CoreSLAM : a SLAM Algorithm in less than 200 lines of C code

Bruno Steux, Oussama El Hamzaoui
Mines ParisTech - Center of Robotics, Paris, FRANCE.

***Abstract*—This paper presents a Laser-SLAM algorithm which has been programmed in less than 200 lines of C-language code. Our idea was to develop and implement a very simple SLAM algorithm that could be easily integrated into our particle-filter based localization subsystem. For our experiments, we have been using a homebrew robotic platform called MinesRover. It's a six wheels robot fully equipped with sensors, including a Hokuyo URG04 laser scanner. A typical example of our experiments is presented, showing the good performance of the algorithm. Furthermore, the full source code of the map update and map matching functions are provided in this paper. This work shows the possibility to perform complex tasks using simple and easily programmable algorithms.**

***Index Terms*—SLAM, laser scanner, mobile robot, localization, mapping, particle filter.**

I. Introduction

THIS article presents a new SLAM algorithm called CoreSLAM. This algorithm was developed as a part of the robotics system called CoreBots developped at Mines ParisTech.

CoreSLAM was named after the tiny size of its code, being smaller than 200 lines of C-language code. It was so small that we decided to publish its source code in a conference paper, so that it would not be just open source, but published source ! What encouraged us to do so is the good performance we've obtained during our experiments.

After a short review of existing SLAM algorithms, we will present the motivations which led us to create the CoreSLAM algorithm. Section III will be devoted to the description of the algorithm. We will then introduce the robotic platform used for our tests, and discuss the results obtained.

II. Previous Work

Many researches attempt to solve the problem of simultaneous localization and mapping, commonly called SLAM. The algorithms developed can be classified according to the types of sensors used or the calculation methods adopted. There are SLAM algorithms based on computer vision by one camera [1] or several cameras [8], and algorithms that use a laser sensor or sonars [4]. Concerning the computation methods, two large families exist. On the one hand, algorithms based on the use of Kalman filters [7], and on the other hand, we find algorithms using particle filters [2], or Rao-Blackwellized particle filters - a mix of particle and Kalman filtering - like FastSLAM[5].

Some research has focused on the comparison of these algorithms in performance and speed of calculation [6], showing that the problem is still not solved in all cases...

Among laser SLAM algorithms based on the use of particle filters, the DP-SLAM is one of the best known [2]. DP-SLAM works by maintaining a joint distribution over robot poses and maps via a particle filter. The algorithm associates a map to each particle, and focuses on the problem of sharing parts of maps among particles in order to minimize memory (and time through map copy). The problem with DP-SLAM is that it is rather complex to integrate into an existing particle filter based localization susbystem, like ours - which is very similar to the one described in [3].

During our review, we've seen that most algorithms are tested on slow robots compared to ours - their speed hardly exceeding 1 m/s -, and/or are using very expensive lasers - generally SICK or IBEO laser scanner with ranges of 100 meters. The speed of our robot - which reaches 3 m/s -, combined with the short range of the Hokuyo URG04, creates major but interesting problems for the tasks of localization and mapping.

In addition, most if not all SLAM algorithms developed so far use methods requiring many lines of code or complex mathematics, and so requiring much effort to understand and validate their operation - generally relying on many different parameters that influence the results. We began our work with an important goal: making a simple algorithm, easy to understand and still offering good performance - and most of all easy to integrate into an existing particle filtering localization framework. We added one last requirement : our algorithm had to be embedded in the end, so it had to minimize memory consumption and use integer computation in critical loop code.

III. Methodology

In contrary to DP-SLAM, we decided to use only one map. The advantage of DP-SLAM over CoreSLAM is thus the thoretical ability not to be lost in long corridors, and this is the goal indeed of the map-per-particle concept - not the loop closing which can't be achieved in DP-SLAM without an external process. As a matter of fact, we decided that this advantage didn't worth the complexity - especially as we could rely on a good odometry on our platform and given that our goal was to close rather small loops (exploring laboratories instead of corridors...).

As the idea of CoreSLAM was to integrate laser information in our localization subsystem based on particle filter, we

had to write two main functions :

- The scan to map distance function, which acts as the likelihood function used to test each state position hypothesis (particle) in the filter. The source code of the `ts_distance_scan_to_map` function is provided below (Algorithm 2). Note that it simply consists in a simple sum of all the map values at all the impact points of the scan (relative to the position of the particle). This is a very fast procedure, using only integer operations, which enables us not to be limited in the number of particles while running in real-time. However, that way to estimate likelihood implies that the map is constructed specifically...
- The map update function, used to build the map as the robot is going forward, that is discussed below in detail.

Building a map compatible with a particle filter is not straightforward : the peakiness of the likelihood function is very important for the efficiency of the filter and thus should be easily controllable. We achieved this by using grey-level maps, the map update consisting in digging holes the width of which directly corresponds to our likelihood function peakiness. For each obstacle detected, the algorithm does not draw a single point, but a function with a hole whose lower point is at the position of the obstacle (see figure 1 for a zoom on a newly created map). As a result, constructed map are not looking like traditional maps, but are looking like surfaces with attracting holes. The integration into the map is done through an $\alpha\beta$ filter (line 74 of Algorithm 4), resulting in a convergence of the map to the newer profiles. The map update is called at the last updated position of the particle filter (being the weighted average of all the particles after likelihood evaluation), optionally with a *latency* (see discussion below).

The source code of the `ts_map_update` function is provided below (Algorithm 3). It uses the function `ts_map_laser_ray` (Algorithm 4), which is the most tricky part. The latter uses a Bresenham algorithm to draw the laser rays in the map, with another enhanced bresenham algorithm inside to compute the right profile. It uses only integer computation and no integer division in critical parts. Even if it is executed only once per step (at 10 Hz in our case), it is critical that this procedure be fast, since due to the high resolution (1cm per pixel) of our map, a big part of our map frame is touched by a laser scan.

CoreSLAM can easily be integrated into a particle filter, but can also be used without. Indeed, the constructed map, with its attracting holes and slopes, can be used with any gradient descent algorithm. The matching algorithm converges more easily towards the obstacle, as the hole-function is acting as a guide. Our stand-alone version uses a very simple Monte-Carlo algorithm to match the current scan with the map and to retrieve the updated position of the robot. Indeed, the stand-alone version was developed in order to tune the parameters of CoreSLAM, i.e. the integration speed of the scans into the map (parameter quality in *ts_map_update* function, see 2), which was set to 50, the width of the "hole" surrounding the impact points in our map (set to a fixed value

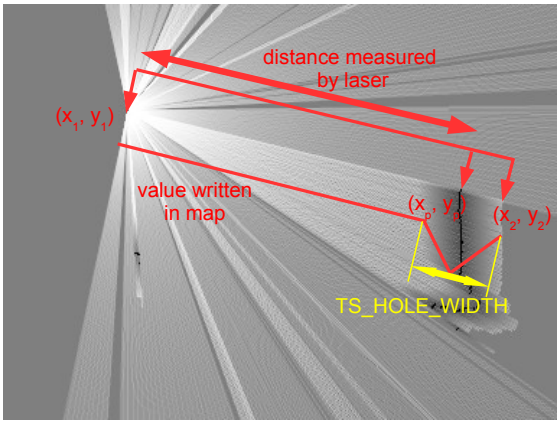

Figure 1. Result of the integration into the map of a range scan by the laser. One can see the "V" shape of the hole drawn around each laser impact.

of 600 mm in our implementation), and the scale of the map (1 point = 1 centimeter in our case). In this version, the odometry can be ignored or be used as a starting point for the Monte-Carlo search (on figures 5 and 6, the odometry is ignored so that odometry and localization by laser can be compared to each other)

The version based on particle filtering is necessary to manage ambiguous situations - and thus is necessary for re-localization (when we start with a full map instead of a blank map). It is also the best for the integration of odometry, since we can cope with non-systematic errors like slippage by integrating into the filter a non-linear error model (by setting the slippage probability to 10%, we make 10% of the particles to stay where they originally lie with high model noise, and 90% of the particles evolve with the odometry with low model noise).

The particle filter is also a very good framework for integrating other sensors than odometry and laser. We have also integrated a GPS (for outdoor use) and a compass (without good success at the moment, the sensor being very sensitive to magnetic noise in our case).

We'll finish our presentation of CoreSLAM by discussing two issues relative to the accuracy of localization and the latency of integration of the data into the map.

Subpixel accuracy : even if our map has a resolution of 1cm, it is possible to measure displacements smaller than 1cm, due to the fact that our `ts_distance_scan_to_map` function takes into account several points to make its calculation. Even a 1mm move can be measured, because some of the laser points will land in another point of the map.

Latency : The concept of latency - the time between the laser scan and its integration into the map - is necessary to measure small displacements relative to the map resolution. For instance, in order to correctly measure the displacement for an object moving at 1cm/s with a map resolution of 1 cm and a frequency of 1Hz, it is necessary to wait for 10 measurements, then the latency should be 10. Indeed the

latency theoretical formula is :

$Latency = \frac{MapResolution * MeasurementFrequency}{RobotSpeed}$

Concerning our robot and given our map resolution, the formula is : $Latency = \frac{0.01*10}{RobotSpeed}$. Our robot operates at a rather high speed (2 m/s). Its speed falls rarely to 0.01 m/s, and even at this speed, a latency equal to 1 is sufficient to do calculations. Moreover, the subpixel accuracy mentioned before makes the task easier. We thus decided to remove the latency management code for CoreSLAM, but recall that it's necessary for applying CoreSLAM to slow moving objects.

We couldn't end the description of our algorithm without discussing loop closure. Our algorithm doesn't cover loop closure, but can be integrated with any loop closing algorithm. Indeed, we are currently developping a tinyLoopCloser algorithm that should complement our CoreSLAM nicely.

IV. Platform description

The platform used for our experiments is the MinesRover (see figure 2), a homebrew robot developped jointly by Mines ParisTech and SAGEM DS. This rover has four driving and steering wheels and two odometry wheels (see figure 3). The mechanical architecture of the robot, based on a rocker-bogie articulation, yields a very good odometry information since the two central free-running wheels always - at least theoretically - keep contact with the ground. This mechanical setup offers a protection against the problem of slippage, without shock absorbers.

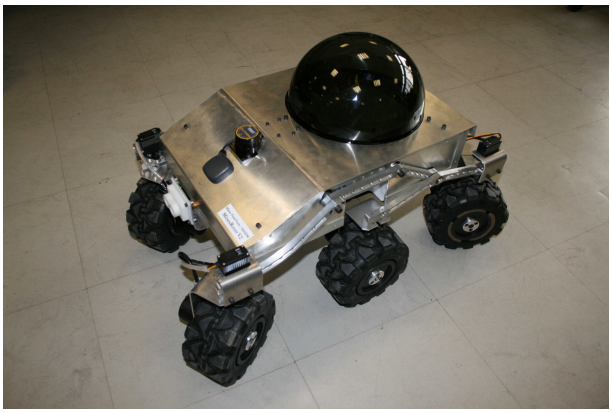

Figure 2. The Mines Rover platform. We can see on the picture the dome for the camera, the steering servomotors, the HOKUYO URG-04 laser range scanner, the GPS receiver (grey square) and the emergency ultrasonic sensor in the front of the robot.

The robot is powered by a 4-elements LiPo battery (14.8V) of 4.1 Ah. The four 45W CC-motors enable a top speed of 3 m/s.

The GPS receiver allows a location accuracy that can reach 1 meter (using one EGNOS satellite). The Hokuyo URG04 laser sensor offers a 10Hz horizontal scanning, with a range of 5.6 meters. In order to retrieve the robot's direction, we combine the results of a compass and GPS. The ultrasonic sensor provides an opportunity to discover any obstacle not detected by the laser (like stair steps), and is mostly used as an emergency stop. The robot also embeds a 5-axis IMU that provides yaw rate information anf the inclination of the robot (not used in our experiments yet).

The electronics of the robot is centered on a Qwerk module, designed by charmedlabs.com. It is a board based on an ARM 9 microprocessor coupled with a Xilinx FPGA logic.

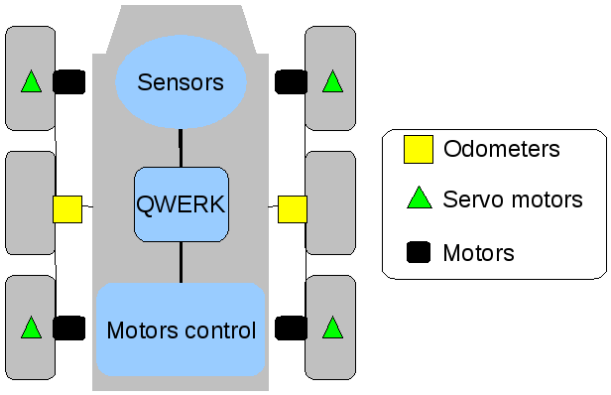

Figure 3. The Mines Rover mechanical architecture. It's a 6 wheel robot, with 4 steering and driving wheels, and 2 free-rotating wheels equipped with 2000 points encoders. The Qwerk module is at the center of the robot. Its 200Mhz microprocessor is able to manage all the sensors and actuators reliably.

A Linux operating system is intalled on the Qwerk and provides USB bus support for the GPS and Hokuyo. The FPGA logic takes care of the I²C bus support, odometer inputs and servo-motor control.

The laser sensor used is a Hokuyo URG-04LX, connected through USB. While being a great sensor for the price, it suffers from several shortcomings :

- Its maximum range is limited to 5.6 meters, which considering the speed of our robot - 3 meters/s - is severly limitating. In our cluttered environnement, a lot of measures by the sensors gave 0 meters (indicating a bad or no reflection) or ambiguous measures (like lower parts of moving chairs, or boxes on the ground next to walls - where both the front of the box or the wall behind can be detected).
- The frequency of measure (10Hz) is also limited. Running at 3m/s for instance into a wall, the wall appears inclined, since the last point of the measure after 240° of turning of the laser, is 240/360*3/10 = 20 centimeters nearer than the first point of measure (though instantly being at the same distance). In our robot, it was necessary to take this into consideration, so we corrected every scan using a constant (for each scan) longitudinal and rotationnal speed.

V. Results

The experiments were conducted in the Electronic Laboratory of Mines ParisTech. This environment was really challenging for the algorithm : it's a highly cluttered environment with boxes and computers lying on the ground, many tables and shelves (see figure 9) that are very difficult to detect and trace using a the laser scanner.

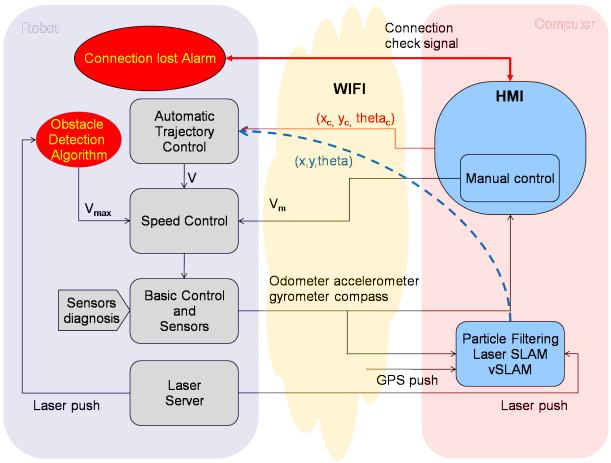

Figure 4. Software architecture diagram of the Mines Rover robotic platform. On the left, the embedded software. On the right side, the part of software running on a desktop PC. Communication between the Mines Rover and the operator is done using an HMI on the desktop PC. The particle filtering algorithm is running on a desktop computer in order to reduce the burden of the ARM9 on the robot. We use a wireless connection, a MIMO router in the robot allowing a good quality of connection.

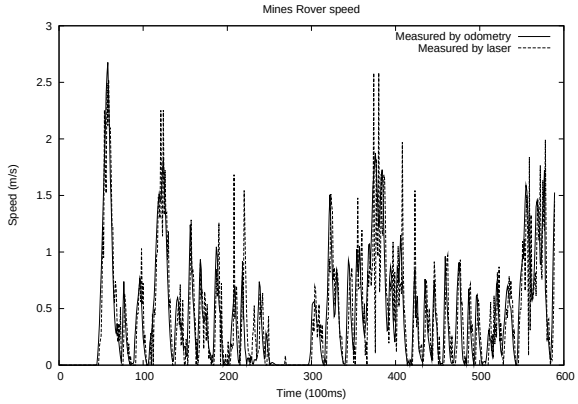

Figure 5. The speed of the robot estimated using odometry alone and our laser CoreSLAM on the other side (without taking into account odometry), during experiment show on 7. Note the good match between measures. We can observe a small latency of the laser measure (of around 1 frame), which is foreseeable considering the nature of the sensor. This demonstrates that our odometry is very good (and perfectly calibrated) and that the laser CoreSLAM is able to cope with the high velocity of the robot (reaching 2.5 m/s in this experiment).

Figures 5, 6 and 7 show results with the same sample of data. On the first two figures, we see the comparison between odometry and estimation of movement by laser alone, and show that they are very similar. The map on figure 7 was constructed by combining laser and odometry information, and shows a good accuracy of reconstruction, the loop being almost closed. Interestingly, the slippage observed in this experiment (see figure 8) was corrected by the laser information in a nice way.

Note that the speed of the robot in our experiments reached 2.5 meters/s and the yaw rate reached 150°/s (measured by odometry and confirmed by the laser measurements), the very high angular speed of the robot being favored by its 4 steering wheels.

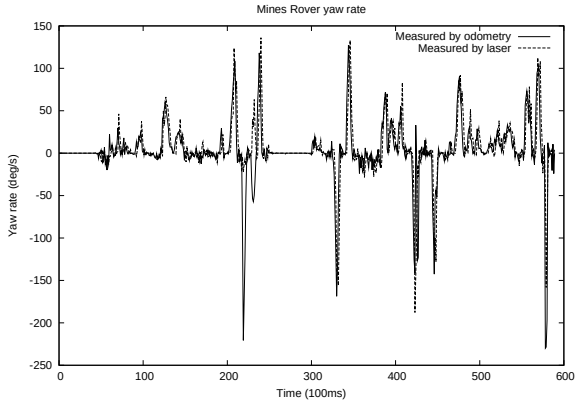

Figure 6. The yaw rate measured by odometry alone and our laser CoreSLAM (without taking into account the available odometry information). Note the yaw rate reaching 250°/s in this experiment - demonstration the very high steering ability of the Mines Rover. Note that in this experiment, correspondong to the map shown on 7, we observed a slippage of odometry (negative peaks on frames 220 and 230) when the driver of the robot hit the right wall (see map 8). The CoreSLAM based on laser alone ignored this.

VI. Conclusions

We have developed a simple and efficient algorithm which is able to perform SLAM using data from a laser sensor. It was designed to be integrated easily and efficiently into any particle filtering-based localization system. It was tested indoor in a very cluttered environment, with a fast robot, using a very affordable laser sensor. The CoreSLAM algorithm shows good performance in this situation, being almost able to close a loop while being open-loop only. As the source code was very small and optimized, we decided to publish it in this paper.

During the next stages of the algorithm development, we will try to use our platform in an outdoor environment by fusing with GPS and compass. We are also working on providing a very short and simple loop closing algorithm (tinyLoopCloser), that automatically detects loop closure and corrects the trajectory of the robot accordingly.

References

[1] Andrew J. Davison. Real-time simultaneous localisation and mapping with a single camera. *Ninth IEEE International Conference on Computer Vision (ICCV'03) - Volume 2*, 2003.

[2] Austin Eliazar and Ronald Parr. Dp-slam: Fast, robust simultaneous localization and mapping without predetermined landmarks. *Proceedings of the International Joint Conference on Artificial Intelligence*, 2003.

[3] Dieter Fox, Sebastien Thrun, Wolfram Burgard, and Frank Dellaert. Particle filters for mobile robot localization.

[4] Feng Lu and Evangelos Milios. Robot pose estimation in unknown environments bu matching 2d range scans. *Journal of Intelligent and Robotic Systems*, 1997.

[5] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam: A factored solution to the simultaneous localization and

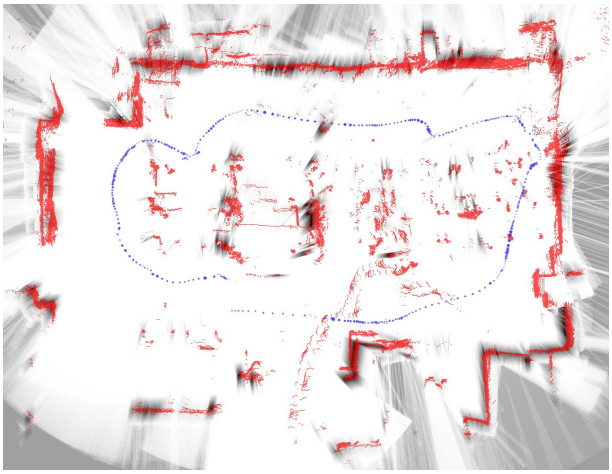

Figure 7. A map of the laboratory obtained during our experiments. In gray, the "holes" map constructed by the robot. In red, we have overlaid all the scans taken by the laser. In blue, the reconstructed trajectory of the robot. The loop closure is almost perfect here. This map was obtained by combining information from odometers and laser (monte-carlo search).

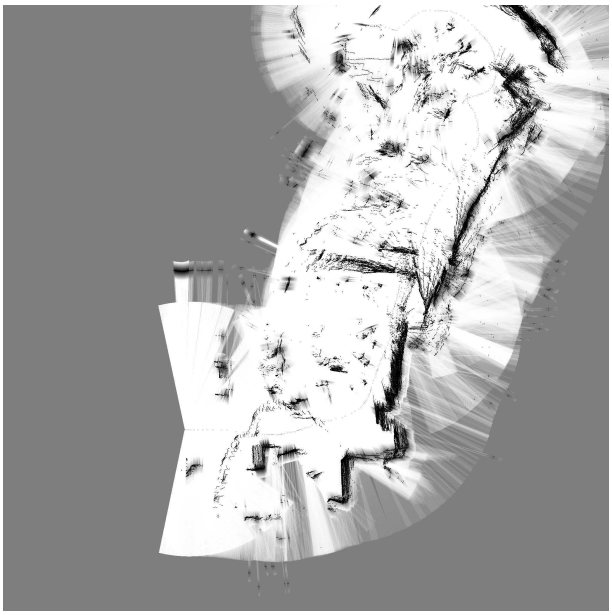

Figure 8. The map constructed using odometry only. Note the slippage when the driver hit the right wall of the lab.

mapping problem. *In Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 593–598, 2002.

[6] Robert Ouellette and Kotaro Hirasawa. A comparison of slam implementations for indoor mobile robots. *Proceedings of the 2007 IEEURSJ, Int. Conference on Intelligent Robots and Systems*, 2007.

[7] Soren Riisgaard and Morten Rufus Blas. *SLAM for Dummies. A Tutorial Approach to Simultaneous Localization and Mapping*.

[8] Joan Sola, Andr e Monin, Michel Devy, and Teresa Vidal-Calleja. Fusing monocular information in multicamera slam. *IEEE TRANSACTIONS ON ROBOTICS, VOL. 24, NO. 5*, 2008.

Bruno Steux

Bruno.Steux@mines-paristech.fr

Assistant Professor at Mines ParisTech - Center of Robotics

Oussama El Hamzaoui

Figure 9. A picture of the lab where the experiments took place. This corresponds to the upper left part of the map shown on 7. Note that the laser sees both the left wall and the shelves, the two open doors and the table legs.

Oussama.El_Hamzaoui@mines-paristech.fr

PhD Student at Mines ParisTech - Center of Robotics

Algorithm 1 CoreSLAM : header and data structures

```
#ifndef _CoreSLAM_H_
#define _CoreSLAM_H_

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define TS_SCAN_SIZE 8192
#define TS_MAP_SIZE 2048
#define TS_MAP_SCALE 0.1
#define TS_DISTANCE_NO_DETECTION 4000
#define TS_NO_OBSTACLE 65500
#define TS_OBSTACLE 0
#define TS_HOLE_WIDTH 600

typedef unsigned short ts_map_pixel_t;

typedef struct {
    ts_map_pixel_t map[TS_MAP_SIZE * TS_MAP_SIZE];
} ts_map_t;

typedef struct {
    double x[TS_SCAN_SIZE], y[TS_SCAN_SIZE];
    int value[TS_SCAN_SIZE];
    int nb_points;
} ts_scan_t;

typedef struct {
    double x, y;     // in mm
    double theta;    // in degrees
} ts_position_t;

void ts_map_init(ts_map_t *map);
int ts_distance_scan_to_map(ts_scan_t *scan, ts_map_t *map, ts_position_t *pos);
void ts_map_update(ts_scan_t *scan, ts_map_t *map, ts_position_t *position,
                   int quality);

#endif // _CoreSLAM_H_
```

Algorithm 2 CoreSLAM : the scan to map distance measure

```
int ts_distance_scan_to_map(ts_scan_t *scan, ts_map_t *map, ts_position_t *pos)
{
    double c, s;
    int i, x, y, nb_points = 0;
    int64_t sum;

    c = cos(pos->theta * M_PI / 180);
    s = sin(pos->theta * M_PI / 180);
    // Translate and rotate scan to robot position
    // and compute the distance
    for (i = 0, sum = 0; i != scan->nb_points; i++) {
        if (scan->value[i] != TS_NO_OBSTACLE) {
            x = (int)floor((pos->x + c * scan->x[i]
                - s * scan->y[i]) * TS_MAP_SCALE + 0.5);
            y = (int)floor((pos->y + s * scan->x[i]
                + c * scan->y[i]) * TS_MAP_SCALE + 0.5);
            // Check boundaries
            if (x >= 0 && x < TS_MAP_SIZE && y >= 0 && y < TS_MAP_SIZE) {
                sum += map->map[y * TS_MAP_SIZE + x];
                nb_points++;
            }
        }
    }
    if (nb_points) sum = sum * 1024 / nb_points;
    else sum = 2000000000;
    return (int)sum;
}

void ts_map_init(ts_map_t *map)
{
    int x, y, initval;
    ts_map_pixel_t *ptr;
    initval = (TS_OBSTACLE + TS_NO_OBSTACLE) / 2;
    for (ptr = map->map, y = 0; y < TS_MAP_SIZE; y++) {
            for (x = 0; x < TS_MAP_SIZE; x++, ptr++) {
                *ptr = initval;
            }
    }
}
```

Algorithm 3 CoreSLAM : full scan map update

```
void ts_map_update(ts_scan_t *scan, ts_map_t *map, ts_position_t *pos, int quality)
{
    double c, s, q;
    double x2p, y2p;
    int i, x1, y1, x2, y2, xp, yp, value;
    double add, dist;

    c = cos(pos->theta * M_PI / 180);
    s = sin(pos->theta * M_PI / 180);
    x1 = (int)floor(pos->x * TS_MAP_SCALE + 0.5);
    y1 = (int)floor(pos->y * TS_MAP_SCALE + 0.5);
    // Translate and rotate scan to robot position
    for (i = 0; i != scan->nb_points; i++) {
        x2p = c * scan->x[i] - s * scan->y[i];
        y2p = s * scan->x[i] + c * scan->y[i];
        xp = (int)floor((pos->x + x2p) * TS_MAP_SCALE + 0.5);
        yp = (int)floor((pos->y + y2p) * TS_MAP_SCALE + 0.5);
        dist = sqrt(x2p * x2p + y2p * y2p);
        add = TS_HOLE_WIDTH / 2 / dist;
        x2p *= TS_MAP_SCALE * (1 + add);
        y2p *= TS_MAP_SCALE * (1 + add);
        x2 = (int)floor(pos->x * TS_MAP_SCALE + x2p + 0.5);
        y2 = (int)floor(pos->y * TS_MAP_SCALE + y2p + 0.5);
        if (scan->value[i] == TS_NO_OBSTACLE) {
            q = quality / 2;
            value = TS_NO_OBSTACLE;
        } else {
            q = quality;
            value = TS_OBSTACLE;
        }
        ts_map_laser_ray(map, x1, y1, x2, y2, xp, yp, value, q);
    }
}
```

Algorithm 4 CoreSLAM : laser ray map update

```
#define SWAP(x, y) (x ^= y ^= x ^= y)

void ts_map_laser_ray(ts_map_t *map, int x1, int y1, int x2, int y2,
                      int xp, int yp, int value, int alpha)
{
    int x2c, y2c, dx, dy, dxc, dyc, error, errorv, derrorv, x;
    int incv, sincv, incerrorv, incptrx, incptry, pixval, horiz, diago;
    ts_map_pixel_t *ptr;

    if (x1 < 0 || x1 >= TS_MAP_SIZE || y1 < 0 || y1 >= TS_MAP_SIZE)
        return; // Robot is out of map

    x2c = x2; y2c = y2;
    // Clipping
    if (x2c < 0) {
        if (x2c == x1) return;
        y2c += (y2c - y1) * (-x2c) / (x2c - x1);
        x2c = 0;
    }
    if (x2c >= TS_MAP_SIZE) {
        if (x1 == x2c) return;
        y2c += (y2c - y1) * (TS_MAP_SIZE - 1 - x2c) / (x2c - x1);
        x2c = TS_MAP_SIZE - 1;
    }
    if (y2c < 0) {
        if (y1 == y2c) return;
        x2c += (x1 - x2c) * (-y2c) / (y1 - y2c);
        y2c = 0;
    }
    if (y2c >= TS_MAP_SIZE) {
        if (y1 == y2c) return;
        x2c += (x1 - x2c) * (TS_MAP_SIZE - 1 - y2c) / (y1 - y2c);
        y2c = TS_MAP_SIZE - 1;
    }

    dx = abs(x2 - x1); dy = abs(y2 - y1);
    dxc = abs(x2c - x1); dyc = abs(y2c - y1);
    incptrx = (x2 > x1) ? 1 : -1;
    incptry = (y2 > y1) ? TS_MAP_SIZE : -TS_MAP_SIZE;
    sincv = (value > TS_NO_OBSTACLE) ? 1 : -1;
    if (dx > dy) {
        derrorv = abs(xp - x2);
    } else {
        SWAP(dx, dy); SWAP(dxc, dyc); SWAP(incptrx, incptry);
        derrorv = abs(yp - y2);
    }
    error = 2 * dyc - dxc;
    horiz = 2 * dyc;
    diago = 2 * (dyc - dxc);
    errorv = derrorv / 2;
    incv = (value - TS_NO_OBSTACLE) / derrorv;
    incerrorv = value - TS_NO_OBSTACLE - derrorv * incv;
    ptr = map->map + y1 * TS_MAP_SIZE + x1;
    pixval = TS_NO_OBSTACLE;
    for (x = 0; x <= dxc; x++, ptr += incptrx) {
        if (x > dx - 2 * derrorv) {
            if (x <= dx - derrorv) {
                pixval += incv;
                errorv += incerrorv;
                if (errorv > derrorv) {
                    pixval += sincv;
                    errorv -= derrorv;
                }
            } else {
                pixval -= incv;
                errorv -= incerrorv;
                if (errorv < 0) {
                    pixval -= sincv;
                    errorv += derrorv;
                }
            }
        }
        // Integration into the map
        *ptr = ((256 - alpha) * (*ptr) + alpha * pixval) >> 8;
        if (error > 0) {
            ptr += incptry;
            error += diago;
        } else error += horiz;
    }
}
```