CSCI–4190 Introduction to Robotic Algorithms, Spring 2003
Lab 1: out Thursday January 16, to be completed by Thursday January 30

Following a path

For this lab, you will learn the basic procedures for using the robot and will program it to follow a specified path. This first lab will be "supervised," meaning that a TA will be in the lab to get you started, answer any questions, and to verify that you have completed the lab.

Here are the steps you must take to complete this lab:

- Sign up for a one hour time slot with your lab partner.

  Go to `http://calendar.yahoo.com`, and log in as `robotalgorithms`. Choose one of the possible time slots and edit the appointment title to put your names there.

- *Before* coming to the lab, read this handout completely, write a program to solve the task, and make sure it compiles!

  Because your time for this lab will be limited, it is important that you come to the lab prepared. Your time in the lab should be spent fine tuning your program, not struggling to get your program working!

- Come to the lab (Amos Eaton 106) at your appointed time. The TA will check your prepared program (see above), review basic procedures for using the robot, and get you started on the lab exercise. A short time before the end of the hour, you will demonstrate your program.

You will get full credit for attending and completing the lab exercise.

The task

You will be given the parameters of a path which consists of straight line segments and circular arcs:

- For the straight line segments, you will be given the distance (in meters) to be traveled.

- For the circular arcs, you will be given the coordinate of the center of rotation (in the robot coordinate frame in meters) and the angle of the arc (in degrees). The robot frame has its origin at the center of the robot, the positive y-axis points forwards, and the positive x-axis points to the right. Since the center of rotation must lie on this x-axis, only the x coordinate needs to be specified. Since the robot will always be going forwards, positive values result in arcs turning right, negative values in arcs turning left.

Your program should make the robot follow this path.

Notes

- For this lab, do not set the velocity of the robot greater than 0.5 m/s. In fact, 0.1 m/s should be fine; this is *not* a race!

- At least one path (the same one as in Exercise 1) will be marked on the floor with tape. Your program should make the robot follow the path closely and end up near the end of the specified path. We will measure how close the robot is to the goal configuration, and visually estimate how closely the robot follows the specified path.

- Your code should be easily modifiable to follow a different path (with possibly a different number of segments), given the measurements. It is acceptable to hard-code the (number of) segments and their parameters in your `main()` program, but the functions that actually move the robot along the path should take the parameters as arguments.
- Remember that when you command a change in velocity, the velocity does not change instantaneously. It takes some amount of time to accelerate or decelerate to the new velocity.
- You should be able to complete this task using only motion commands (i.e., without using odometry feedback) and `sleep()` function calls. You will find that the robot will not be able to execute motions exactly (which is part of the point of this lab).

  If you want to try using odometry feedback, you may do so (after demonstrating your program without feedback) for a small amount of extra credit.
- Your functions will probably need internal parameters to adjust how the robot executes a path segment. These parameters might, for example, compensate for the distance covered while the robot accelerates or decelerates.

An introduction to the Magellan Pro robot

The Magellan Pro is a commercial robot made by the iRobot corporation. The robot is equipped with an on-board PC running Red Hat Linux. It is 40.6cm in diameter and 25.4cm tall and has 16 SONAR sensors, 16 IR sensors, and 16 bump switches. Limited operation and diagnostics can be performed using the tuning knob and LCD screen on the top of the robot. It can be driven using a joystick or controlled by programs written with the *Mobility Robot Integration Software*.

The Mobility software is a distributed, object-oriented toolkit for building control software for single and multi-robot systems. Mobility is built on the Common Object Request Broker Architecture (CORBA) 2.X standard Interface Definition Language (IDL). However, we have written a simple library that isolates you from all the details of the CORBA interface. The CORBA foundation allows your programs to run on a desktop computer (`maximal.cs.rpi.edu`) and operate the robot by accessing the CORBA objects (running on the robot) that encapsulate the robot interface.

Running a program on the robot

Once you have a written and compiled your program, to run it on the robot you must do the following:

1. Turn the robot on. Wait for it to boot.
2. Log into `maximal.robotics.cs.rpi.edu` and start the base server.

   ```
   $ robotstart
   ```

   The SONAR sensors will start making clicking sounds.
3. Run your program. For example:

   ```
   $./myProgram
   ```
4. When your program has finished running, stop the base server:

   ```
   $ robotstop
   ```

   The SONAR sensors will stop firing.

Robot programming interface

The Mobility Robot Object Model connects to different objects within the robot structure to give programmers a very general beginning. A lot of the complications have been removed for you through the implementation of a simple `Robot` class. This section describes the methods available to you through this class.

- `Robot(int argc, char *argv[], bool verbose=false)` — constructor for the robot class. `argc` and `argv` should just be those passed to `main`; the `Robot` class will extract the robot name from the commandline if it is specified. If `verbose` is `true`, lots of debugging information will be printed.
- `void forward(float dist, float v)` — drives a given distance (in meters) forward using a trapezoidal velocity profile with a "cruising" velocity v.
- `void turn(float theta)` — turns the robot the given angle (in radians). Counterclockwise turns are positive; clockwise turns are negative. Also uses a trapezoidal velocity profile.
- `void setVelocity(float v, float w)` — sets forward velocity `v` (in m/s) and angular velocity `w` (in radians/s). Setting both velocities to zero will stop the robot.

  The robot has a "watchdog timer" which will stop the robot if its velocity commands are not "refreshed." The `forward()` and `turn()` functions do this internally, but if you set velocities yourself, you must periodically set them again to keep moving at that velocity. See the sample programs for examples.
- `void setMaxVelocity(float v)` — the maximum allowable velocity (in m/s) for trapezoidal velocity profiles.
- `float getMaxVelocity()` — returns the maximum velocity (in m/s) for trapezoidal velocity profiles.
- `void setAcceleration(float a)` — sets the acceleration (in m/s^2) that will be used in trapezoidal velocity profiles.
- `float getAcceleration()` — returns the acceleration (in m/s^2) for trapezoidal velocity profiles.
- `void getOdometry(float &x, float &y, float &theta)` — retrieves the location (`x` and `y` in meters) and angle (in radians) of the robot relative to its starting location, with the x-axis pointing to the robot's right and the y-axis pointing forward.
- `MobilityGeometry::SegmentData_var getSonar()` — returns a pointer to the mobility object containing SONAR data. SONAR data is returned as a set of 16 line segments in the world coordinates. Each segment starts at the appropriate sensor, extends in the direction that the sensor faces, and is as long as the measured range. See the sample programs for examples.
- `MobilityGeometry::SegmentData_var getIR()` — returns a pointer to the mobility object containing IR data. Like the SONAR data, IR data is returned as a set of 16 line segments; see the sample programs for examples.

Source code

You can find the source code for the `Robot` class in the directory:

`/usr/local/magellan/src`

A Sample Program

This is a simple program that moves the robot forward until the infrared sensors detect that it is within 0.5 m from a wall.

The robot class that implements your simple robot interface is defined in robot.h. Include this in all of your programs for the lab.

```
#include "robot.h"
#include <iostream>
#include <math.h>
```

Pass the command line arguments to the `Robot` constructor. In this case they won't actually be used for anything, but in some cases it is useful to specify the name of the robot to talk to on the command line. (You can do this with `$ ./myProgram -robot MagellanPro`)

```
int main( int argc, char **argv )
{
  Robot R( argc, argv );
```

The program sets up the variables for determining distance from the wall and a storage point for infrared sensor data.

```
  // Now, here is a loop that continually checks robot sensors and
  // sends drive commands
  MobilityGeometry::SegmentData_var irData;
  float haltdist = 0.5;    // 50cm halts
  float tempdist;          // Computed temp dist value
  float x, y, theta;

  cout << "********** Mobility IR Example ***********" << endl;
```

The main program loop. Set the robot to go, check the sensors, stop if appropriate. The program will stop under two conditions: The robot is within 0.5m from a wall or the user has requested a stop by hitting the enter key.

```
  while( 1 ) {
    // forward velocity, no rotational velocity
    R.setVelocity( 0.1, 0.0 );

    // grab the sonar data
    irData = R.getIR();

    tempdist = sqrt(
        (irData->org[0].x - irData->end[0].x) *
        (irData->org[0].x - irData->end[0].x) +
        (irData->org[0].y - irData->end[0].y) *
        (irData->org[0].y - irData->end[0].y));

    cout << "dist: " << tempdist << endl;

    if( tempdist < haltdist )
      break;

    // if the user pressed return
    if( mbyUtility::chars_ready() > 0 ) {
      cout << "stopped by user" << endl;
      break;
```

```
    }

    // Wait a small time (0.1 sec) before iterating
    omni_thread::sleep( 0, 100000000 );
  }

  R.setVelocity( 0.0, 0.0 );       // stop the robot
  R.getOdometry( x, y, theta );
  cout << "X: " << x << endl;
  cout << "Y: " << y << endl;
  cout << "theta: " << theta << endl;
  return 0;
}
```

Notice that in this program we didn't use the built-in `forward` command. This is because `forward` doesn't allow us to continually check incoming sensor information; it merely sends the robot blindly forward for the specified distance.

Another Sample Program

Following is an even simpler program that demonstrates moving the robot along a curved path until the user presses return. Notice that both the translational and rotational velocities are set.

```
#include "robot.h"
#include <iostream>
#include <math.h>

int main( int argc, char **argv )
{
  Robot R( argc, argv );

  while( 1 ) {
    // forward velocity AND rotational velocity
    R.setVelocity( 0.1, M_PI / 8.0 );

    // if the user pressed return
    if( mbyUtility::chars_ready() > 0 ) {
      cout << "stopped by user" << endl;
      break;
    }

    // Wait a small time (0.1 sec) before iterating
    omni_thread::sleep( 0, 100000000 );
  }

  R.setVelocity( 0.0, 0.0 );    // stop the robot
  return 0;
}
```