MOTION COORDINATION

Orestis Akribopoulos
Research Academic Computer Technology Institute
Patras, Greece
akribopo@cti.gr

Andreas Cord-Landwehr
Heinz-Nixdorf-Institute & Department of Computer Science,
University of Paderborn
Paderborn, Germany
cola@uni-paderborn.de

Henning Hasemann
Braunschweig University of Technology
Braunschweig, Germany
hasemann@ibr.cs.tu-bs.de

Barbara Kempkes
Heinz-Nixdorf-Institute & Department of Computer Science,
University of Paderborn
Paderborn, Germany
barbaras@uni-paderborn.de

1. Introduction

This chapter covers the motion functionality of the Roomba and Moway devices. We begin the chapter with explaining the design and the interfaces of the roomba backend and continue by a small introduction on how these interfaces can be used in applications. Further, we introduce the Moway Robots, which are small-size programmable robots. We conclude this chapter with a short introduction into the theoretical field of motion algorithms and give hints concerning currently ongoing work at the roomba backend.

2. Wiselib Roomba Backend

Some experiments were conducted using a mobile robot swarm, based on *iRobot Roomba 500 Series* vacuum cleaner robots. The robots have a disk-like shape with a diameter of 33cm and a height of 8cm. Motion control is provided by two independently suspended wheels, located on the robots diameter. They allow it to move forwards or backwards, and to turn with arbitrary curvature as well as in-place. All Roombas are equipped with six infrared sensors for sensing the proximity of physical obstacles as well as three bump sensors enabling them to detect physical contact to walls or each other. Each robot can track its pose using an integrated odometer. However this is subject to cumulative error, and therefore generally non suited for long-term absolute pose calculation.

The Wiselib Roomba interface consists of a set of concepts and templated classes. The Roomba itself is modeled as a `Roomba` class which provides access to some basic movement and sensor query operations. It does so by interfacing with an instance of the Serial Communication Wiselib concept for exchanging commands and sensor data with the robot. The protocol for this communication is defined by the *iRobot Roomba Open Interface Specification* [iRo].

Before a `Roomba` instance can be used it thus must be initialized with a reference to a serial communication

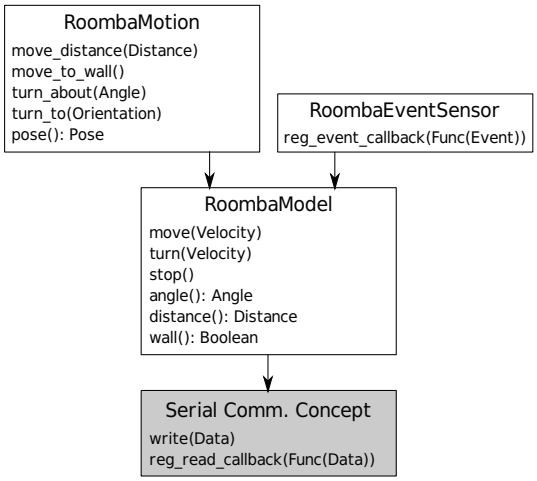


Figure 1: General structure of the Roomba Backend components

object. Additionally, the user can provide a number of flags in order to control which sensor data will be monitored by the interface. After initialization, the `move(...)` , `turn(...)` and `stop()` methods can be used for starting and stopping motion at any desired velocities. Additionally, the call operator and the `wall()` method will return the monitored sensor data in the format specified by the Open Interface Specification, whereas the `state()` and the `reg_new_data_callback(...)` methods provide information on the availability of (new) sensor data. For example, consider the following piece of code (template parameters are left out for readability):

```
1  roomba.init(
2    uart,
3    RoombaModel::WALL | RoombaModel::POSITION
4  );
5
6  // Start moving with 200 mm/s.
7  roomba.move(200);
8
9  // Here you would probably want to wait some time.
10
11 // Now stop motion.
12 roomba.stop();
13
14 if(roomba.wall()) // is the wall bumper being pressed?
15   doSomething();
```

As you can see from this example, controlling a robot this way can become a little tedious for more complex movement patterns. With this approach one would write lots of code that is only concerned with waiting for movements and interruptions of those wait phases in case events like wall contact happen. The reason for this is that the `Roomba` class was specifically designed to reflect the Roomba interface with as little abstraction as possible. In order to be able to do more complex things more easily we provide - based on this simple interface - two "rich interfaces" for the Roomba: The `RoombaMotion` and `RoombaEventSensor` classes.

The `RoombaMotion` class enhances the motion capabilities of the Roomba by incorporating odometry and sensor information directly into its motion methods. This way can provide methods for moving a certain distance, turning about a given angle or moving until an event such as physical contact with a wall is detected. Last but not least, the class tracks the robots odometer output to estimate its current position and orientation (or "pose") which can be queried at all times [1]. The usage of this class is simple. First create an instance of `Roomba` as described above, then initialize a `RoombaMotion` with a reference to it:

```
1  roomba_motion.init(roomba);
2
```

[1] Note however that due to rounding that is done by the robot before data transmission this is very prone to error accumulation

```
3   // Move 2m.
4   roomba_motion.move_distance(2000);
5
6   // Turn 90 degrees.
7   roomba_motion.turn_about(RoombaAngle(Math::PI / 2.0));
8
9   // Move until you hit an obstacle.
10  roomba_motion.move_to_wall();
11
12  // Print out the current estimated position.
13  cout << roomba_motion.pose().position.x << ","
14       << roomba_motion.pose().position.y << endl;
```

Note that in contrast to the `Roomba` class all `turn_...` and `move_...` calls wait for the motion to finish before returning.

The `RoombaEventSensor` on the other hand allows for elegant and lightweight access to events. While the `Roomba` continuously delivers new sensor data, the user is typically only interested in changes in those data. `RoombaEventSensor` provides an easy way to be informed when such a change occurs via a callback. The current implementation allows solely for detection of "bumper pressed" (i.e. wall hit) events, but is easily extendable to arbitrary other sensor-data based events as well.

3. Roomba Motion

Delivery of messages between unconnected networks was established in FRONTS by robots (cf. End-to-End Communication Chapter). Those robots transport undeliverable messages over communication gaps between the networks. Here, we exemplarily describe the basic algorithms and implementation techniques used for the robot motions as well as the interaction with the Roomba backend. We also give a hint, how to easily implement further and more elaborated movement patterns than those shipped with the current Wiselib library. For this section we assume that the Roomba's motion is controlled by a connected PC (`OsModel`) on which the program is executed.

The current exemplary implementation of robot motion patterns is given by the class `RoombaMovement`. The class provides the two movement patterns *straight line walk* and *random walk*:

1 The *straight line* movement pattern is essentially defined by the two parameters speed and period time. With this, the Roomba moves at a straight line at the given speed for the given time. After this operation, it moves at same speed and for the same time period backwards. This process is repeated until the `stop()` function is called or the Roomba hits a wall with its front bumper.

2 The *random walk* movement pattern is essentially defined by a speed parameter and constants that define a minimal and maximal turn angle as well as minimal and maximal movement time. For one step the Roomba uniformly at random chooses a turn angle at which it turns. After the turn it chooses uniformly at random a movement time in the given range and moves that time at the given speed. If the Roomba accidentally hits a wall (which is noticed by the front bumper) it just moves the forward-moved distance backwards.

In order to use the `RoombaMovement` class, it is necessary to call the `init()` method for initialization, one of the methods `set_movement_pattern_to_line()` and `set_movement_pattern_to_random_walk()` to setup the movement-pattern and to start the movement by calling `start(int speed)` with a given speed value. An example of the initialization procedure looks as the following:

```
1   // Initialize roomba movement
2   roomba_movement.init(
3     *roomba,   // Roomba_P
4     0,         // OsModel_P::Radio
5     *debug,    // OsModel_P::Debug
6     *my_clock, // OsModel_P::Clock
7     *timer,    // OsModel_P::Timer
8     *my_rand   // OsModel_P::Rand
9   );
10
11  // Choose the line movement pattern.
12  // Parameter gives the time (in ms) the roomba walks in one direction.
```

```
13  roomba_movement.set_movement_pattern_to_line( 15000 );
14
15  // Starts the movement of the roomba.
16  // Parameter gives the speed of the roomba (0-500).
17  // The roomba can also be started over radio.
18  roomba_movement.start( 300 );
```

On implementation terms, the `RoombaMovement` class utilizes the `RoombaMotion` class as a backend to set speed, turns, and movements of the Roomba device. The only interesting point about this implementation is that movements are established by timer callbacks. Hence, in later versions of the class it will be possible to go away from the PC-platform for controlling the Roomba to more low-cost hardware. Exemplary, we illustrate the interaction with the move operation of the `RoombaMotion` class.

```
1   // Setting the roomba motion to a given speed
2   void move( int16_t speed ) {
3     dest_speed_ = speed;
4     changing_speed_ = true;
5     move_callback( 0 );       // setting a callback with no userdata
6                         // callback periode is given by a constant
7   }
8
9   // setting the roomba motion to zero
10  void stop_movement() {
11    roomba_motion_.move( 0 );
12    cur_speed_ = 0;
13    dest_speed_ = 0;
14    changing_speed_ = false;
15  }
```

These two methods (in case of movement) can be used to firstly set a given movement speed and then register a callback that after a specified time period calls the `stop_movement()` method and by this ends the movement. This programming pattern can be found throughout the class design and essentially all movement-classes follow this design.

4. SunSPOT and Moway Robots

Moway is a small-size programmable robot, which is programmed from a PC using MOWAYGUI a software tool based on flow charts– or using the programming languages C or Assembly. Moway is mainly designed to perform Practical minirobotics applications. It features a drive system with the following controls: speed control, time control, traveled distance control, general speedometer and angle control. A SunSPOT has been mounted on top of each robot in order to execute our algorithms and to enable the communication with the surrounding sensor network.

The communication between SunSPOT and Moway is established using the SunSPOT's I/O connector in conjunction with Moway's expansion connector. SunSPOT's I/O connector provides the General Purpose digital I/O lines, which correspond to pins D0 through D4, and the the high current output pins, H0-H3. Moway's expansion connector has 8 pins: PIN3-PIN8 can be configured either as inputs or outputs, PIN1 is V_{cc} and PIN2 is GND. We have connected SunSPOT's digital output pins D0-D4 and H0, to pins 3-8 of the Moway expansion connector respectively. We have also connected SunSPOT's GND to expansion connector's GND. The 6 SunSPOT's digital pins can represent 12 different states, as a result Moway is able to execute 12 different operations. On Table 1, the mapping between SunSPOT's digital pins configurations and Moway instructions is described.

In order to control the motion of the Moway Robot, SunSPOT sets to high state the corresponding pins. Moway repeatedly reads the expansion connector's pins and executes the corresponding instruction. In this way, the functionality of the Moway is exposed to SunSPOT, where the implementation of movement patterns is easier.

Currently, there are two motion modes implemented on SunSPOT, the *Remote Control* and the *Random Walk*:

1 *Remote Control*: In this approach, we are able to remotely instruct the Moway robot to move to a desired location. A listening thread is executed on SunSPOT which receives remote commands over 802.15.4 packets. All possible commands are summarized on Table 1. The user defines the kind of movement, while in the case of a forward movement the corresponding duration as well.

SunSPOT's pins states						Moway
D0	D1	D2	D3	D4	H0	**Operation**
0	0	0	0	0	0	Stop
1	0	0	1	0	0	Forward
1	0	0	1	1	0	Slow Forward
1	0	0	1	1	1	Fast Forward
0	1	1	0	0	0	Right turn 30^o
0	1	1	1	0	0	Right turn 60^o
0	1	1	1	1	0	Right turn 90^o
0	1	1	1	1	1	Right turn 120^o
0	0	1	0	0	0	Left turn 30^o
0	0	1	1	0	0	Left turn 60^o
0	0	1	1	1	0	Left turn 90^o
0	0	1	1	1	1	Left turn 120^o

Table 1: Mapping of SunSPOT's pins configurations to Moway instructions.

2 *Random walk*: The random walk pattern is similar to the Roomba movement pattern which is described on section 8.3. Initially, the specific movement pattern chooses randomly one of the possible forward movements as well as the movement duration. The movement duration is within the minimal and maximal predefined constants. After the forward movement, it chooses one of the possible turn movements and then another forward movement.

If the Moway robot detects an obstacle, either in Remote Control or in Random Walk mode, automatically executes a 45^o clockwise or counterclockwise turn depending on the position of the obstacle.

5. Motion Algorithms

This section gives an overview over the different kinds of robot motion algorithms. Most of these algorithms were not implemented so far, as more sensoric information would be necessary to execute them. Hence, this section provides ideas for future implementations of motion algorithms for robot swarms.

Section 8.3 dealt with robots which were used to transmit messages between immobile sensors. Due to very limited sensoric information about their surrounding, the robots could not do any better than performing a random walk on the area between the sensors and transmit messages whenever possible. This section is dedicated to algorithms which need more sensoric input, but are ,on the other hand, better suited for transmitting messages between two sensors.

Like in Section 8.3, consider two sensors placed in a plane terrain, which want to exchange messages. As they are placed too far away from each other, they cannot communicate directly. Mobile robots are used to transmit the messages in the following way: In the beginning, they form a chain between the sensors, such that every robot can communicate with its two neighbors. This chain, however, may be arbitrarily long and winding. It may even intersect itself. The sensors can use this robotic chain to exchange messages, but the more messages are sent, the more energy is wasted for the long distances over which the messages are transmitted. The goal is, therefore, to let the robots move as close to the line between the sensors as possible. Figure 2 depicts the situation.

If the robots and sensors were equipped with a precise GPS, the sensors could exchange their positions and each robot could compute a target position on the line between the sensors. Using its GPS, it could move to this target position and the goal would be achieved. However, the robots are able to move to the line between the sensors even without GPS and without communication. The only information that they need is the relative position of their neighbors to their own position. A strategy for the movement of the robots, called Go-To-The-Middle [DKLM06], which achieves this goal is indicated in Figure 3(a). The robots act in discrete, synchronous time steps. In each time step, each robot observes the positions of its two neighbors relative to its own position and then moves to the middle between the neighbors.

It can be shown that using Go-To-The-Middle, the robots converge to positions which are evenly distributed on the line between the sensors. In the worst case, it takes $\Theta(n^2 \log n)$ time steps until the robots are in distance at most 1 from the positions they converge to.

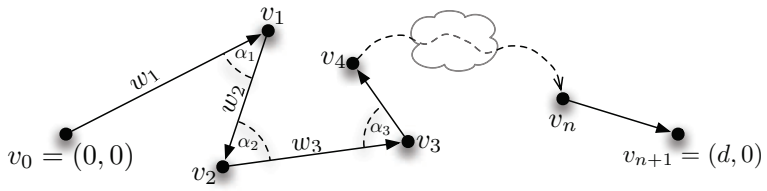
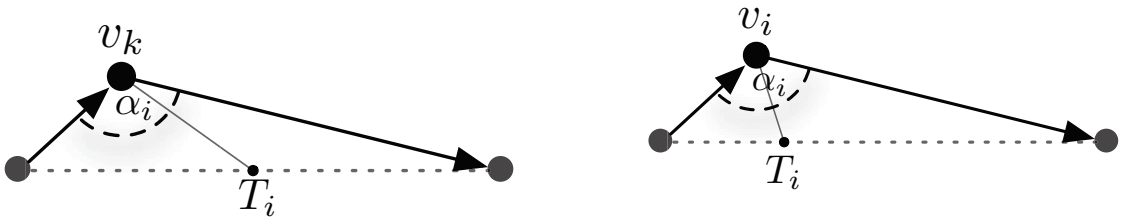


Figure 2: One possible start configuration for a robot chain between two sensors



(a) The Go-To-The-Middle strategy

(b) The Move-On-Bisector strategy

Figure 3: Two different motion strategies

A similar strategy, Move-On-Bisector [DKKM10], achieves the same goal. In this case, the robots do not need to measure the positions of their neighbors. Here, they are only required to determine the direction in which their neighbors are positioned. In exchange, they need to measure these directions continuously and not only at a few points in time. If this is possible, the robots can continuously move in direction of the bisector of the angle which is formed by its two neighbors (see Figure 3(b) for an illustration). Then, it takes time $O(n)$ in the worst case until the robots reach the line between the sensors. This strategy does not guarantee that the robots distribute evenly on this line, but they do not only converge to the line, they actually reach it.

6. Further Improvements

In order to improve upon precision of movement with the Roomba and to increase flexibility for future enhancement (e.g. other robot types), development of a new Wiselib robot backend has been started. By not querying the Roombas odometer but its wheel rotation counters directly introduces more precise motion than the current interface offers. Modularity has been also increased in a way that pose estimation can be managed independently from motion control which enables the usage of external odometers or positioning systems for even higher precision.

Since such precision problems are not at all restricted to this specific type of robot, there has been research on tolerating [KL10], [NY00] and measuring [BF95] odometry errors as well as on reducing errors e.g. by means of collaboration [RDM00][2]. In FRONTS we also researched how robot motion can be lead by radio waves of discrete intensities (as measurable with the used hardware) [HKK11], [Has10] or how a swarm of robots can build up a triangulation infrastructure to help in localization and thus, precise movement [FKK+11], [FKKS10]. Furthermore, research has been started on decomposition of motion into simple geometric steps [FHKS11]. Often such singular steps (such as moving on a circle) are implementable more precisely than arbitrary motion. Practical issues of this approach in the case of the Roomba are work in progress.

[2]Naturally, this list is by no means complete