Now we consider specifying each remaining function in detail. Consider for our purposes a robot with a ring of sonars placed radially around the robot. This imagined robot will be differential-drive, so that the sonar ring has a clear "front" (aligned with the forward direction of the robot). Furthermore, the robot accepts motion commands of the form shown above, with a rotational velocity parameter and a translational velocity parameter. Mapping these two parameters to individual wheel speeds for each of the two differential-drive chassis' drive wheels is a simple matter.

There is one condition we must define in terms of the robot's sonar readings, `ObstaclesInWay()`. We define this function to be true whenever any sonar range reading in the direction of the goal (within 45 degrees of the goal direction) is short:

```
private boolean ObstaclesInWay(angle goalAngle, sensorvals sonars) {
    int minSonarValue;
    minSonarValue=MinRange(sonars, goalAngle
                        -(pi/4),goalAngle+(pi/4));
    return (minSonarValue < 200);
} // end ObstaclesInWay() //
```

Note that the function `ComputeTranslation()` computes translational speed whether the robot is wall-following or heading toward the goal. In this simplified example, we define translation speed as being proportional to the largest range readings in the robot's approximate forward direction:

```
private int ComputeTranslation(sensorvals sonars) {
    int minSonarFront;
    minSonarFront = MinRange(sonars, -pi/4.0, pi/4.0);
    if (minSonarFront < 200) return 0;
    else return (Math.min(500, minSonarFront - 200));
} // end ComputeTranslation() //
```

There is a marked similarity between this approach and the potential field approach described in section 6.3.2. Indeed, some mobile robots implement obstacle avoidance by treating the current range readings of the robot as force vectors, simply carrying out vector addition to determine the direction of travel and speed. Alternatively, many will consider short-range readings to be repulsive forces, again engaging in vector addition to determine an overall motion command for the robot.

When faced with range sensor data, a popular way of determining rotation direction and speed is to simply subtract left and right range readings of the robot. The larger the difference, the faster the robot will turn in the direction of the longer range readings. The following two rotation functions could be used for our Bug2 implementation:

```
private int ComputeGoalSeekRot(angle goalAngle) {
    if (Math.abs(goalAngle) < pi/10) return 0;
    else return (goalAngle * 100);
} // end ComputeGoalSeekRot() //

private int ComputeRWFRot(sensorvals sonars) {
    int minLeft, minRight, desiredTurn;
    minRight = MinRange(sonars, -pi/2, 0);
    minLeft = MinRange(sonars, 0, pi/2);
    if (Math.max(minRight,minLeft) < 200) return (400);
                                        // hard left turn
    else {
        desiredTurn = (400 - minRight) * 2;
        desiredTurn = Math.inttorange(-400, desiredTurn, 400);
        return desiredTurn;
    } // end else
} // end ComputeRWFRot() //
```

Note that the rotation function for the case of right wall following combines a general avoidance of obstacles with a bias to turn right when there is open space on the right, thereby staying close to the obstacle's contour. This solution is certainly not the best solution for implementation of Bug2. For example, the wall follower could do a far better job by mapping the contour locally and using a PID control loop to achieve and maintain a specific distance from the contour during the right wall following action.

Although such simple obstacle avoidance algorithms are often used in simple mobile robots, they have numerous shortcomings. For example, the Bug2 approach does not take into account robot kinematics, which can be especially important with nonholonomic robots. Furthermore, since only the most recent sensor values are used, sensor noise can have a serious impact on real-world performance. The following obstacle avoidance techniques are designed to overcome one or more of these limitations.

6.4.2 Vector field histogram

Borenstein, together with Koren, developed the vector field histogram (VFH) [77]. Their previous work, which was concentrated on potential fields [176], was abandoned due to the method's instability and inability to pass through narrow passages. Later, Borenstein, together with Ulrich, extended the VFH algorithm to yield VFH+ [323] and VFH*[322].

One of the central criticisms of Bug-type algorithms is that the robot's behavior at each instant is generally a function of only its most recent sensor readings. This can lead to undesirable and yet preventable problems in cases where the robot's instantaneous sensor readings do not provide enough information for robust obstacle avoidance. The VFH techniques overcome this limitation by creating a local map of the environment around the robot. This local map is a small occupancy grid, as described in section 5.7 populated only by relatively