A SLAM toolbox in Matlab

Joan Solà
LAAS-CNRS

March 20, 2009

1 The SLAM toolbox presentation

In a typical SLAM problem, one or more robots navigate an environment, discovering and mapping landmarks on the way by means of their onboard sensors. Observe in Fig. 1 the existence of robots of different kinds, carring a different number of sensors of different kinds, and observing landmarks of different kinds. All this variety of data is handled by the present toolbox in a way that is quite transparent.

In this toolbox, we organized the data into three main groups, see Table 1. The first group contains the objects of the SLAM problem itself, as they appear in Fig. 1. A second group contains objects for simulation. A third group is designated for graphics output.

Apart from the data, we have of course the functions. We organized them in three levels, from most abstract and generic to the basic manipulations, as is sketched in Fig. 2. The highest level deals exclusively with the structured

Table 1: All data structures.

Purpose	SLAM	Simulator	Graphics
Robots	`Rob`	`SimRob`	
Sensors	`Sen`	`SimSen`	`SenFig`
Landmarks	`Lmk`	`SimLmk`	
Observations	`Obs`	`SimObs`	
Map	`Map`		`MapFig`
Motion control	`Con`	`SimCon`	
Non observables	`Nob`		
Time	`Tim`		

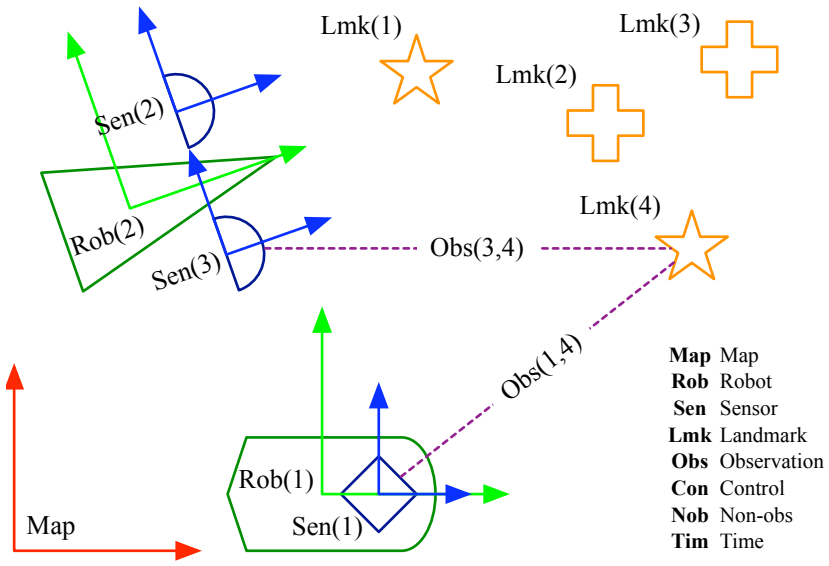

Figure 1: Overview of the SLAM problem with the principal data structures.

data we mentioned just above, and calls functions of an intermediate level, the interface level. The interface level functions split these structures into more mathematically meaningful elements, check data types to decide on the applicable methods, and call the basic functions that constitute the basic level.

1.1 Data organization

It follows a brief explanation of the SLAM data structures, the Simulation and Graphic structures, and the plain data types.

1.1.1 SLAM data

For a SLAM system to be complete, we need to consider the following parts:

`Rob:` A set of robots.

`Sen:` A set of sensors.

`Lmk:` A set of landmarks.

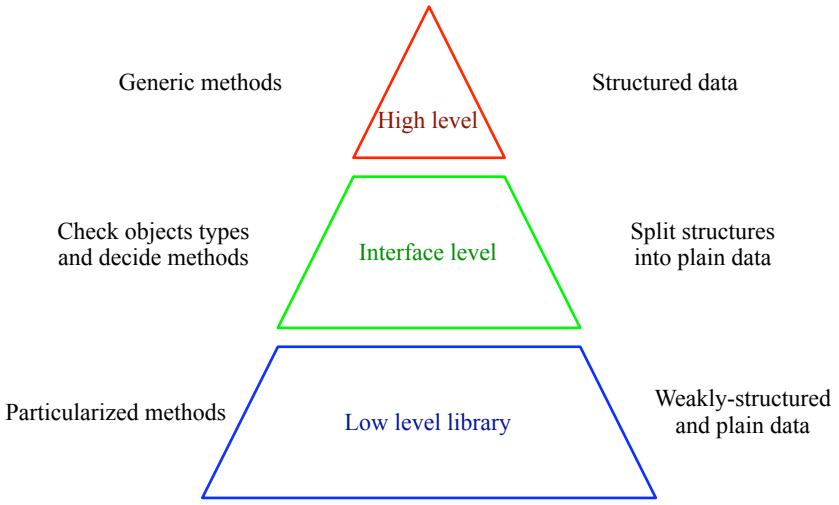

Figure 2: Overview of the levels of abstraction of the functions and its relation to data structuration. Functions and scripts in the High and Interface levels are in the `%SLAMtoolbox/HighLevel/` directory. The Low Level library occupies all other directories.

`Map:` A stochastic map containing the states of robots, landmarks, and eventually sensors.

`Obs:` The set of landmark observations made by the sensors.

`Nob:` For sensors providing partial measurements, some prior on the non-measured parts. This is necessary for landmark initialization.

`Con:` A set of control commands for the robot motions.

`Tim:` A few time-related variables.

Our toolbox will consider these objects as the main existing data. In Matlab, these objects are defined as structures holding a variety of fields.[1] These structures have 3-letter names as in the list just above, {`Map, Rob, Sen, Lmk, Obs, Nob, Con, Tim`}. These structures are reproduced in Figs. 3 to 7 for reference. To hold any number of such objects, we use arrays of these structures. For example, all the data related to robot number 2 is stored in `Rob(2)`. To access the rotation matrix defining the orientation of this robot we simply use

[1] These fields can in turn be structures if the data complexity requires so.

`Rob(2).frame.R`

A special case I want to mention here is `Obs`, because observations relate sensors to landmarks: the data associated to the observation of landmark `lmk` from sensor `sen` is stored in `Obs(sen,lmk)`. Before reading on, please revisit Fig. 1 to see how simple things are.

It follows a reproduction of the arborescences of the principal structures in the SLAM data.

```
Rob(r)          Robot structure, containing:
    .id         * robot id
    .name       * robot name
    .type       * robot type
    .motion     * motion model
    .sensors    * list of installed sensors
    .frame      * frame structure, containing:
        .x      - 7-vector, position and orientation x = [t;q]
        .P      - covariances matrix of x
        .t      - position
        .q      - orientation quaternion
        .R      - rotation matrix, R = q2R(q)
        .Rt     - transposed R
        .it     - inverse position, it = -Rt*t
        .iq     - inverse quaternion, iq = q2qc(q)
        .Pi     - PI matrix, Pi = q2Pi(q)
        .Pc     - conjugate PI matrix, Pc = pi2pc(Pi)
        .r      - range in the SLAM map Map
    .vel        * velocity stucture, containing
        .x      - 6-vector, linear and angular velocities
        .P      - covariances matrix of x
        .r      - range in the SLAM map Map
    .state      * state structure, containing
        .x      - robot's state vector, x = [frame.x;vel.x]
        .P      - covariances matrix of x
        .size   - size of x
        .r      - range in the SLAM map Map
```

Figure 3: The `Rob` structure array with its arborescense.

```
Sen(s)          Sensor structure, containing:
    .id         * sensor id
    .name       * sensor name
    .type       * sensor type
    .robot      * robot it is installed to
    .frame      * frame structure, containing:
        .x      - 7-vector, position and orientation x = [t;q]
        .P      - covariances matrix of x
        .t      - position
        .q      - orientation quaternion
        .R      - rotation matrix, R = q2R(q)
        .Rt     - transposed R
        .it     - inverse position, it = -Rt*t
        .iq     - inverse quaternion, iq = q2qc(q)
        .Pi     - PI matrix, Pi = q2Pi(q)
        .Pc     - conjugate PI matrix, Pc = pi2pc(Pi)
        .r      - range in the SLAM map Map
    .par        * sensor parameters
        .k      - intrinsic params
        .d      - distortion vector
        .c      - correction vector
        .imSize - image size
    .state      * state structure, containing
        .x      - sensor's state vector, x = frame.x or x = []
        .P      - covariances matrix of x
        .size   - size of x
        .r      - range in the SLAM map Map
```

Figure 4: The `Sen` structure array with its arborescense.

```
Lmk(l)          Landmark structure, containing:
    .id         * landmark id
    .type       * sensor type
    .used       * landmark is used in the map
    .state      * state structure, containing
        .x      - landmark's state vector
        .P      - covariances matrix of x
        .size   - size of x
        .r      - range in the SLAM map Map
    .par        * other lmk parameters
```

Figure 5: The `Lmk` structure array with its arborescense.

```
Map             Map structure, containing:
    .used       * vector of flags indicating non-free positions
    .x          * state vector's mean
    .P          * covariances matrix
    .size       * size of the map, in number of states
```

Figure 6: The `Map` structure with its arborescense.

```
Obs(s,l)        Observation structure, containing:
    .sen        * index to sensor in Sen()
    .lmk        * index to landmark in Lmk()
    .sid        * sensor id
    .lid        * landmark id
    .exp        * expectation
        .e      - mean
        .E      - covariance
    .nob        * non-observable part
        .n      - mean
        .N      - covariance
    .meas       * measurement
        .y      - mean
        .R      - covariance
    .inn        * innovation
        .z      - mean
        .Z      - covariance
        .iZ     - inverse covariance
        .MD2    - squared Mahalanobis distance
    .pApp       * predicted appearance
    .cApp       * current appearance
    .sc         * matching quality score
    .vis        * lmk is visible from sensor
    .measured   * lmk has been measured by the feature scanner
    .matched    * lmk has been matched by the feature matcher
    .updated    * lmk has been updated in SLAM
    .H_r        * Jacobian of observation function wrt robot pose
    .H_s        *                                   wrt sensor
    .H_l        *                                   wrt landmark
```

Figure 7: The `Obs` structure array with its arborescense.

1.1.2 Simulation data

This toolbox also includes simulated scenarios. We use for them the following objects, that come with 6-letter names to differentiate from the SLAM data:

`SimRob:` Virtual robots for simulation.

`SimSen:` Virtual sensors for simulation.

`SimLmk:` A virtual world of landmarks for simulation.

`SimObs:` A virtual sensor capture, equivalent to the raw data of a sensor.

The simulation structures `SimXxx` are simplified versions of those existing in the SLAM data. Their arborescence is much smaller, and sometimes they may have absolutely different organization. It is important to understand that none of these structures is necessary if the toolbox is to be used with real data.

It follows a reproduction of the arborescences of the principal simulation data structures.

```
SimRob(r)       Simulated robot structure, containing:
    .id         * robot id
    .name       * robot name
    .type       * robot type
    .motion     * motion model
    .sensors    * list of installed sensors
    .frame      * frame structure, containing:
        .x      - 7-vector, position and orientation x = [t;q]
        .t      - position
        .q      - orientation quaternion
        .R      - rotation matrix, R = q2R(q)
        .Rt     - transposed R
        .it     - inverse position, it = -Rt*t
        .iq     - inverse quaternion, iq = q2qc(q)
        .Pi     - PI matrix, Pi = q2Pi(q)
        .Pc     - conjugate PI matrix, Pc = pi2pc(Pi)
    .vel        * velocity stucture, containing
        .x      - 6-vector, linear and angular velocities
```

Figure 8: The `SimRob` structure array with its arborescense.

```
SimSen(s)       Simulated Sensor structure, containing:
    .id         * sensor id
    .name       * sensor name
    .type       * sensor type
    .robot      * robot it is installed to
    .frame      * frame structure, containing:
        .x      - 7-vector, position and orientation x = [t;q]
        .t      - position
        .q      - orientation quaternion
        .R      - rotation matrix, R = q2R(q)
        .Rt     - transposed R
        .it     - inverse position, it = -Rt*t
        .iq     - inverse quaternion, iq = q2qc(q)
        .Pi     - PI matrix, Pi = q2Pi(q)
        .Pc     - conjugate PI matrix, Pc = pi2pc(Pi)
    .par        * sensor parameters
        .k      - intrinsic params
        .d      - distortion vector
        .c      - correction vector
        .imSize - image size
```

Figure 9: The `SimSen` structure array with its arborescense.

```
SimLmk          Simulated landmarks structure, containing:
    .ids        * N-vector of landmark identifiers
    .points     * 3-by-N array of 3D points
    .lims       * limits of playground in X, Y and Z axes
        .xMin   - minimum X coordinate
        .xMax   - maximum X coordinate
        .yMin   - minimum Y coordinate
        .yMax   - maximum Y coordinate
        .zMin   - minimum Z coordinate
        .zMax   - maximum Z coordinate
    .dims       * dimensions of playground
        .l      - length in X
        .w      - width in Y
        .h      - height in Z
    .center     * central point
       .xMean   - central X
       .yMean   - central Y
       .zMean   - central Z
```

Figure 10: The `SimLmk` structure with its arborescense.

```
SimObs(s)       Simulated observation structure, containing:
    .sen        * index to sensor in Sen()
    .ids        * n-vector of measured ids
    .points     * m-by-n array of measured points
```

Figure 11: The `SimObs` structure array with its arborescense. `m` is the dimension of the measurement space. For vision, we have `m = 2`.

1.1.3 Graphics data

This toolbox also includes graphics output. We use for them the following objects, which come also with 6-letter names:

`MapFig:` A 3D figure showing the world, the robots, the sensors, and the current state of the map.

`SenFig:` One figure for each sensor, visualizing its *measurement space*.

The graphics structures `XxxFig` contain data related to the graphical output. There is one figure for the 3D map, and one figure for each one of the sensors, showing the measurement space (Fig. 12).

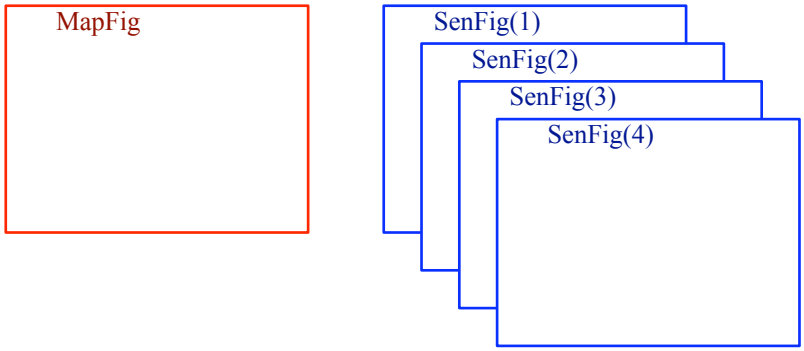

Figure 12: The set of figures. There is a 3D figure `MapFig` containing the current state of the map, and one figure `SenFig(s)` for each sensor `s` to display the information in the measurement spaces. The structures `MapFig` and `SenFig(s)` contain the handles to all graphics objects drawn.

It follows a reproduction of the arborescences of the principal graphics structures.

```
MapFig          Map figure structure, containing:
    .fig        * figure number and handle
    .axes       * axes handle
    .ground     * handle to floor object
    .simRob     * array of handles to simulated robots
    .simSen     * array of handles to simulated sensors
    .simLmk     * handle to simulated landmarks
    .estRob     * array of handles to SLAM robots
    .estSen     * array of handles to SLAM sensors
    .estLmk     * handles to SLAM landmarks, containing:
        .mean   - array of handles to landmarks means
        .ellipse - array of handles to landmarks ellipses
        .label  - array of handles to landmarks labels
```

Figure 13: The `MapFig` structure with its arborescense.

```
SenFig(s)       Sensor figure structure, containing:
    .fig        * figure number and handle
    .axes       * axes handle
    .raw        * handle to raw data
    .measure    * array of handles to landmarks measurements
    .ellipse    * array of handles to landmarks ellipses
    .label      * array of handles to landmarks labels
```

Figure 14: The `SenFig` structure array with its arborescense.

1.1.4 Low-complexity or plain data

The structured data we have seen so far is composed of chunks of lower complexity structures and plain data. This low-complexity data is the data that the low-complexity functions take as inputs.

For low-complexity data we mean:

logicals and scalars: Any Matlab scalar value such as `a = 5` or `b = true`.

vectors and matrices: Any Matlab array such as `v = [1;2]`, `w = [1 2]` `c = [true false]` or `M = [1 2;3 4]`.

character strings: Any Matlab alphanumeric string such as `type = 'pinHole'` or `dir = '%HOME/temp/'`.

frames: Frames are Matlab structures that we created to store data belonging to 3D frames (see Fig. 15 for an instance of the `frame` structure). We do this to avoid having to compute multiple times rotation matrices and other frame-related constructions. Low-level functions involving frame transformations admit frame structures as input data, and therefore frames are here considered also plain data. A frame is specified with a translation vector and an orientation quaternion (type `help quaternion` at the Matlab prompt), giving a state vector of 7 components. This is the essential frame information, stored in field `frame.x`, and the only one that is included as part of the SLAM state vector if the frame is to be estimated. From this 7-vector, all other fields of `frame` are created or updated using the `updateFrame` function.[2]

1.2 Functions

The SLAM toolbox is composed of functions of different importance, defining three levels of abstraction. They are stored in subdirectories according to their field of utility. There is a particular directory, `HighLevel`, with two scripts and a limited set of high-level and interface-level functions. All other directories contain low-level functions.

[2] Do not worry too much about fields `.Pi` and `.Pc` by now. They are here to help computing some Jacobians of functions involving frame transformations. Consult the help contents of `toFrame.m` or [1] in case you are interested in finding out more.

```
frame       Frame structure, containing:
    .x      * the state 7-vector
    .t      * translation vector,     t  = x(1:3)
    .q      * orientation quaternion, q  = x(4:7)
    .R      * rotation matrix,        R  = q2R(q)
    .Rt     * transposed R,           Rt = R'
    .it     * inverse position,       it = -Rt*t
    .iq     * inverse or conjugate quaternion, iq = q2qc(q)
    .Pi     * PI matrix,              Pi = q2Pi(q)
    .Pc     * conjugate PI matrix,    Pc = q2Pi(iq)
```

Figure 15: The `frame` structure and its fields.

1.2.1 High level

The high level scripts are located in the directory `%SLAMtoolbox/HighLevel/`.

There are two main scripts that constitute the highest level:

1. `universalSLAM.m`, the main script. It initializes all data structures and figures, and performs the temporal loop by first simulating motions and measurements, second estimating the map and localization (the SLAM algorithm itself), and third visualizing all the data.

2. `userData.m`, a script containing the data the user is allowed to enter to configure the simulation. It is called by `universalSLAM.m` at the very first lines of code.

Other high-level scripts exist to help initializing all the structured data. They are called by `universalSLAM` just after `userData`:

```
createSLAMstructures.m
createSimStructures.m
createGraphicsStructures.m
```

Finally, some high-level functions exist for creating all graphics figures. They are called by `createGraphicsStructures.m`:

```
createMapFig.m
createSenFig.m
```

1.2.2 Interface level

The interface level functions are also located in the directory `%SLAMtoolbox/HighLevel/`.

The interface level functions interface the high-level scripts and structured data with the low-level functions and the low-complexity data. These functions serve three purposes:

1. Check the type of structured data and select the appropriate methods to manipulate them.
2. Split the structured data into smaller parts of low-complexity data.
3. Call the low-level functions with the low-complexity data (see Section 1.1.4), and assign the outputs to the appropriate fields of structured data.

A good example of interface function is `motion.m`, whose code is reproduced in Fig. 16.

1.2.3 Low level library

There are different directories storing a lot of low-level functions. Although this directory arborescence is meant to be complete, you are free to add new functions and directories (do not forget to add these new directories to the Matlab path). The only reason for these directories to exist is to have the functions organized depending on their utility.

The toolbox is delivered with the following directories:

```
DataManagement/         Certain data manipulations
DetectionMatching/      Features detection and matching
EKF/                    Extended Kalman Filter
Graphics/               Graphics creation and redrawing
Kinematics/             Motion models
Observations/           Observation models
Simulation/             Methods exclusive to simulation
FrameTransforms/        Frame transformations
Math/                   Some math functions
Slam/                   Low-level functions for EKF-SLAM
```

The functions contained in this directories take plain data as input, and deliver plain data as output.

```
function Rob = motion(Rob, Con, dt)

% MOTION  Robt motion.
%   Rob = MOTION(Rob, Con, DT) performs one motion step
%   to robot Rob with control signals Con, following the
%   motion model in Rob.motion. The time increment DT is
%   used only if the motion model requires it.
%
%   See also CONSTVEL, ODO3, UPDATEFRAME.


switch Rob.motion               % check robot's motion model

    case  {'constVel'}
        Rob.state.x = constVel(Rob.state.x,Con.u,dt);

        Rob.frame.x = Rob.state.x(1:7);
        Rob.vel.x   = Rob.state.x(8:13);
        Rob.frame   = updateFrame(Rob.frame);

    case  {'odometry'}
        Rob.frame   = odo3(Rob.frame,Con.u);
        Rob.frame   = updateFrame(Rob.frame);

    otherwise
        error('Unknown motion model');
end
```

Figure 16: The interface function `motion.m`.

References

[1] Joan Solà. *Towards Visual Localization, Mapping and Moving Objects Tracking by a Mobile Robot: a Geometric and Probabilistic Approach.* PhD thesis, Institut National Polytechnique de Toulouse, 2007.