CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, or www.circuitcellar.com. Entire contents copyright ©2004 Circuit Cellar Inc. All rights reserved.

FEATURE ARTICLE

by Rich LeGrand

Closed-Loop Motion Control for Mobile Robotics

Rich recently came up with two closed-loop drive train designs for mobile robots. All it took were some inexpensive permanent magnet motors and a simple feedback scheme. In this article, he covers everything from PID control and tuning to trajectory generation and operational space control for two robot bases. He also explains the software.

The mechanical components that make up robots aren't getting much cheaper. That's the bad news. The good news, of course, is that electronics and processors continue their steady march toward higher performance and lower cost. As an engineer, you have to figure out how to use the extra resources and capabilities most effectively. In some cases, extra processing allows you to use lower-cost mechanical components, which can effectively reduce the system's overall cost.

With this idea in mind, I'll tackle the robot drive train armed with a modest amount of computing power. When I'm done, you'll have two different closed-loop drive train designs that would make R2-D2's head spin with envy. I'll show you how to accomplish this using inexpensive permanent magnet motors and a clever feedback scheme that requires no mechanical overhead. Let's begin by addressing the topic of motors.

MOTOR OPTIONS

Clearly, selecting a motor is one of the most important decisions to make when designing the drive train. If you've played with motors and gears, you've certainly developed the following intuition: more gear reduction results in less speed and more torque, and less gear reduction results in more speed and less torque. Modified RC servomotors, which are often used in robot drive trains, have large gear reductions (typically 300:1) resulting in high torques and low speeds. As a result, robots that use them are typically slow. Large gear reductions make some problems easier, as you'll see, but no one wants a slow robot if they can help it. If you've used these motors, you might have wanted to trade in some torque for some speed.

The 9-V Lego motors shown in Photo 1 are readily available. They are part of the Mindstorms kit, so you can build the rest of your robot base out of Legos, which is a good thing for the mechanically challenged like me.

Fortunately, the Lego motor has much less gear reduction (14:1) and is well suited for attaching a wheel directly to the output shaft. If you've done this, you know that you get a quick robot, but it's difficult to control. Particularly, when you shut off power to the motor, the robot takes much longer to stop. With less gear reduction, the robot will coast!

Sometimes stopping quickly is important (e.g., stairs ahead), or your robot may do something hazardous to its health. This brings up an important point: motors with high gear reductions stop quickly when power is removed. But this is a silly reason to use these motors. It's like telling a beginning driver to drive only in first gear to avoid getting into an accident. What the driver really needs is better driving skills. Similarly, what you need is a better motor controller.

CLOSING THE LOOP

A motor controller that doesn't receive any feedback information from the motor is an open-loop controller. The main advantage of open-loop controllers is simplicity. All that's required is a way to control the voltage or current going to the motor. The Lego Mindstorms RCX controller, for example, uses an open-loop motor controller that allows you to select from several voltages (by varying the pulse-width modulator duty cycle) depending on how much speed and torque you want to deliver to your robot's wheels.

A disadvantage of open-loop control is inaccuracy. Others include the inability to deal with uncontrollable variables such as bumps in a floor, inclines, and low batteries, which can create an undesirable situation by slowing or stop-

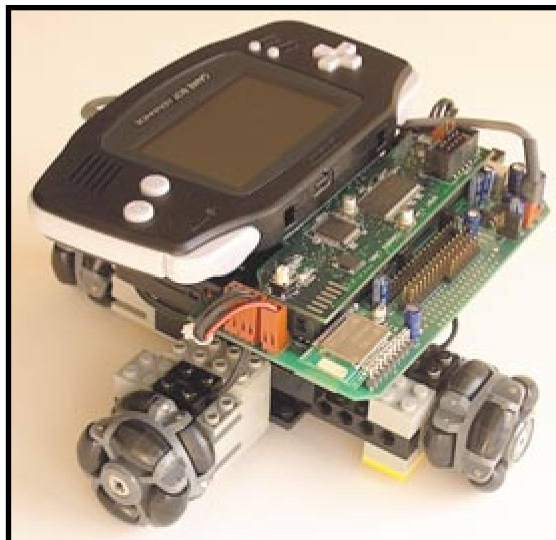

Photo 1—*Lego is an excellent medium for implementing robots such as the four-wheeled HUMR. The Gameboy Advance acts as the controller and provides 32-bit RISC processing performance, a color LCD, and sound.*

Method	Description	Advantages	Disadvantages	Sources
Optical-mechanical encoders	A rotating slotted disk is placed between a light source and detector to infer position.	No drift. Digital output integrates easily with digital controllers. Long lifetime.	Typically expensive. Requires extra cables.	Hewlett Packard, computer mouse
Mechanical encoders	Switches are triggered by the motor's motion to infer position.	No drift.	Typically expensive. Requires extra cables. Imposes drag. Output needs debouncing. Lower speed. Shorter lifetime.	Vishay, various industrial vendors
Hall effect sensors	When used with a magnet, they can sense metallic (ferrous) gear teeth to infer position.	No drift. Uses existing gear in gear train.	Typically expensive. Requires extra ables, extra magnet, and ferrous gear.	Allegro, various industrial vendors
Back EMF	The back EMF voltage of a motor is measured to infer velocity.	No extra mechanical components or cables. Typically inexpensive.	Drift. Requires A/D converter and extra computation to obtain position.	Acroname, Charmed Labs

Table 1—*There are a few popular feedback methods for sensing motor motion. Back EMF sensing is often overlooked, but its advantages can be attractive to many robotics applications.*

ping the motor. What's required is a way to sense the motion of the motor and compensate by increasing or reducing the power. Closed-loop controllers can do both.

Closed-loop controllers are found in all sorts of places: thermostats, cruise control systems, and elevators just to name a few. Almost without exception, commercial robots use closed-loop motor control. Even the Roomba, a $199 vacuuming robot, uses a closed-loop motor controller.

Closed-loop control requires a method for sensing the motor's motion. Table 1 lists some popular methods. The method that will work best for you depends as much on project constraints as your preferences. The majority of closed-loop motor controllers use optical-mechanical encoders for position feedback, but the extra cabling and mechanical complexity are usually worth avoiding. I chose back EMF as a feedback method. The mechanical simplicity (no mechanics) and lack of cables make it an attractive option.

Back EMF exploits the fact that permanent magnet motors are also generators. When a motor spins, a voltage is generated across its terminals. The voltage, referred to as the back EMF voltage, is directly proportional to the motor's velocity. Thus, when sensing the back EMF voltage with an A/D converter, for example, you can infer the motor's velocity. When the voltage is integrated (summed) over time, the position can be inferred as well.

The main disadvantage of back EMF sensing is that the inferred position drifts over time with respect to the actual position because of noise in the back EMF voltage. In practice, however, the error introduced by position drift is small when compared to the error introduced by wheel slippage alone. This performance can be obtained with a 9-V Lego motor and robot controller from Charmed Labs, which uses back EMF sensing.

PID CONTROL

Closed-loop motor control entails both sensing and controlling the motor's motion. I have described different sensing techniques. The general consensus is that an H-Bridge with pulse-width-modulation (PWM) is the best method for controlling the motor. (For more information, refer to L. Mays's article, "Muscle for High-Torque Robotics," *Circuit Cellar*, issue 153, 2003.) Here, a PWM signal switches an H-Bridge to control the voltage going into the motor and its speed. Note that I will refer to this type of controller throughout the rest article as a PWM controller. Its input will be referred to as the PWM value.

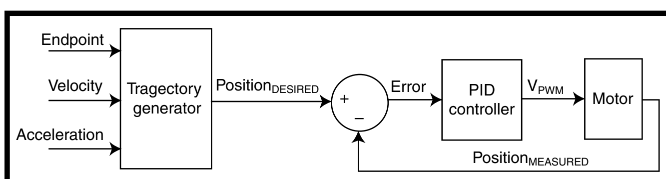

Figure 1—*A PID controller is typically used to control the velocity and position of a motor. I'll focus on implementing a PID position controller, which is shown here with the trajectory generator.*

When combining sensing and control in a closed-loop controller, the word "loop" can be taken literally. A control loop typically entails a software loop that repeatedly executes a control algorithm. Each repetition of the control algorithm is called a control cycle. The control algorithm can be described simply with the following expressions, which are evaluated once per control cycle:

$$error = V_{DESIRED} - V_{MEASURED}$$
$$V_{CONTROL} = f(error)$$

Basically, there is a function, f, which determines the value to send to your controller ($V_{CONTROL}$) as a function of the error. $V_{MEASURED}$ is the measured (sensed) value, and $V_{DESIRED}$ is the desired (commanded) value. The difference between the values is the control error. Note that one of the simplest control methods is the bang-bang controller, which you can find in your thermostat.

$$V_{CONTROL} = \begin{cases} \text{Heat if error} > 1° \\ \text{Cool if error} < -1° \\ \text{Off otherwise} \end{cases}$$

My thermostat doesn't automatically select between heating and cooling; it would be nice if it did. Many robots use bang-bang controllers for their motors. However, PID control is a much more effective technique:

$$V_{CONTROL}(k) = V_{PROPORTIONAL}(k) + V_{INTEGRAL}(k) + V_{DERIVATIVE}(k)$$

$$V_{PROPORTIONAL}(k) = K_P \times error(k)$$

$$V_{INTEGRAL}(k) = K_I \times \sum_{j=0}^{k} error(j)$$

$$V_{DERIVATIVE}(k) = K_D \times \left[error(k) - error(k-1)\right]$$

$$error(k) = V_{DESIRED}(k) - V_{MEASURED}(k)$$

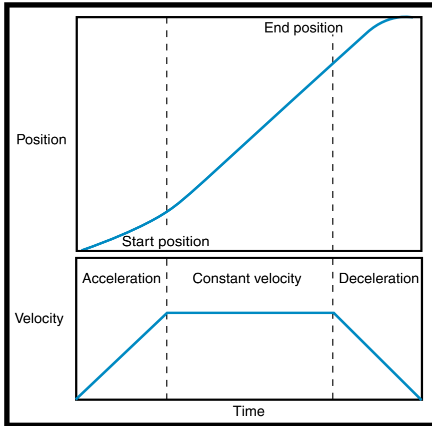

Figure 2—*The trapezoid trajectory gets its name from the shape of the velocity profile. It is used to move a motor to a desired end-position in a controlled manner.*

The control value at time k is equal to the sum of the proportional, integral, and derivative terms ($V_{PROPORTIONAL}$, $V_{INTEGRAL}$, and $V_{DERIVATIVE}$) at time k. K_P, K_I, and K_D are the PID gains, which are easily determined through experimentation, a process known as tuning. The PID controller is popular because of its effectiveness and relative simplicity. All that's required is a set of reasonable gain values.

Let's consider the relevant problem of controlling a motor's position (see Figure 1). $V_{CONTROL}$ becomes V_{PWM}, which is the PWM value sent to the motor's PWM controller. $V_{DESIRED}$ becomes $Position_{DESIRED}$, which is the desired position of the motor, and $V_{MEASURED}$ becomes $Position_{MEASURED}$, which is the measured position of the motor.

To get a feel for how the PID controller works, consider the proportional term by itself. If the position error is large, so is the proportional term and hence the resultant PWM value, V_{PWM}. This causes the motor to move quickly toward the desired position. As the motor closes in on the desired position (as the error decreases), the proportional term decreases, which slows the motor. Thus, the proportional term does almost all of the work. The other integral and derivative terms correct for problems that the proportional term cannot correct by itself. You will better understand these terms when I cover tuning a PID loop.

TRAJECTORY GENERATION

The PID controller is designed to get to the desired position ($Position_{DESIRED}$) as fast as possible. If your robot's only speed is "as fast as possible," it may cause harm to you and others. It's often useful to specify the speed and acceleration when commanding the motor controller. This is where the trajectory generator comes in. It produces a continuous stream of positions, or waypoints, for the PID controller to use to regulate the motor's motion. The trajectory generator sits outside the PID control loop as shown in Figure 1.

For example, a simple trajectory generator provides constant acceleration until the desired velocity is reached. It holds this velocity until it nears the desired endpoint position. Next, it provides constant deceleration until the endpoint is reached. This trajectory results in the velocity profile shown in Figure 2. It's called a trapezoid trajectory because of its shape. For simplicity, the acceleration and deceleration are equal in magnitude. Figure 2 also shows the corresponding positions associated with the velocity profile. These positions are quickly fed into the PID position controller, usually once per control cycle. The idea is that the trajectory generator provides positions rapidly enough so the motor moves smoothly along the desired trajectory.

Generating trapezoid trajectories is relatively straightforward. Refer to the *Circuit Cellar* ftp site for a working implementation.

WRITING CODE

Let's write some real code! I'm a big fan of C++. Its class and class inheritance concepts mean that I don't have to write as much code. And writing less code is right up there with watching less TV and reducing my cholesterol.

This closed-loop motor controller lends itself nicely to a class hierarchy. `CAxesOpen` is the base class and an easy starting point. It implements an

Listing 1—*`CAxesOpen` and `CAxesClosed` form the first two classes in the motor controller class hierarchy and a majority of the code. The complete implementation can be found on the* Circuit Cellar *ftp site.*

```
#define AC_MAX_AXES     4
class CAxesOpen
{
public:
    CAxesOpen(int servoAxes);      // Constructor
    virtual ~CAxesOpen();          // Destructor

    virtual int GetPosition(int axis);
    void SetPWM(int axis, int pwm);
//...
};
class CAxesClosed : public CAxesOpen
{
public:
    CAxesClosed(int servoAxes, int operationalAxes=1);
      // Constructor
    virtual ~CAxesClosed();        // Destructor

    void Periodic();               // Called once per control cycle
    bool Done(int axis);
// Returns true if trajectory is finished
    void SetPIDGains(int pGain, int iGain, int dGain);
    void Stop(int axis);           // Stop immediately
    void Hold(int axis, bool val); // Hold current position
    virtual void Move(int axis,    // Perform trajectory move
        int endPosition, int velocity, int acceleration);
    virtual int GetPosition(int axis);
protected:
// Called by GetPosition()
    virtual void ForwardKinematics(const int servoVal[], int
      operVal[]);
// Called by Periodic()
    virtual void InverseKinematics(const int operVal[], int
      servoVal[]);
private:
    void TrapezoidTrajectory();
    void PIDControl();
```

(Continued)

Listing 1—*Continued.*

```
   int m_servoAxes, m_operationalAxes;
// Trajectory input parameters
   int m_trajectoryEndPosition[AC_MAX_AXES];
   int m_trajectoryVelocity[AC_MAX_AXES];
   int m_trajectoryAcceleration[AC_MAX_AXES];
   unsigned int m_trajectory;     // True if trajectory is active
// Trajectory generator output
   int m_generatedTrajectoryVelocity[AC_MAX_AXES];
   int m_generatedTrajectoryPosition[AC_MAX_AXES];
// PID controller variables
   int  m_pGain, m_iGain, m_dGain;
   int  m_desiredPosition[AC_MAX_AXES];
   int  m_errorIntegral[AC_MAX_AXES];
   int  m_errorPrevious[AC_MAX_AXES];
   unsigned int m_hold; // For maintaining the current position
};
```

Listing 2—*The code that implements the PID position controller (`PIDControl()`) is surprisingly simple. It is called from `Periodic()`, which is called once per control cycle.*

```
void CAxesClosed::Periodic()
{
   TrapezoidTrajectory();
   InverseKinematics(m_generatedTrajectoryPosition,
    m_desiredPosition);
   PIDControl();
}
void CAxesClosed::PIDControl()
{
   int error, pwm;
   for (int axis=0; axis<m_servoAxes; axis++)
   {
      if (m_trajectory || m_hold)
      {
         error = m_desiredPosition[axis] -
          CAxesOpen::GetPosition(axis);
         pwm = m_pGain*error +
               m_iGain*m_errorIntegral[axis] +
               m_dGain*(error - m_errorPrevious[axis]);
         m_errorIntegral[axis] += error;
         m_errorPrevious[axis] = error;
      }
      else
         pwm = 0;
      SetPWM(axis, pwm);
   }
}
```

Listing 3—*A simple program for tuning the PID control loop entails holding the current position. Typically, `Periodic()` is called from a timer-generated interrupt service routine, which effectively makes the control loop a background process. But to simplify implementation and testing, I used a `while()` loop to call `Periodic()` here.*

```
#define TIMER_PERIOD  5000  // Microseconds
main()
{
   CAxesClosed cAxis(1);    // One axis
   cAxis.SetGains(100, 0, 0);
   cAxis.Hold(0, true);     // Hold current position
   while(1)
   {
      ResetTimer();
      cAxis.Periodic();
      while(GetTimer()<TIMER_PERIOD);
   }
}
```

open-loop motor controller, which allows you to set the PWM value and get the position of each motor axis within a set of axes. As Listing 1 shows, `CAxesOpen` is extremely simple with its two public functions. Note that `SetPWM()` accepts a signed PWM value. It is intended that the sign of this value determine the direction of the motor.

`CAxesOpen` is an easy first step, but it isn't very useful by itself. You need to implement another class that closes the loop. The closed-loop controller class is called (not surprisingly) `CAxesClosed`, and inherits from `CAxesOpen`. As shown in Listing 1, `CAxesClosed` is a little more complex. The complete source code is posted on the *Circuit Cellar* ftp site.

`CAxesClosed` is bigger, but it's doing almost all of the work in the closed-loop control system. It implements the PID control algorithm and the trapezoid trajectory generator, and ties it all together.

Using `CAxesClosed` entails periodically calling `Periodic()` for each control cycle from an external source. Looking at the contents of `Periodic()` in Listing 2, it makes a call to the trajectory generator (`TrapezoidTrajectory()`) and the PID controller (`PIDControl()`). It also makes a call to `InverseKinematics()`, which contains the kinematics of your robot base, if applicable. `CAxesClosed`'s implementation of `InverseKinematics()` doesn't do anything useful, but a derived class can override this member function if it wishes, as you will see.

Call the `Move()` function when you want your motor to move. It takes the desired trajectory parameters as inputs and initiates a trajectory, which will hopefully result in the desired motion. But, before you can move any motors, you need to tune the PID control loop.

TUNING

You can use `CAxesClosed` to help tune the PID control loop. Listing 3 provides a simple program that you can use for tuning purposes. You instantiate `CAxesClosed` with one axis for tuning. The actual control loop is the `while` loop that calls `Periodic()`. The `ResetTimer()` and `GetTimer()` functions reset and read the timer value, respectively. The implementations of these functions are

platform-specific and keep the calls to Periodic() evenly spaced in time.

The control frequency is the number of control cycles executed per second. In general, the higher the control frequency, the better the control. Your processor's available bandwidth typically determines the control frequency, however. Choose a frequency that spares enough bandwidth for the other computing tasks you've slated for your processor. Note that calling Periodic() from within a while loop will consume all of your processor's bandwidth (see Listing 4). Periodic() is intended to be called from within an interrupt-driven timer routine to prevent this from happening. Calling Periodic() from within a while loop is much easier to implement and debug, so you can defer the added complexity for now.

The call to Hold() before the while loop enables the PID loop but not the trajectory generator. In other words, it causes the motor to hold its position. If you try to move the motor, it will resist. And if you manage to turn the motor and then let go, the motor zips back to its original position and resumes holding its position. At least that's what is supposed to happen. It only happens when the PID loop is reasonably well tuned.

There is plenty of literature available regarding how to tune a PID control loop. I tuned mine by hand, which means I determined a set of PID gains through experimentation.

Before you begin tuning by hand, it is useful to attach a wheel to the motor shaft, preferably the wheel you will be using on your robot. This makes it easier to turn the shaft and witness the control loop's response. When I said "by hand," I meant it literally! Also, make sure the position feedback and PWM control have the same *sign*. When providing the motor with a positive PWM value, the position feedback value should increase as the motor moves under its own power. Similarly, when providing negative PWM, the position feedback should decrease.

As mentioned earlier, the proportional gain does most of the work. Tuning should begin by adjusting this value. Start with a small value like, say, 10 (e.g., cAxis.SetGains(10, 0, 0)).

Listing 4—*CDiffBase derives from CAxesClosed and allows operational space control of a differential base by substituting its own kinematics routines.*

```
#define DIFF_AXES        2
#define TRANSLATE_AXIS   0
#define ROTATE_AXIS      1
CDiffBase : public CAxesClosed
{
public:
  CDiffBase(int rotationScale, int translationScale, int controlFreq);
  virtual ~CDiffBase();
  virtual void Move(int axis,
      int endPosition, int velocity, int acceleration);
  virtual int GetPosition(int axis);
private:
  virtual void ForwardKinematics(const int servoVal[], int
   operVal[]);
  virtual void InverseKinematics(const int operVal[], int
   servoVal[]);
  int m_convNumerator[DIFF_AXES];
  int m_convDemonimator[DIFF_AXES];
};
CDiffBase::CDiffBase(int translationScale, int rotationScale, int
 controlFreq)
  : CAxesClosed(DIFF_AXES, DIFF_AXES)
{
// Initialize scaling constants
  m_convDenominator[TRANSLATE_AXIS] = translationScale;
   // (ticks/meter)
  m_convNumerator[TRANSLATE_AXIS] = 1000;  // (millimeter/meter)
  m_convDenominator[ROTATE_AXIS] = rotationScale;
   // (ticks/revolution)
  m_convNumerator[ROTATE_AXIS] = 6283;     // 2 × pi × 1000
   // (milliradians/revolution)
  m_controlFreq = controlFreq;
  SetGains(500, 0, 500);
}
void CDiffBase::ForwardKinematics(const int servoVal[], long
 operVal[])
{ // ">> 1" is equivalent to dividing by 2
  operVal[0] = (servoVal[0] + servoVal[1])>>1;    // Translation
  operVal[1] = (servoVal[1] - servoVal[0])>>1;    // Rotation;
}
void CDiffBase::InverseKinematics(const int operVal[], int
 servoVal[])
{
  servoVal[0] = operVal[0] + operVal[1];          // Left wheel
  servoVal[1] = operVal[0] - operVal[1];          // Right wheel
}

void CDiffBase::Move(int axis,
  int endPosition, int velocity, int acceleration);
{
// Convert endPosition, velocity, and acceleration to "ticks"
  endPosition = endPosition*m_convDenominator[axis]
   /m_convNumerator[axis];
  velocity = velocity*m_convDenominator[axis]*m_control
      Freq/m_convNumerator[axis];
  acceleration = acceleration*m_convDenominator[axis]*
     m_controlFreq*m_controlFreq/m_convNumerator[axis];

// Execute move command
   CAxesClosed::Move(axis, endPosition, velocity, acceleration);
}
```

After recompiling and running, grab the wheel, turn it a good half turn, and then release it. (Do this with the wheel off the ground!) This is called perturbing the control system, which is a simple way of determining the response of the PID control loop.

After releasing the motor, it should

at least attempt to return to its original position. Increasing the proportional gain causes the motor to return to its original position more quickly after being perturbed. Increasing the gain further will eventually result in the motor overshooting the original position and then oscillating back and forth. Continue to increase the proportional gain until the motor starts to oscillate in this manner after being perturbed.

Oscillation is more pronounced in motors with low friction or drag. Motors with large gear reduction tend to have more friction and oscillate less. The Lego motors have little friction, which makes them efficient but more challenging to control. To reduce the oscillation, you need to increase the derivative gain. The derivative gain adds velocity damping: the derivative of the error is equal in magnitude but opposite in sign to the motor velocity. Thus, the derivative term wants to slow down the motor in direct proportion to the motor velocity.

What else slows things down in proportion to velocity? Friction. Increasing the derivative gain is like adding friction to your motor, which may sound like something you should avoid. However, increasing the derivative gain does not adversely affect the efficiency of your motor as friction does. Go ahead and continue to increase the derivative gain until the oscillation is under control. If increasing the derivative gain does not remove the oscillation, the proportional gain may be too high. Try reducing the proportional gain, setting the derivative gain to zero, and starting over. We are shooting for a motor that returns to its original position as quickly as possible and then stops (not oscillates) after being perturbed. It's acceptable for the motor to overshoot the original position slightly before coming to rest.

After the oscillation is under control, you can choose to be satisfied and stop here, or you may want to see how far you can push things. If so, try increasing the proportional gain again, which will bring the oscillations back. Again, you can quell the oscillations by increasing the derivative gain. You can proceed in this manner until you reach a proportional gain that results in oscillations that cannot be sufficiently reduced by increasing the derivative gain. At this point, you've pushed things too far. Reduce the proportional gain and find a suitable derivative gain that produces the desired response after perturbing the motor.

Sometimes, friction or another disturbance prevents the motor from reaching its desired position. That's where the integral term comes in. It ensures that the error will always reach zero in the steady state. For example, if the motor comes to rest at a position that is relatively close to its desired position, the error is small. And, if the error is small, the proportional term may not provide enough PWM to move the motor the extra distance to reach its desired position. This is likely to happen if your motor has static friction, or if your robot's wheels have external forces acting on them, such as those from an incline or hill.

Because the integral term is integrating the error over time, a nonzero

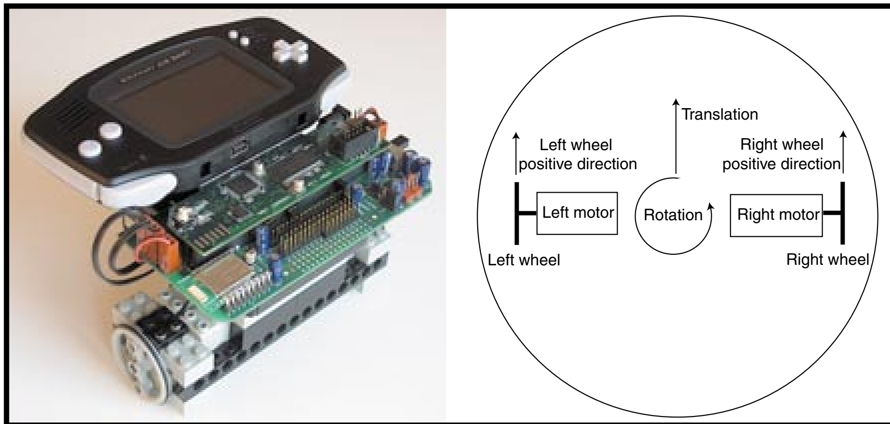

Photo 2—*The differential robot base is popular because of its simplicity. Only two motors are required. It's usually more intuitive to specify motion in terms of translation and rotation instead of left and right wheel motion.*

error causes the integral term to grow gradually over time. Eventually, the integral term will become large enough to overcome the external force (e.g., friction or an incline). The larger the integral gain, the faster it overcomes the external force. But bear in mind that a sufficiently large integral gain can introduce oscillations to your system. Because the Lego motors have little static friction, I choose to set the integral gain to zero. However, most control systems can benefit from a small amount of integral gain. It depends on how much steady-state error your motor typically experiences and how much you are willing to tolerate in your system.

As I mentioned before, a higher control frequency results in better control. A higher control frequency allows you to increase the PID gains (particularly the proportional gain) for a better, faster responding system. A control frequency of a few hundred hertz is usually sufficient, however. When changing the control frequency, be sure to retune PID gains to achieve the best response.

TESTING A SINGLE AXIS

After tuning your PID control loop, you can fire up the trajectory generator and move your motor. This entails calling `Move()` with your desired trajectory parameters. `Move()` starts the trajectory generator `TrapezoidTrajectory()`. If everything is working, the motor smoothly accelerates to the desired velocity, holds the desired velocity, and then decelerates and stops exactly at the desired position. Nice! You can play with the trajectory parameters until the motor moves to your satisfaction. Note that it is possible to specify velocity and acceleration values that are beyond your motor's capabilities. The source code for performing the test is posted on the *Circuit Cellar* ftp site.

You may have noticed that I have completely ignored units so far, which makes specifying reasonable trajectory values difficult. (What does a velocity of 1,000 actually mean?) I will remedy this in the next few sections. The lack of recognizable units in the lower levels of the control system saves a significant amount of computing bandwidth.

I've described how to implement a complete closed-loop PID position controller with trajectory generator for a set of motor axes. This alone gives you precise control of your robot's motors. I could stop here, but this is where things start getting interesting! The next sections will integrate the kinematics of the robot base into the control system. Kinematics makes the robot base easier and more intuitive to control.

DIFFERENTIAL ROBOT BASE

The differential base is the classic robot base and probably the most popular. It consists of two motors and drive wheels, as shown in Photo 2. Often, the base is circular and the wheels are mounted colinearly with the center of the base. This allows the robot to rotate around the center of the base, which simplifies path planning.

The differential base has two motors and two degrees of freedom. When thinking about its motion, you usually think of it moving forward, turning, moving backward, etc. In other words, you imagine the robot translating, rotating, or both. In general, describing the robot's motion in terms of translation and rotation is more intuitive than describing the motion of each wheel (e.g., right wheel clockwise, left wheel counterclockwise, etc.)

I call the degrees of freedom (or axes) with which I prefer to describe motion (translation and rotation) my operational space. The motors themselves (left and right wheels) make up the servo space, or alternatively the joint space when it applies to robot manipulators.[1]

The mathematical expressions I use when mapping from servo space to operational space are called the forward kinematics. The forward kinematics for the differential base are:

$$\text{translation} = \frac{r\left(\text{right_wheel} + \text{left_wheel}\right)}{2}$$

$$\text{rotation} = \frac{r\left(\text{right_wheel} - \text{left_wheel}\right)}{b}$$

where *r* is the wheel radius of both wheels, and *b* is the distance between them. Note that the *right_wheel*, *left_wheel*, and *rotation* variables are expressed in radians. The inverse of these expressions, which map operational space to servo space, is called the inverse kinematics:

$$\text{left_wheel} = \frac{\left(\text{translation} - \frac{b \times \text{rotation}}{2}\right)}{r}$$

$$\text{right_wheel} = \frac{\left(\frac{b \times \text{rotation}}{2} + \text{translation}\right)}{r}$$

Note that most robotics professionals and scholars are accustomed to seeing the kinematics expressions in matrix form instead of the expanded form shown here.

Now you're ready to implement a differential base controller class called `CDiffBase` (see Listing 4). Most importantly, note that `CDiffBase` is derived from `CAxesClosed` and overrides `InverseKinematics()` and `ForwardKinematics()` with its own versions. These versions contain the simplified differential base kinematics, which allow `CAxesClosed` to convert operational space axis positions into servo space axis positions and vice versa. As a result, `CDiffBase` now accepts motion commands (e.g., `Move()`) in terms of translation (axis index 0) and rotation (axis index 1), which make up its operational space.

Let's examine what is going on. Assume a working implementation of your base class `CAxesOpen` such that the left wheel is axis index 0 and the right wheel is axis index 1. Next in the hierarchy, `CAxesClosed` calls `InverseKinematics()` from within `Periodic()` (see Listing 2). `InverseKinematics()` takes operational space positions from the trajectory generator as inputs and outputs the corresponding servo space positions. The PID controller uses these positions to control the position of each motor. Note that almost all of the work is done within `CAxesClosed`. `CDiffBase` is simple by comparison.

WHERE ARE THE UNITS?

When commanding your robot base, specifying position, velocity, and acceleration with recognizable units is important. But what units should you choose? In the interest of saving computation, it's a good idea to express these units as integers instead of floating point values. Thus, the units need to provide reasonable resolution when expressed as integers. Considering this, I've chosen millimeters for translation units and milliradians for rotation units.

To perform unit conversion, `CDiffBase` overrides `Move()` and performs conversion before passing the arguments down to `CAxesClosed` (see Listing 4). Performing unit conversion in this manner is efficient because it occurs once per `Move()` command instead of once per control cycle.

Unit conversion for the differential base relies on two scaling constants (`translationScale` and `rotationScale`), which are passed into the `CDiffBase` constructor (see Listing 4). These scaling constants are specified in units that can be easily determined through calibration. For example, `translationScale` is specified in ticks per meter. Here, "ticks" are the units that are passed into `CAxesClosed::Move()` after unit conversion. What are ticks exactly? For the translation axis, they are units of distance; for the rotation axis, they are units of angular displacement. Calibrating your robot base will reveal the sizes of these units.

CALIBRATION

Calibration accurately determines the values of `translationScale` and `rotationScale`. Determining these constants through calibration is usually the method that yields the most accurate robot base.

There are lots of ways to calibrate your base, but the freewheeling method is probably the easiest. Run a program that continuously prints the positions of the operational space axes (translation and rotation) by calling `CAxesClosed::GetPosition()`. While running this program, record

the translation value that is being printed. Roll the robot in a straight line for 1 m, and record the translation value again. Subtracting the two values will yield the number of ticks per meter of translation, which is `translationScale`. Use the same method to determine `rotationScale`, except rotate the robot one revolution (360°) to determine the number of rotation ticks per revolution, which is `rotationScale`. You may download an implementation of the calibration program from the *Circuit Cellar* ftp site.

You may be wondering why `ForwardKinematics()` and `InverseKinematics()` in Listing 4 ignore *r* and *b*. Note that *r* and *b* only serve as scaling constants for translation and rotation, you can move them out of your kinematics calculations and into the scaling constants (`translationScale` and `rotationScale`). Because you determine your scaling constants through calibration, the values of *r* and *b* do not need to be explicitly known. That makes life easier!

AFFORDABLE HUMR

In the previous section, I covered implementing an operational space differential robot controller. Although a differential base is simple, it only has two degrees-of-freedom. This limitation, for example, makes it impossible to translate in a particular direction unless the wheels are oriented in that direction. In other words, a differential robot needs to turn or rotate before it can head in a particular direction.

A robot that is confined to a plane (e.g., the floor) has three degrees of freedom at most. These are commonly represented as two translation degrees of freedom (x and y) and rotation, as shown in Figure 3. In the field of mobile robotics, a robot with three degrees of freedom in a plane is considered holonomic. The differential base, for example, is lacking a degree of freedom and is considered nonholonomic.

Why is a holonomic robot useful? Holonomic robots can accelerate in any direction at any time, which makes them highly maneuverable and surprisingly easy to control. Consider

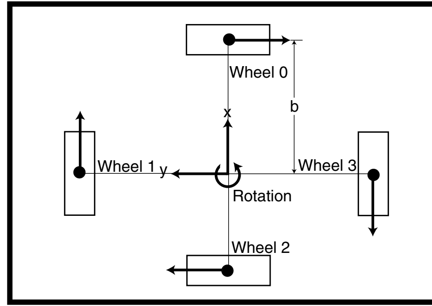

Figure 3—*Only three omniwheels are required for holonomic motion, but four omniwheels (shown here) provide more power and accuracy. When equipped with 9-V Lego motors and 4-cm (diameter) omniwheels, the HUMR has a top speed of 50 cm/s and a top acceleration of 200 cm/s*^{2}.

a simple example of moving to a specific goal location. A differential robot must first turn toward the goal before it moves toward it. A holonomic robot simply heads straight toward the goal (no turning required), which is simpler and typically faster.

There are many different holonomic robot designs, but probably the simplest consists of independently driven omnidirectional wheels (see Photo 1). Omnidirectional wheels, or "omniwheels," have passive rollers evenly distributed along the outside periphery of the wheel. The rollers provide the omniwheel with an extra degree of freedom that permits the wheel to move orthogonally (sideways) with respect to the regular wheel motion. I'll refer to this extra degree of freedom as the minor axis of the omniwheel. The major axis is the degree of freedom regularly associated with wheels. When a motor is attached to an omniwheel, the minor axis remains passive and the major axis becomes powered.

It is often the case that a robot will have as many motors as degrees of freedom. When the number of motor axes exceeds the number of degrees of freedom, the system is underconstrained. The Holonomic Underconstrained Mobile Robot (HUMR) has four omniwheels, which is one more than is necessary for holonomic motion (see Photo 1). The four-wheeled HUMR has the advantage of increased power and accuracy as well as simplified kinematics and Lego construction.

As with the differential robot base, the servo space of the HUMR provides an unintuitive operational space. But, more importantly, because the base is underconstrained, it's possible for the wheels to fight each other in a sort of tug-of-war. The operational space controller prevents this while providing intuitive control.

HUMR CLASS AND KINEMATICS

Figure 3 shows the four wheels and motors of the HUMR base, which are numbered zero through three. The arrows indicate the positive motion direction of each motor. The inverse kinematics, which is fairly easy to derive using vector math, provide the motion constraints required for our underconstrained system. Deriving the forward kinematics may seem tricky because there are four wheels (knowns) and three operational axes (unknowns), but the HUMR's simple right-angle geometry makes it possible to derive these expressions by inspection. Forward kinematics are as follows:

$$x = \frac{r(\text{wheel } 1 - \text{wheel } 3)}{2}$$

$$y = \frac{r(-\text{wheel } 0 + \text{wheel } 2)}{2}$$

$$\text{rotation} = \frac{-r(\text{wheel } 0 + \text{wheel } 1 + \text{wheel } 2 + \text{wheel } 3)}{4b}$$

Inverse kinematics are as follows:

$$\text{wheel } 0 = \frac{-(y + b \times \text{rotation})}{r}$$

$$\text{wheel } 1 = \frac{(x - b \times \text{rotation})}{r}$$

$$\text{wheel } 2 = \frac{(y - b \times \text{rotation})}{r}$$

$$\text{wheel } 3 = \frac{-(x + b \times \text{rotation})}{r}$$

where *r* is the wheel radius, and *b* is the distance from any wheel to the center of the robot.

The implementation of HUMR class `CHumrBase` is similar to `CDiffBase`. `CHumrBase`, like its sibling, only requires two scaling constants: one for the translation axes (`translationScale`) and one for the rotation axis (`rotationScale`). As before, you can move the *r* and *b* terms out of the kinematics expres-

sions and into the scaling constants to minimize the computation (see Listing 5). Furthermore, I recommend that the scaling constants be determined through the freewheeling calibration process.

EXPENSIVE FRISBEE

One of the more impressive results of holonomic motion is the ability to rotate while translating in a straight line. I call this Frisbee motion because it resembles the motion of, well, a Frisbee. However, if your robot simply translates along its x-axis while rotating, for example, it will go in a circle, not in a straight line. This is because the x- and y-axes are always pointing in the same direction with respect to the robot. When the robot rotates, so do the axes. In order to move in a straight line, you need to rotate the translation axes in the opposite direction with respect to the robot's rotation. Rotating the x- and y-axes entails a simple application of sine and cosine with respect to the robot's

Listing 5—*The `InverseKinematics()` routine implements Frisbee mode by rotating the x- and y-axes. By embedding different mathematical expressions in `InverseKinematics()`, all sorts of different and interesting modes are possible.*

```
void CHumrBase::ForwardKinematics(const int servoVal[], int
 operVal[])
{
   operVal[X_AXIS] = ( servoVal[1] - servoVal[3])>>1;
   operVal[Y_AXIS] = (-servoVal[0] + servoVal[2])>>1;
   operVal[ROTATE_AXIS] =
     (-servoVal[0] - servoVal[1] - servoVal[2] - servoVal[3])>>2;
}
void CHumrBase::InverseKinematics(const int operVal[], int
 servoVal[])
{
   int cosRotation, sinRotation, rotation;
   int diffRotOperVal[2], diffOperVal[2];
   if (m_frisbee)
   {
     rotation = GetPosition(ROTATE_AXIS);  // Get rotation angle
     cosRotation = CosLut(rotation);
     sinRotation = SinLut(rotation);
// Calculate velocity (compute difference)
     diffOperVal[X_AXIS] = operVal[X_AXIS] -
      m_prevOperVal[X_AXIS];
     diffOperVal[Y_AXIS] = operVal[Y_AXIS] -
      m_prevOperVal[Y_AXIS];
// Rotate x and y axes and scale down by shifting right by 10
     diffRotOperVal[X_AXIS] = (diffOperVal[X_AXIS]*cosRotation +
        diffOperVal[Y_AXIS]*sinRotation)>>10;
     diffRotOperVal[Y_AXIS] = (-diffOperVal[X_AXIS]*sinRotation +
        diffOperVal[Y_AXIS]*cosRotation)>>10;
// Add to new position
     m_newOperVal[X_AXIS] += diffRotOperVal[X_AXIS];
     m_newOperVal[Y_AXIS] += diffRotOperVal[Y_AXIS];
// Save for next iterration
     m_prevOperVal[X_AXIS] = operVal[X_AXIS];
     m_prevOperVal[Y_AXIS] = operVal[Y_AXIS];
   }
   else
   {
     m_newOperVal[X_AXIS] = operVal[0];
     m_newOperVal[X_AXIS] = operVal[1];
   }
   servoVal[0] = -m_newOperVal[Y_AXIS] - operVal[ROTATE_AXIS];
    // Wheel 0
   servoVal[1] =  m_newOperVal[X_AXIS] - operVal[ROTATE_AXIS];
    // Wheel 1
   servoVal[2] =  m_newOperVal[Y_AXIS] - operVal[ROTATE_AXIS];
    // Wheel 2
   servoVal[3] = -m_newOperVal[X_AXIS] - operVal[ROTATE_AXIS];
    // Wheel 3
}
```

rotation angle:

$$x' = \cos(\text{rotation}) \times x + \sin(\text{rotation}) \times y$$
$$y' = -\sin(\text{rotation}) \times x + \cos(\text{rotation}) \times y$$

Implementing this functionality can be easily accomplished by embedding these expressions in the `InverseKinematics()` function (see Listing 5). Note that `ForwardKinematics` doesn't need to be changed for now because it doesn't affect the HUMR's motion. I've introduced a new variable called `m_frisbee` into `InverseKinematics`, which, when set, rotates the x- and y-axes with respect to the rotation angle and enables Frisbee motion. It assumes that you have a reasonably efficient means of obtaining the sine and cosine of the rotation angle, which is specified in milliradians. The `CosLut()` and `SinLut()` functions employ a look-up table of sine and cosine values scaled (multiplied) by 1,024 to make integer multiplies accurate. Scaling the results back down (dividing by 1024) is accomplished by right-shifting the results by 10 bits. The implementations of `SinLut()`, `CosLut()`, the rest of `CHumrBase`, and the video clips of Frisbee motion are posted on the *Circuit Cellar* ftp site.

Your implementation of Frisbee mode allows you to change the rotation angle however you wish without affecting the direction of motion. Aside from being cool to look at, Frisbee mode can be useful when a sensor on top of the robot needs to pan back and forth, for example. I'm sure you can think of other modes that are useful for whatever you want your robot to accomplish. Or, you may want to experiment with a different operational space. Implementing these ideas only requires modifying the kinematics routines.

IMPROVED PERFORMANCE

I've covered quite a bit a ground in this article: PID control, tuning, trajectory generation, and operational space control for two different robot bases. I have implemented a class hierarchy that makes constructing closed-loop operational space controllers straightforward and applicable to practically any robot base. And I have accomplished this with software that consumes only a modest amount of computing power. For your next robot or motion-control project, keep these ideas and methods in mind. Your robot will thank you with greatly improved performance! ■

Rich LeGrand has 10 years of experience in robotics and multimedia systems. When he's not tuning PID loops by hand, he works at Charmed Labs, which provides advanced embedded technologies for consumer and educational use. He holds a B.S.E.E. and B.A.C.S. from Rice University and an M.S.E.E. from North Carolina State University. Rich has authored domestic and international patents for holonomic robot control. You can reach him at rich@charmedlabs.com.

PROJECT FILES

To download the code, go to ftp.circuitcellar.com/pub/Circuit_Cellar/2004/169.

REFERENCE

[1] O. Khatib, "A Unified Approach to Motion and Force Control of Robot Manipulators: The Operational Space Formulation," *IEEE Journal Robotics and Automation*, RA3, 1987.

RESOURCE

Vector math, www-2.cs.cmu.edu/~pprk/physics.html.

SOURCES

Omnidirectional wheels
Acroname, Inc.
(720) 564-0373
www.acroname.com

Xport 2.0 and Xport Robot Controller
Charmed Labs
(201) 444-7327
www.charmedlabs.com

Mindstorms Robotics Invention System
Lego Co.
+45 79506070
www.lego.com

Gameboy Advance and Gameboy Advance SP
Nintendo of America, Inc.
(800) 255-3700
www.nintendo.com