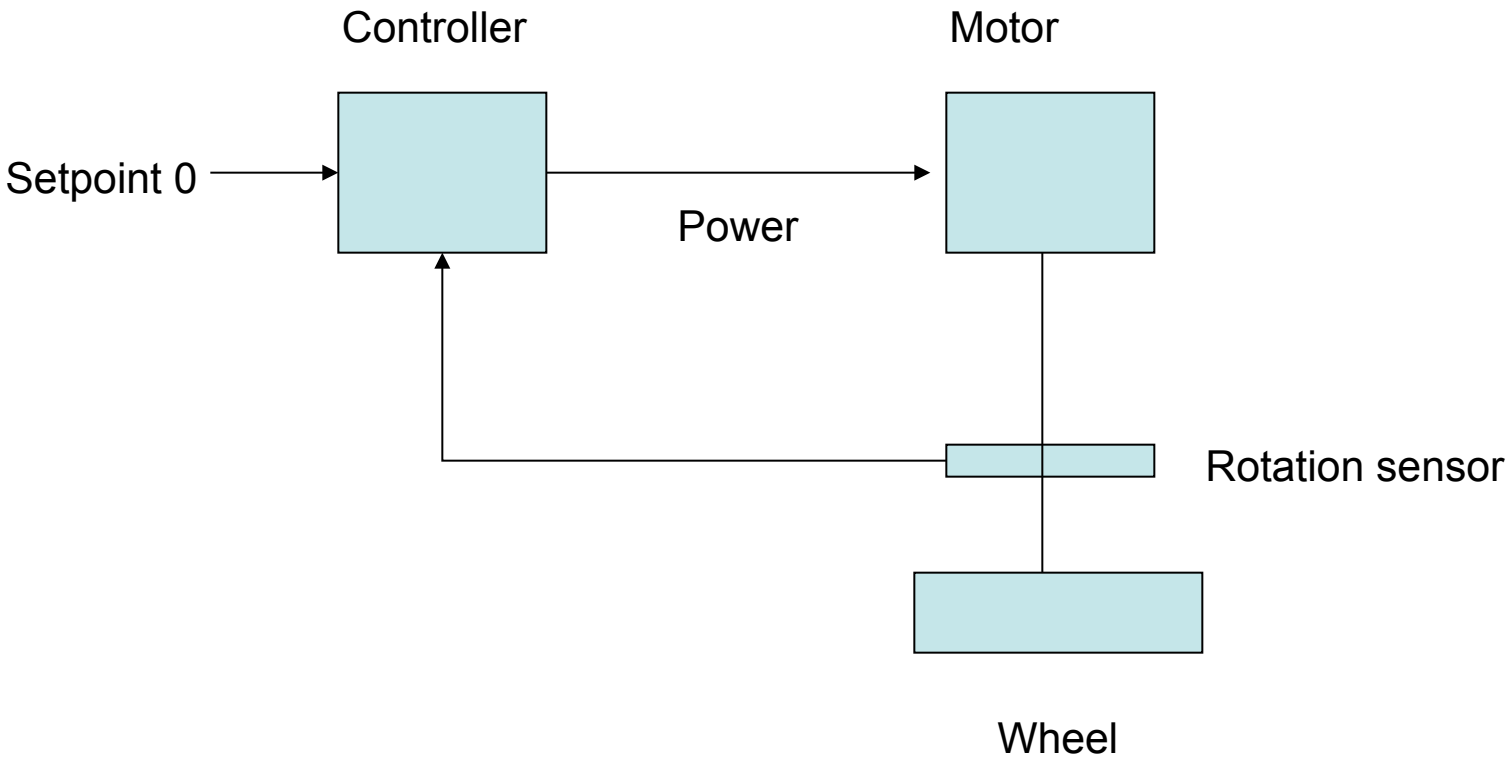
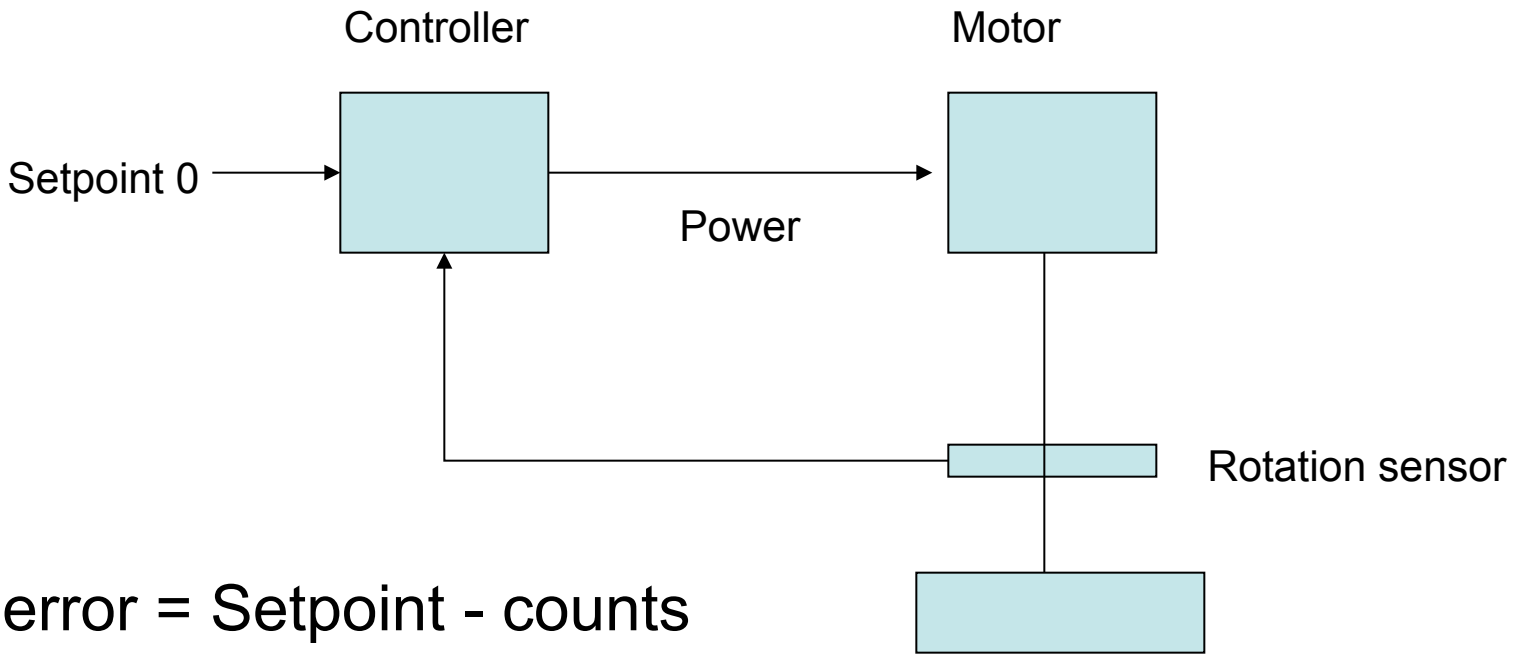



**Fig. 2: A familiar real-world example of feedback control can be found in the "cruise control" feature common in many automobiles.**


Controller
Motor
Setpoint 0
Power
Rotation sensor
Wheel




error = Setpoint - counts

Power = Pgain * error

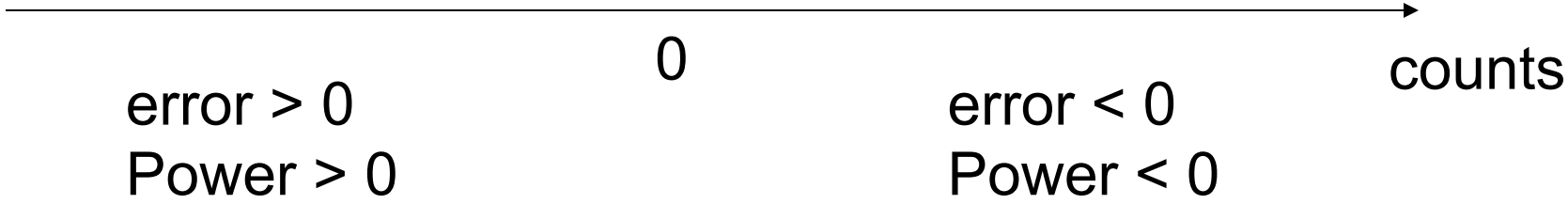


**FIGURE 5.10** Proportional Controller, $P_{gain} = 10$

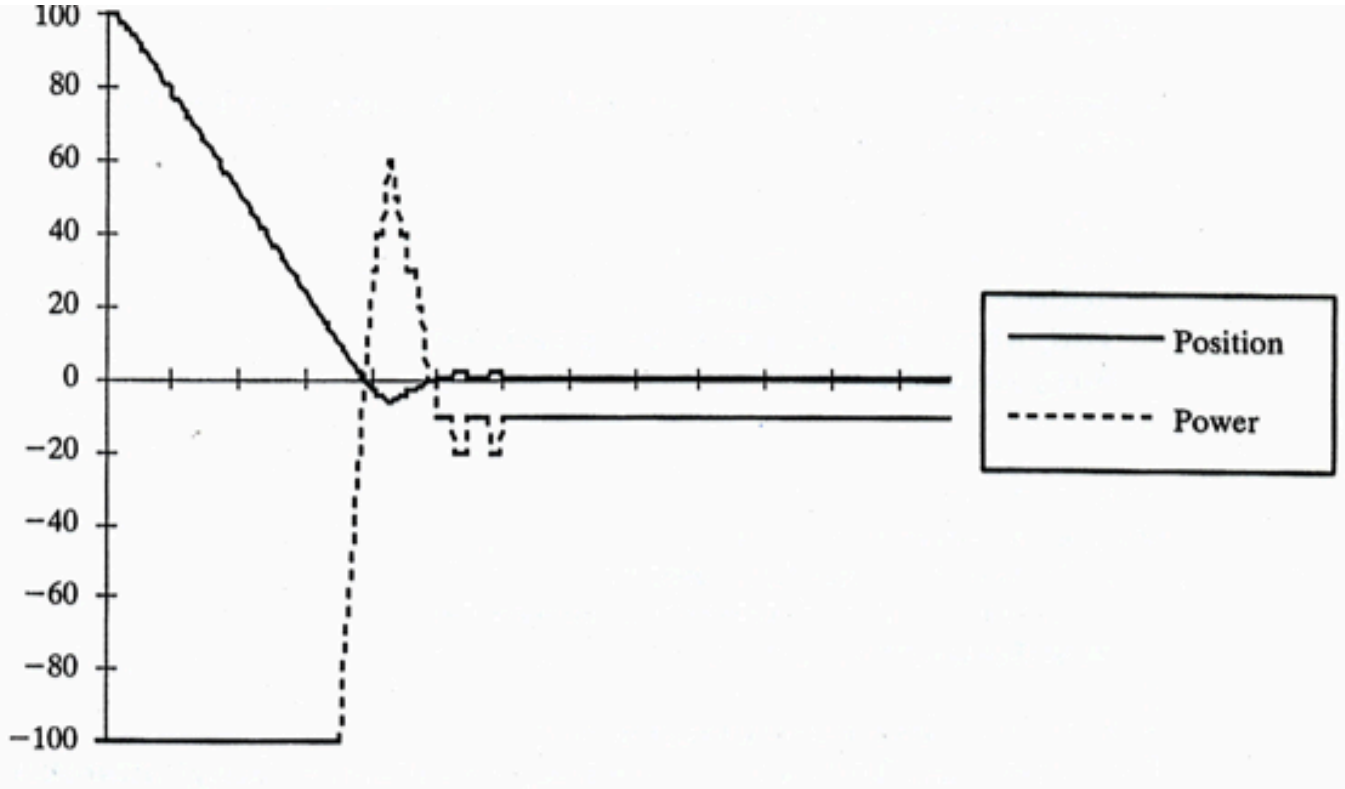
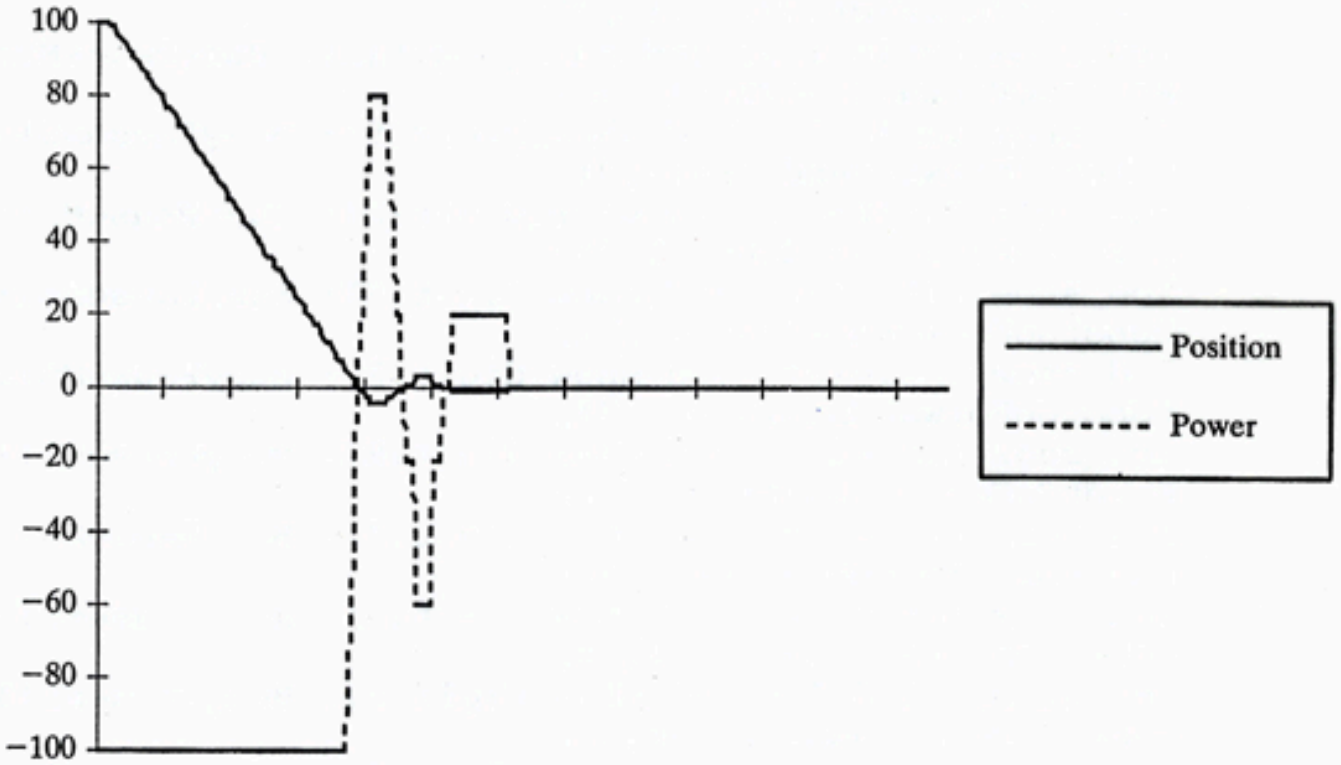


**FIGURE 5.11** Proportional Controller, $P_{gain} = 20$



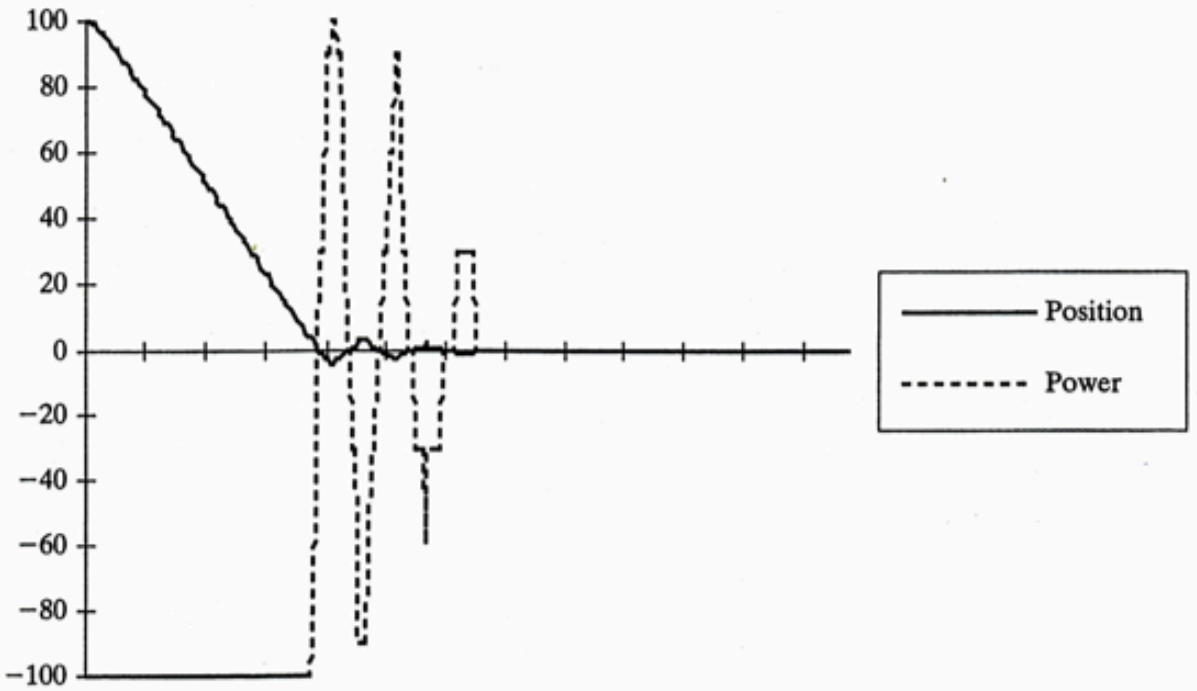



**FIGURE 5.12** Proportional Controller, $P_{gain} = 30$

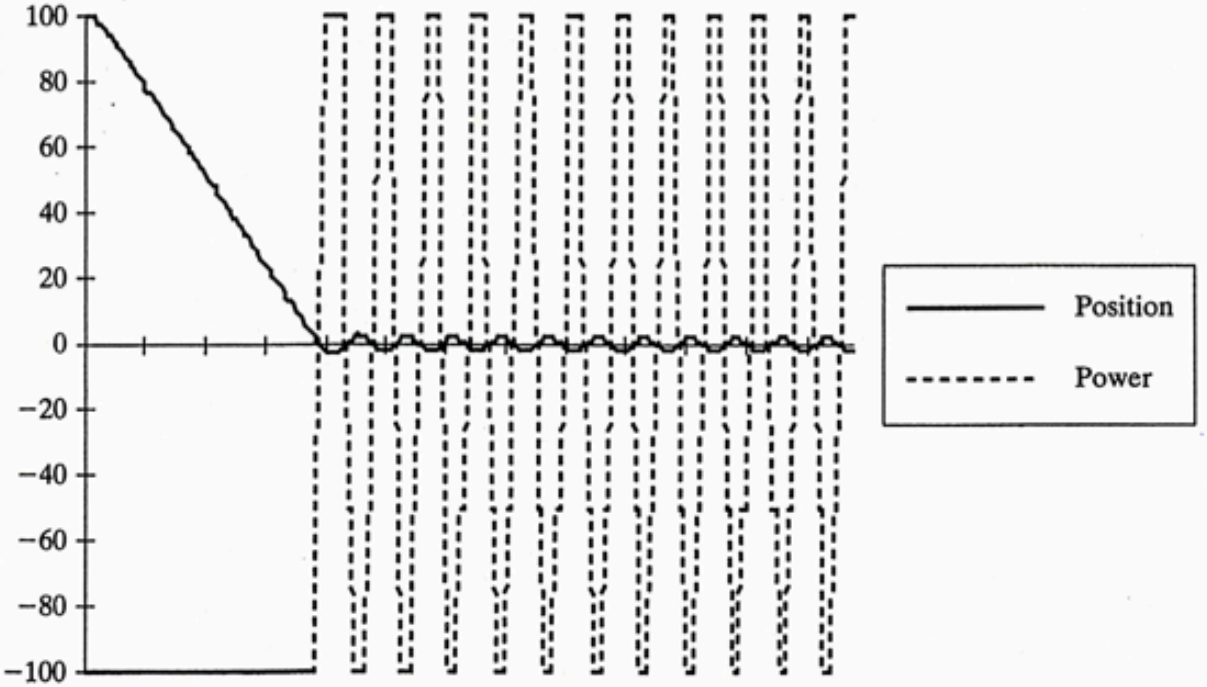
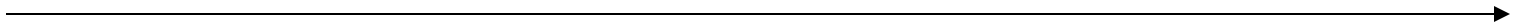



**FIGURE 5.13** Proportional Controller, $P_{gain} = 50$

error = Setpoint - counts

Power = Pgain * error  - Dgain * velocity



0

counts

error > 0
P term > 0

velocity >0
D term < 0

error < 0
P term < 0

velocity <0
D term > 0

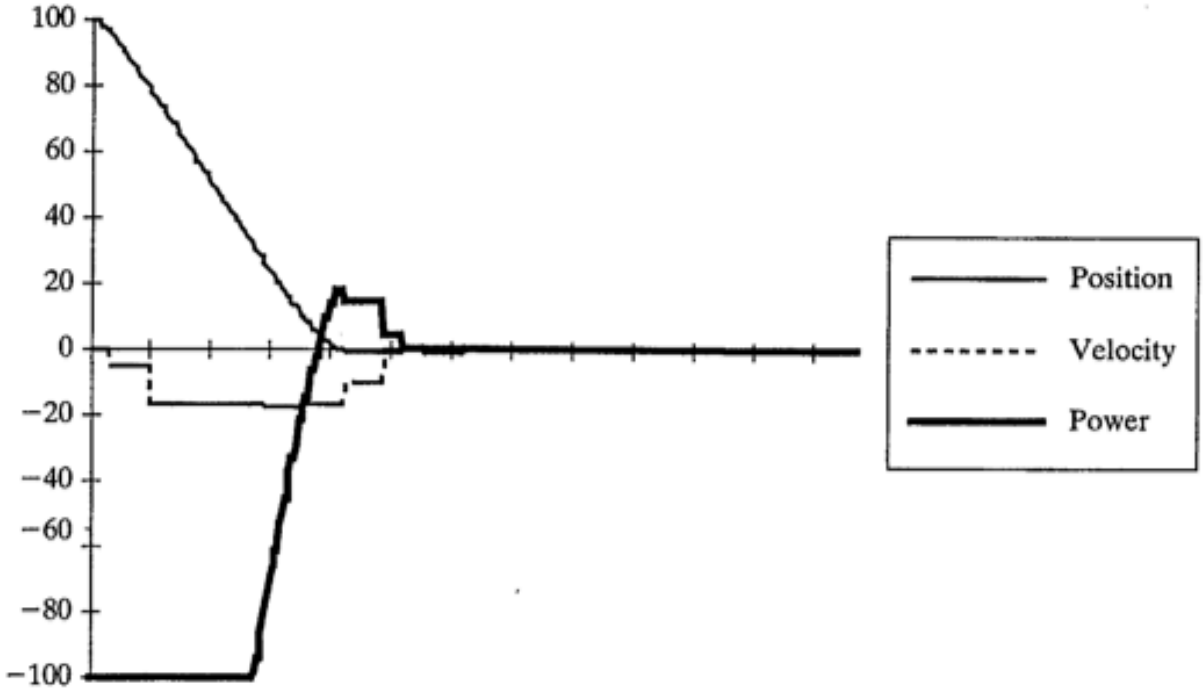



**FIGURE 5.14** Proportional-Derivative Controller, $P_{gain} = D_{gain} = 1$

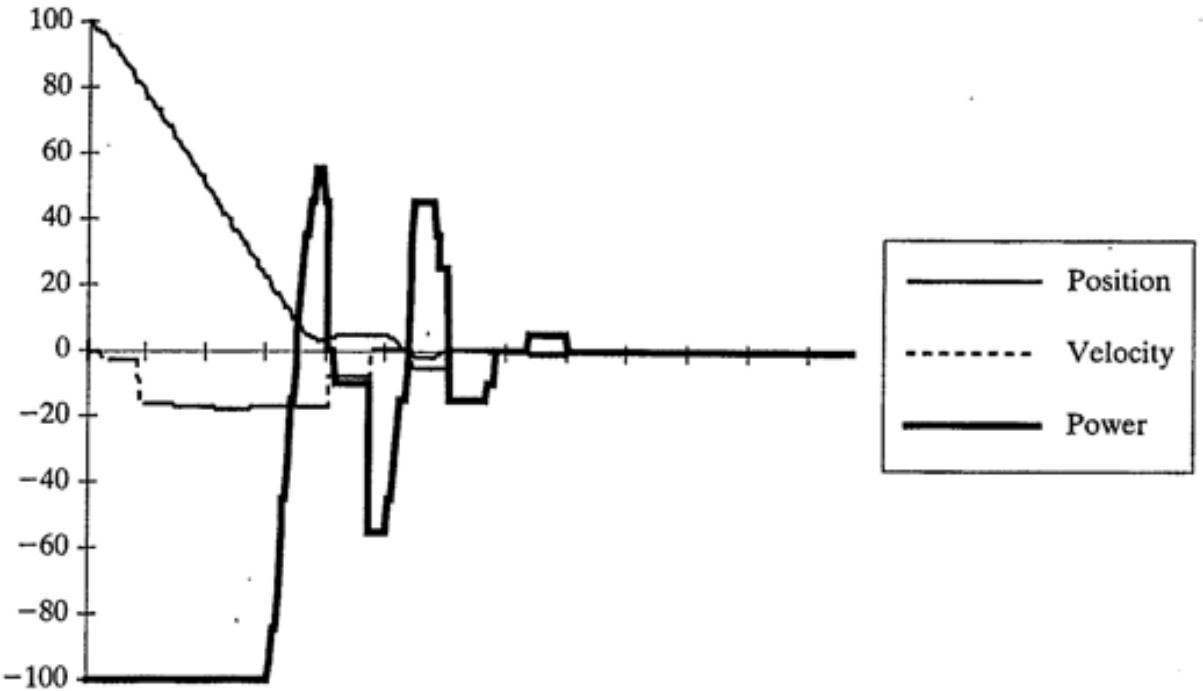
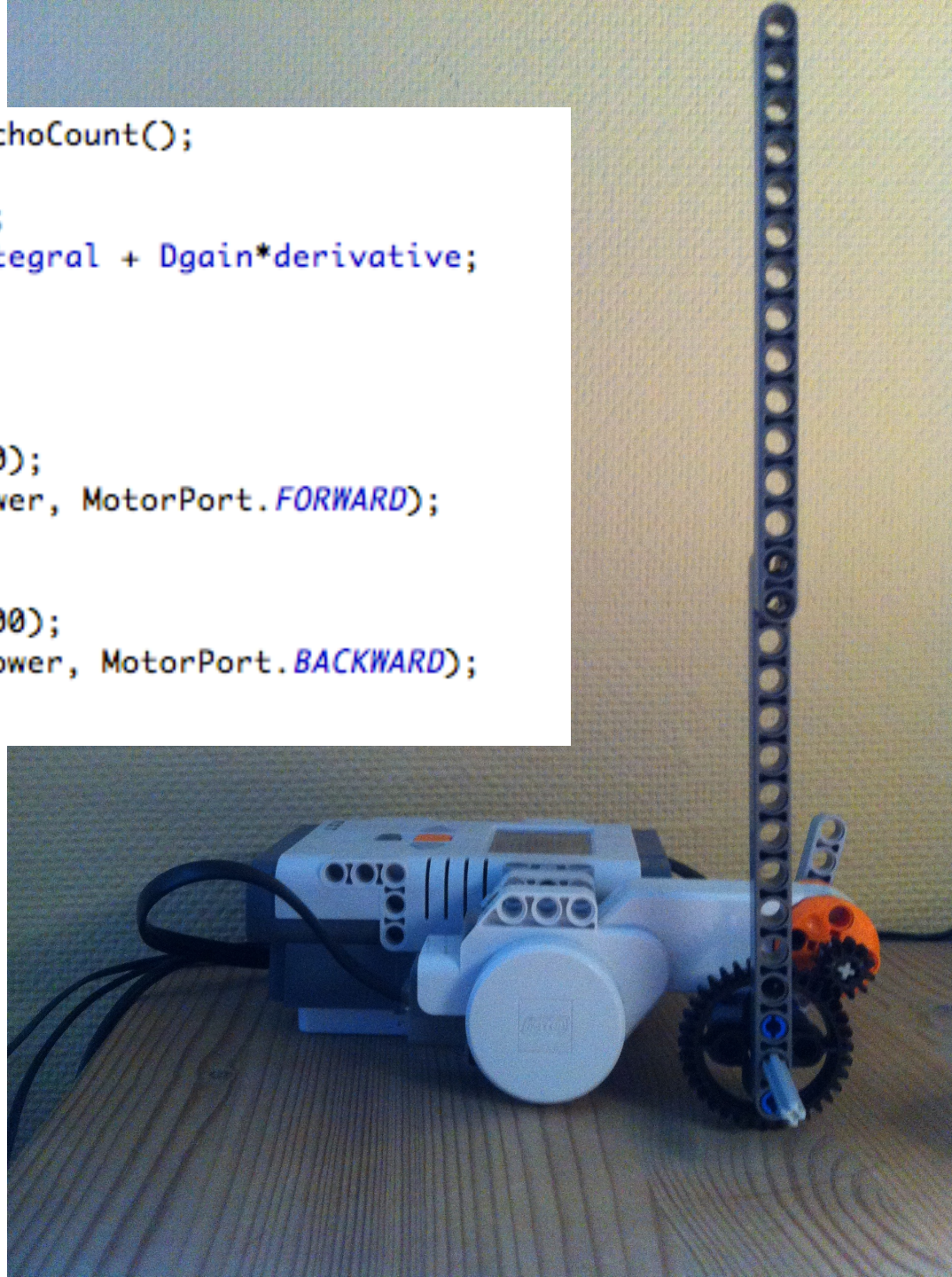


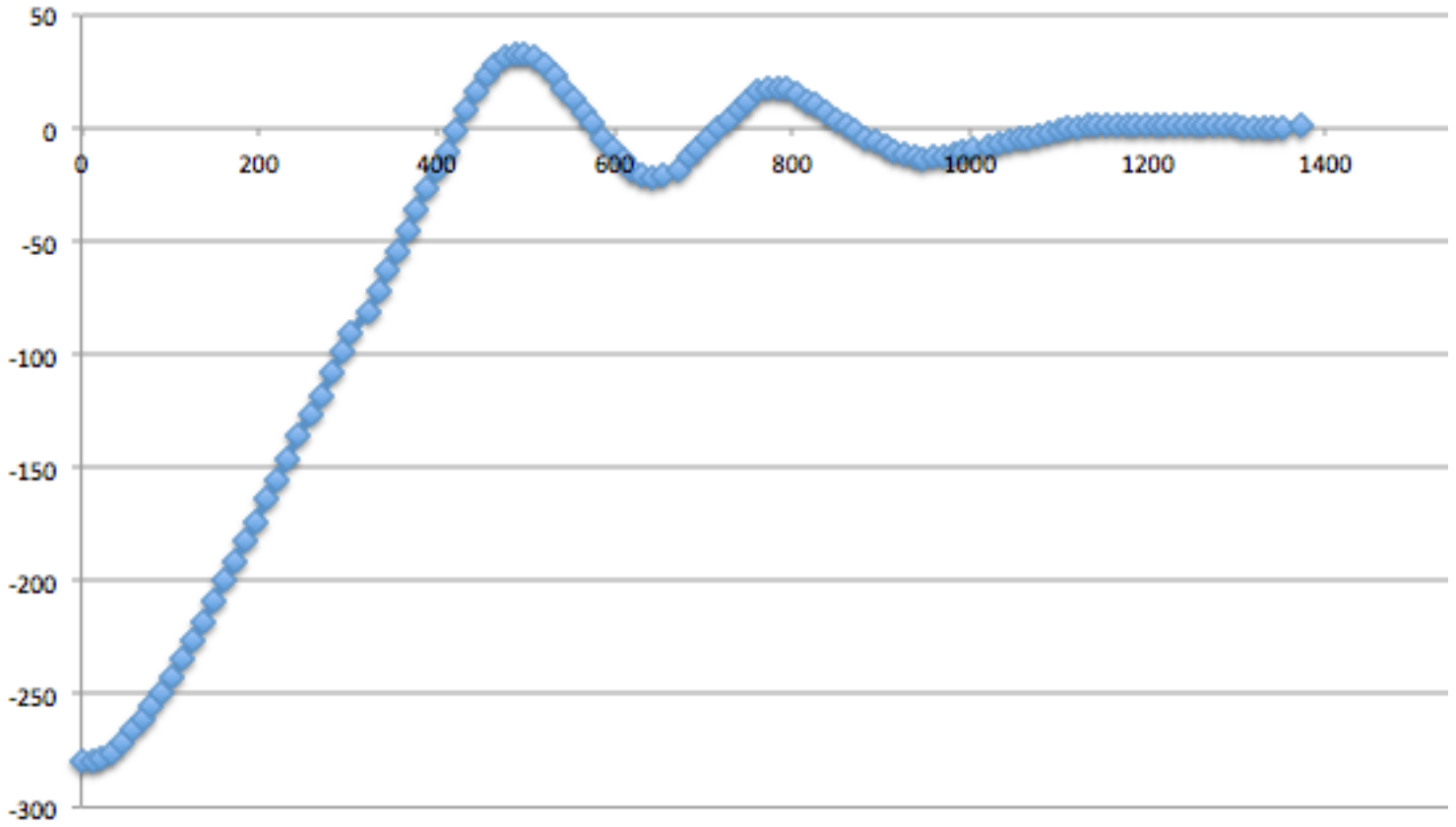

**FIGURE 5.15** Proportional-Derivative Controller, $P_{gain} = 10$; $D_{gain} = 5$

```
counts= encoder10_counts;
velocity= encoder10_velocity;

power= pgain * (0 - counts) - dgain * velocity;
motor(0, power);
```

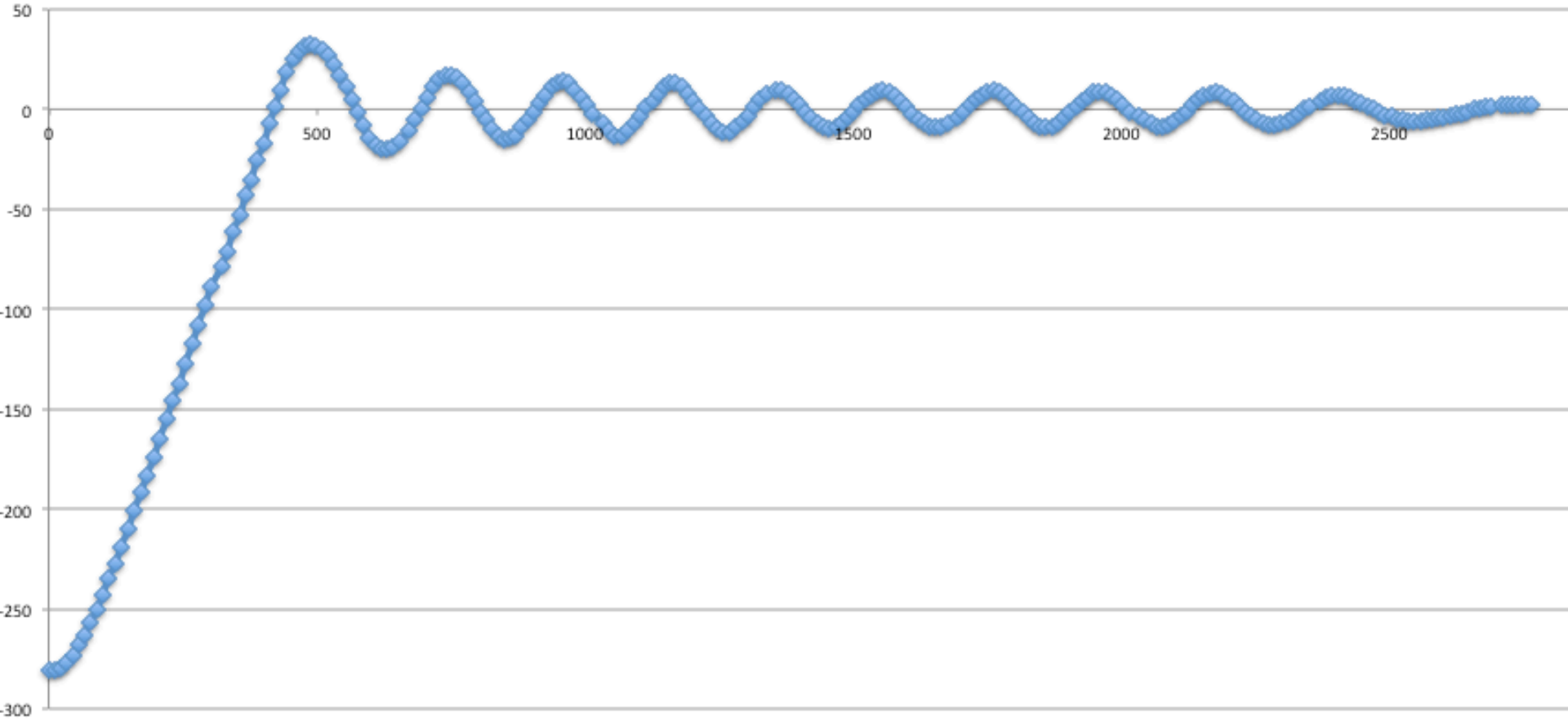
```
error = setpoint - motor.getTachoCount();
integral = integral + error;
derivative = error - lastError;
power = Pgain*error + Igain*integral + Dgain*derivative;

lastError = error;

if ( power > 0){
     power = Math.min(power, 100);
     motor.controlMotor((int)power, MotorPort.FORWARD);
}
else{
     power = Math.max(power, -100);
     motor.controlMotor(-(int)power, MotorPort.BACKWARD);
}
```



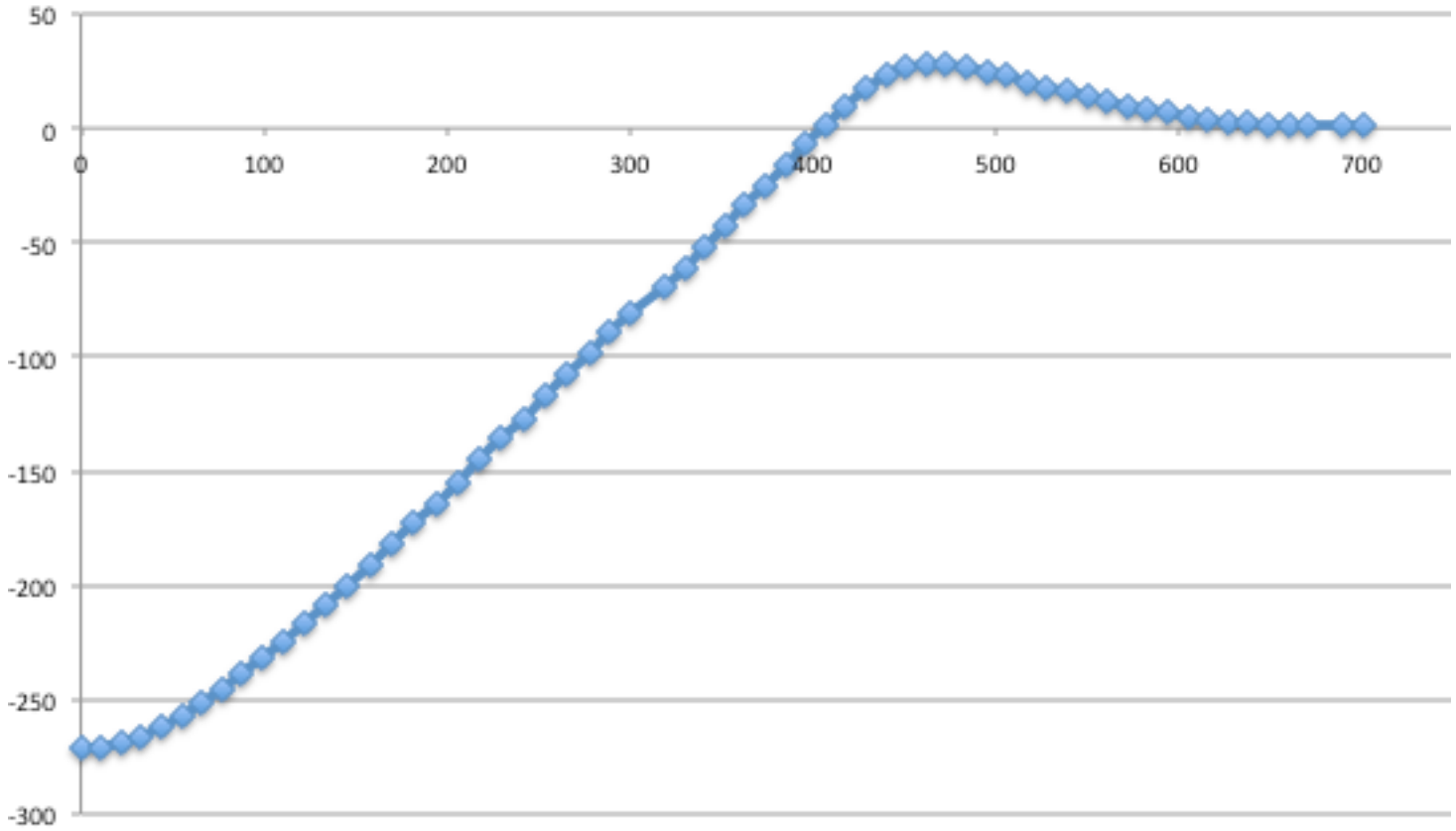
Pgain = 5  Igain = 0  Dgain = 0



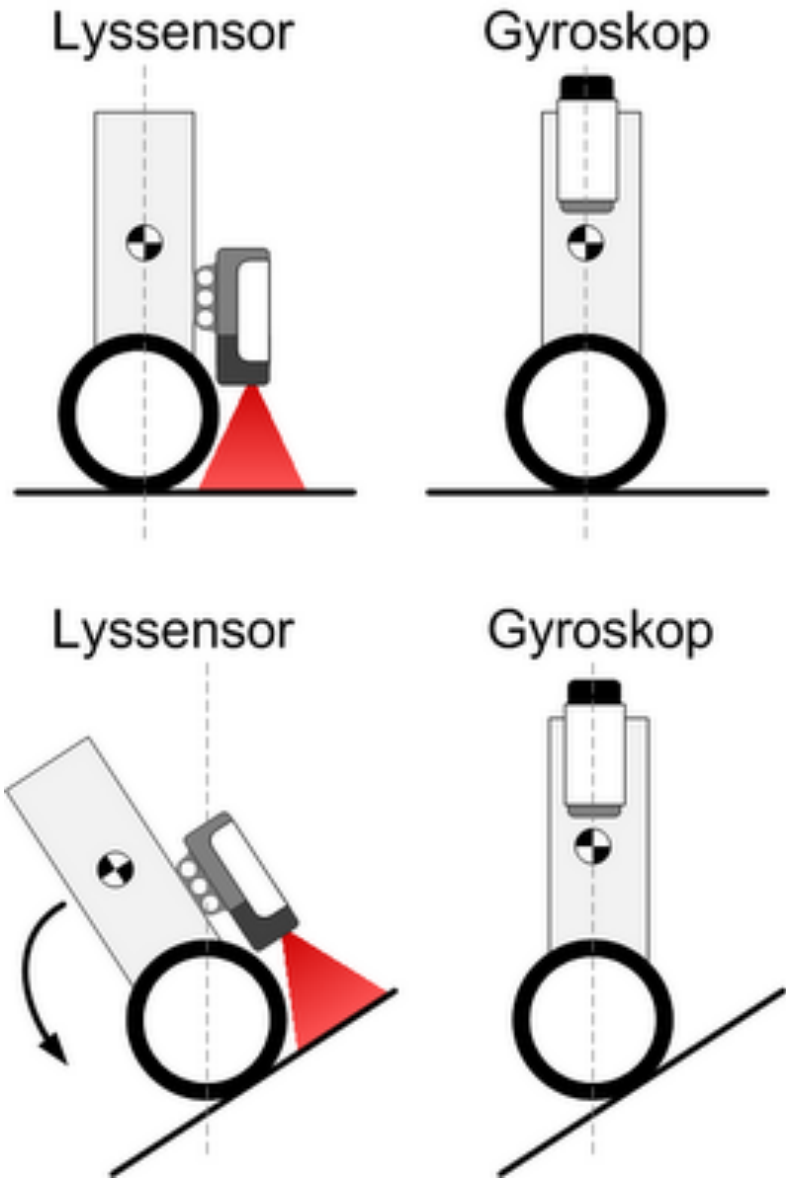
Pgain = 10  Igain = 0  Dgain = 0



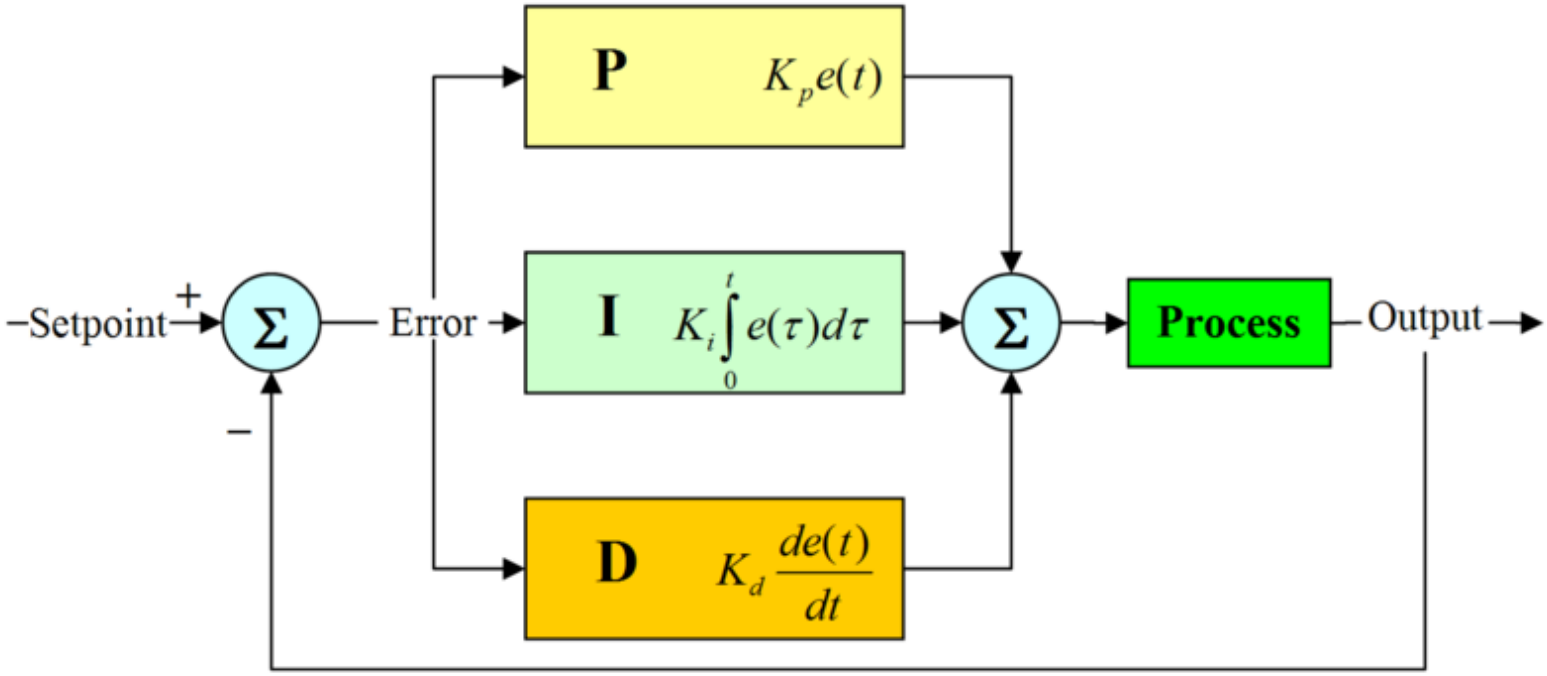
Pgain = 5  Igain = 0.01  Dgain = 10




NXT


Lyssensor
Gyroskop
Lyssensor
Gyroskop

```
double Psi = gyro.getAngle();
double PsiDot = gyro.getAngleVelocity();

// ctrl.tiltAngle() is used to drive the robot forwards and backwards
double Phi = motors.getAngle() - ctrl.tiltAngle();
double PhiDot = motors.getAngleVelocity();
// Proportional Error
double error = Psi * K_psi + Phi * K_phi + PsiDot * K_psidot + PhiDot
* K_phidot;
// Integral Error
int_error += error;
// Derivative Error
double deriv_error = error - prev_error;
prev_error = error;
// Power sent to the motors
double pw = error * Kp + deriv_error * Kd + int_error * Ki;
```


P
$K_p e(t)$
I
$K_i \int_0^t e(\tau)d\tau$
D
$K_d \frac{de(t)}{dt}$
Setpoint
+
−
Σ
Error
Σ
Process
Output

# Embedded Java

The leJOS API classes provide access to the hardware.  Especially the i/o devices.

Battery

Button

LCD

Sound

lejos.nxt

# Class Battery

java.lang.Object
  └ **lejos.nxt.Battery**

---

```
public class Battery
extends Object
```

Provides access to Battery.

---

## Method Summary

| | |
|---|---|
| static float | **getVoltage**()<br>Returns the battery voltage in volts. |
| static int | **getVoltageMilliVolt**()<br>Returns the battery voltage in millivolts. |
| static boolean | **isRechargeable**() |

```
public class Battery
{
  private static final int RECHARGEABLE = 0x8000;
  private static final int VOLTAGE_MASK = 0x3fff;

  private Battery()
  {
          //nothing
  }

  /**
   * Returns the battery status.
   * Low bits are the voltage in mV, bit 0x8000 is set if the rechargeable
   * battery pack is in use.
   * @return
   */
  private static native int getBatteryStatus();

  /**
   * Returns the battery voltage in millivolts.
   *
   * @return Battery voltage in mV.
   */
  public static int getVoltageMilliVolt()
  {
      return getBatteryStatus() & VOLTAGE_MASK;
  }

  public static boolean isRechargeable()
  {
      return (getBatteryStatus() & RECHARGEABLE) != 0;
  }

  /**
   * Returns the battery voltage in volts.
   *
   * @return Battery voltage in Volt.
   */
  public static float getVoltage()
  {
    return Battery.getVoltageMilliVolt() * 0.001f;
  }
}
```

lejos.nxt

# Class Button

java.lang.Object
  └ **lejos.nxt.Button**

**All Implemented Interfaces:**
ListenerCaller


---

```
public class Button
extends Object
implements ListenerCaller
```

Abstraction for an NXT button. Example:

```
Button.ENTER.waitForPressAndRelease();
Sound.playTone (1000, 1);
```

---

## Field Summary

| | |
|---|---|
| static Button[] | **BUTTONS**<br>Array containing ENTER, LEFT, RIGHT, ESCAPE, in that order. |
| static Button | **ENTER**<br>The Enter button. |
| static Button | **ESCAPE**<br>The Escape button. |
| static Button | **LEFT**<br>The Left button. |
| static Button | **RIGHT**<br>The Right button. |

## isDown

```
public final boolean isDown()
```

Check if the current state of the button is down.

**Returns:**

`true` if button is down, `false` if up.

---

## isUp

```
public final boolean isUp()
```

Check if the current state of the button is up.

**Returns:**

`true` if button is down, `false` if up.

---

## waitForPress

```
public final void waitForPress()
```

Wait until the button is released.

---

## waitForPressAndRelease

```
public final void waitForPressAndRelease()
```

Wait until the button is released.

lejos.nxt

# Class LCD

java.lang.Object
└ **lejos.nxt.LCD**

---

```
public class LCD
extends Object
```

Text and graphics output to the LCD display.

The LCD is a 100 x 64 pixel black & white graphical display.

When the LCD class is used to output text each character or digit has a width of 6 pixels and a height of 8 pixels.
This means that there is room for 8 lines of 16 characters on the LCD.

## drawString

```
public static void drawString(String str,
                              int x,
                              int y)
```

Display a string on the LCD at specified x,y co-ordinate.

**Parameters:**

`str` - The string to be displayed
`x` - The x character co-ordinate to display at.
`y` - The y character co-ordinate to display at.

---

## drawInt

```
public static void drawInt(int i,
                           int x,
                           int y)
```

Display an int on the LCD at specified x,y co-ordinate.

**Parameters:**

`i` - The value to display.
`x` - The x character co-ordinate to display at.
`y` - The y character co-ordinate to display at.

---

## drawInt

```
public static void drawInt(int i,
                           int places,
                           int x,
                           int y)
```

lejos.nxt

# Class Sound

java.lang.Object
└ **lejos.nxt.Sound**

---

```
public class Sound
extends Object
```

NXT sound routines.

## beep

```
public static void beep()
```

Beeps once.

---

## twoBeeps

```
public static void twoBeeps()
```

Beeps twice.

---

## beepSequence

```
public static void beepSequence()
```

Downward tones.

---

## beepSequenceUp

```
public static void beepSequenceUp()
```

Upward tones.

---

## buzz

```
public static void buzz()
```

Low buzz

## beep

```
public static void beep()
```

Beeps once.

---

## twoBeeps

```
public static void twoBeeps()
```

Beeps twice.

---

## beepSequence

```
public static void beepSequence()
```

Downward tones.

---

## beepSequenceUp

```
public static void beepSequenceUp()
```

Upward tones.

---

## buzz

```
public static void buzz()
```

Low buzz

```
/**
 * Upward tones.
 */
public static void beepSequenceUp()
{
    systemSound(true, 2);
}
```

## beep

```
public static void beep()
```

Beeps once.

---

## twoBeeps

```
public static void twoBeeps()
```

Beeps twice.

---

## beepSequence

```
public static void beepSequence()
```

Downward tones.

---

## beepSequenceUp

```
public static void beepSequenceUp()
```

Upward tones.

---

## buzz

```
public static void buzz()
```

Low buzz

```
public static int C2 = 523;


else if (aCode == 2)// C major arpeggio
    for (int i = 4; i < 8; i++)
    {
        playTone(C2 * i / 4, 100);
        pause(100);
    }
```

# playTone

```
public static void playTone(int freq,
                            int duration)
```

```
public static void playTone(int freq, int duration)
{
    playTone(freq, duration, VOL_MAX);
}
```

```
/**
 * Plays a tone, given its frequency and duration.
 * @param aFrequency The frequency of the tone in Hertz (Hz).
 * @param aDuration The duration of the tone, in milliseconds.
 * @param aVolume The volume of the playback 100 corresponds to 100%
 */
static native void playFreq(int aFrequency, int aDuration, int aVolume);
```

## playSample

```
public static int playSample(File file,
                             int vol)
```

Play a wav file

**Parameters:**

`file` - the 8-bit PWM (WAV) sample file

`vol` - the volume percentage 0 - 100

**Returns:**

The number of milliseconds the sample will play for or < 0 if there is an error.

```
/**
 * Internal method used to play sound sample from a file
 * @param page the start page of the file
 * @param offset the start of the samples in the file
 * @param len the length of the sample to play
 * @param freq the sampling frequency
 * @param vol the volume 100 corresponds to 100%
 */
static native void playSample(int page, int offset, int len, int freq, int vol);
```

# NXT Programming

---

## Lesson 1

In this lesson we build a LEGO car to be controlled by the LEGO Mindstorms NXT. Then we **install the leJOS Java system**, [1], and use this to **compile and upload** a Java program to the NXT. The program will make the car follow a black line on a white surface.

### The 9797 LEGO car

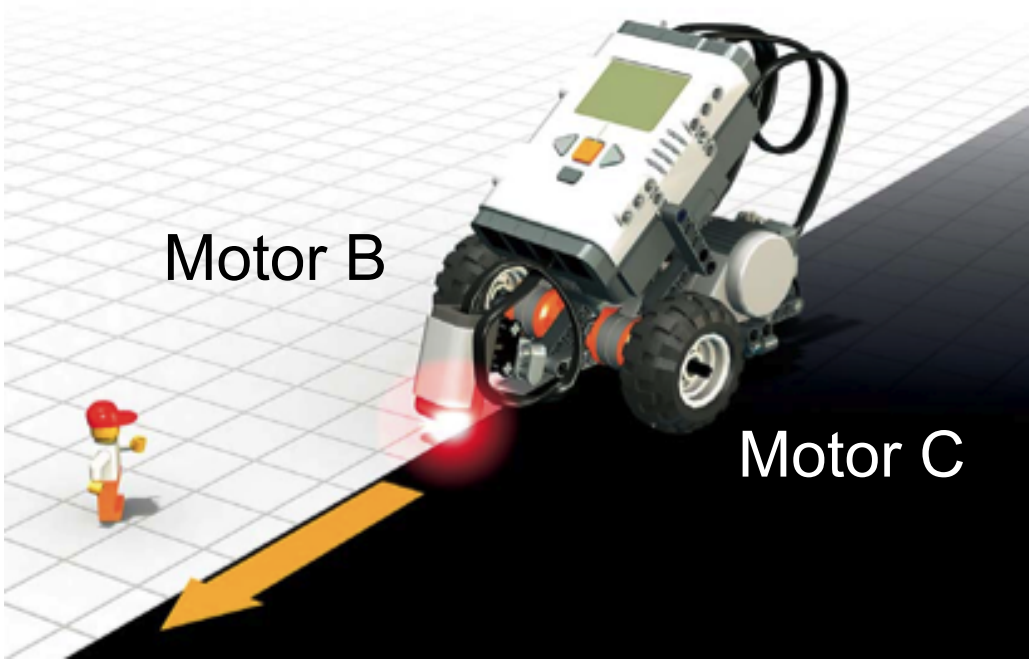
In the LEGO Mindstorms Education NXT Base Set 9797 there is a building instruction for a car, page 8 to page 22. Page 32 to page 34 shows how a light sensor can be added to the car. Build this car with a light sensor added.

**Figure 1** The 9797 LEGO car with two motors.

### A Java Control Program: LineFollower

The first Java program that we are going to execute on the NXT is the following Java program that makes the LEGO car follow a black line on a white surface: (`LineFollower.java`):

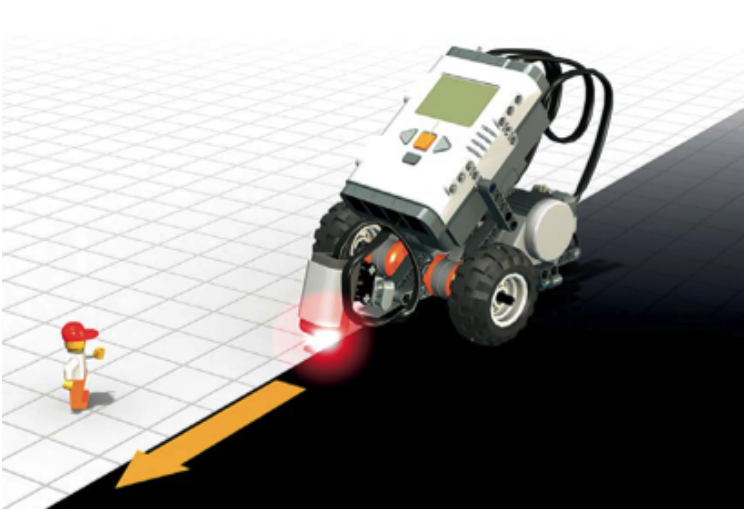
```
LightSensor light = new LightSensor(SensorPort.S3);
final int blackWhiteThreshold = 45;


// Use the light sensor as a reflection sensor
light.setFloodlight(true);


LCD.drawInt(light.readValue(), 3, 9, 0);


if (light.readValue() > blackWhiteThreshold)
```

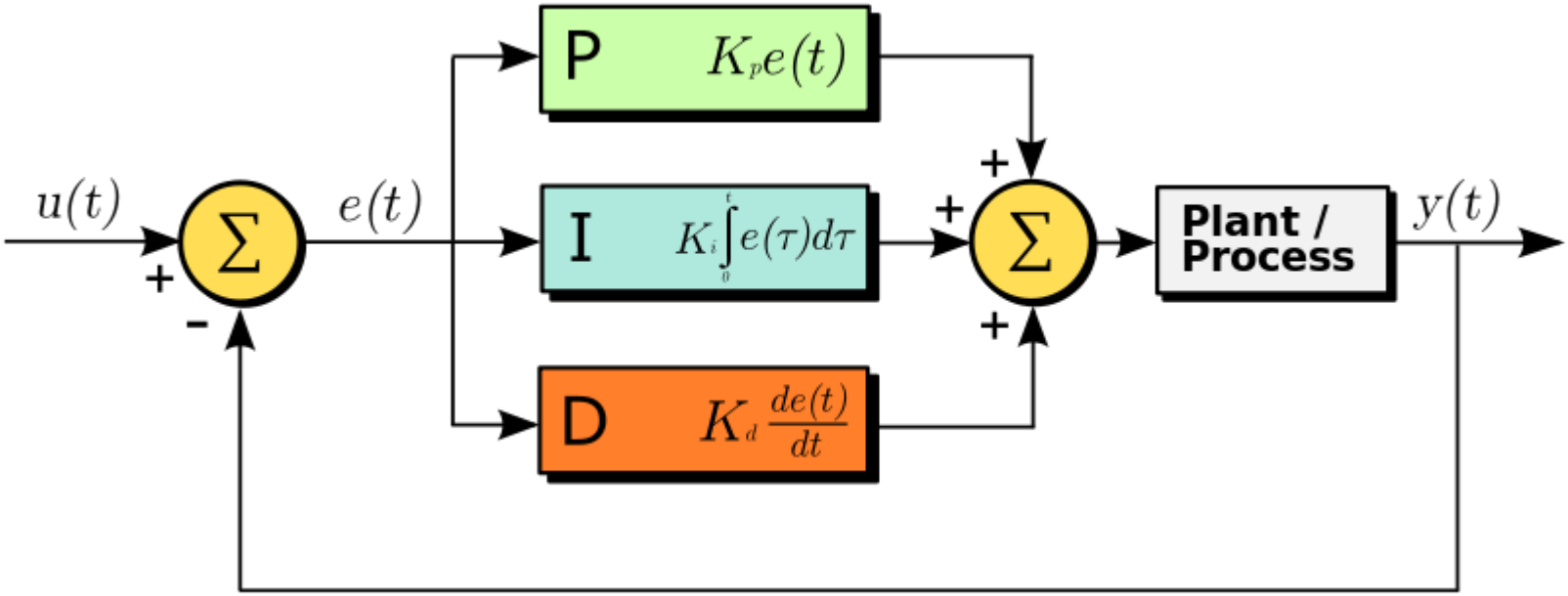



```
MotorPort.B.controlMotor(power, forward);
MotorPort.C.controlMotor(0,stop);

MotorPort.B.controlMotor(0,stop);
MotorPort.C.controlMotor(power, forward);
```

```
while (! Button.ESCAPE.isDown()){

    if (light.readValue() > blackWhiteThreshold){
        // On white, turn left
        LCD.drawString(right, 0, 1);
        MotorPort.B.controlMotor(power, forward);
        MotorPort.C.controlMotor(0,stop);
    }
    else{
        // On black, turn right
        LCD.drawString(left, 0, 1);
        MotorPort.B.controlMotor(0,stop);
        MotorPort.C.controlMotor(power, forward);
    }
    LCD.drawInt(light.readValue(), 3, 9, 0);
    Thread.sleep(10);

}
```

```
while (! Button.ESCAPE.isDown()){

    lightValue = sensor.readRawValue();
    error = lightValue - offset;
    integral = integral + error;
    derivative = error - lastError;
    turn = Kp*error + Ki*integral + Kd*derivative;
    leftPower  = (int)(minPower + turn);
    rightPower = (int)(minPower - turn);

    if ((lastError >=0 && error <=0) || (lastError <=0 && error >=0))
       integral = 0;

    lastError = error;
    Car.forward(leftPower, rightPower);
    Thread.sleep(10);

}
```




P
$K_p e(t)$
I
$K_i \int_0^t e(\tau)d\tau$
D
$K_d \frac{de(t)}{dt}$
$u(t)$
$e(t)$
$y(t)$
Plant / Process
+
-
+
+
+

---

**Date:**
**Duration of activity:**
**Group members participating:**

---

Furthermore, each activity should be described by:

- a **goal** (or goals) for this activity, maybe with a list of subgoals,
- a **plan** for the activity including a description of methods that lead to fulfillment of the goal(s),
- **results** obtained including descriptions of
  - experiments together with a description that other groups can use to reproduce your experiments,
  - programming attemts with program segments and links to programs,
  - output from programs,
  - measurements,
  - pictures of LEGO models,
  - problems encountered.

- a **conclusion** with a status and suggestions for what to do next.
- **references** to papers, web pages or copied material.