An EKF-SLAM toolbox in Matlab

Joan Solà
LAAS-CNRS

March 27, 2009

Contents

1 Quick start

Hi there! To start the toolbox, do the following:

1. Create a directory for the toolbox (we'll call it `%SLAMtoolbox/`).
2. Move the .ZIP file here. Unzip it.
3. Open Matlab. Add all directories and subdirectories in `%SLAMtoolbox/` to the Matlab path.
4. Execute `universalSlam` from the Matlab prompt, or
5. Edit `userData.m`. Read the help lines. Explore options and create new robots and sensors.
6. Edit and run `universalSlam.m`.

2 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, Fig. 2.

Apart from the data, we have of course the functions. Functions are organized in three levels, from most abstract and generic to the basic manipulations, as is sketched in Fig. 3. The highest level, called *High Level*, deals exclusively with the structured data we mentioned just above, and calls functions of an intermediate level called the *Interface Level*. The interface level functions split the data structures into more mathematically meaningful elements, check objects types to decide on the applicable methods, and call the basic functions that constitute the basic level, called the *Low Level Library*.

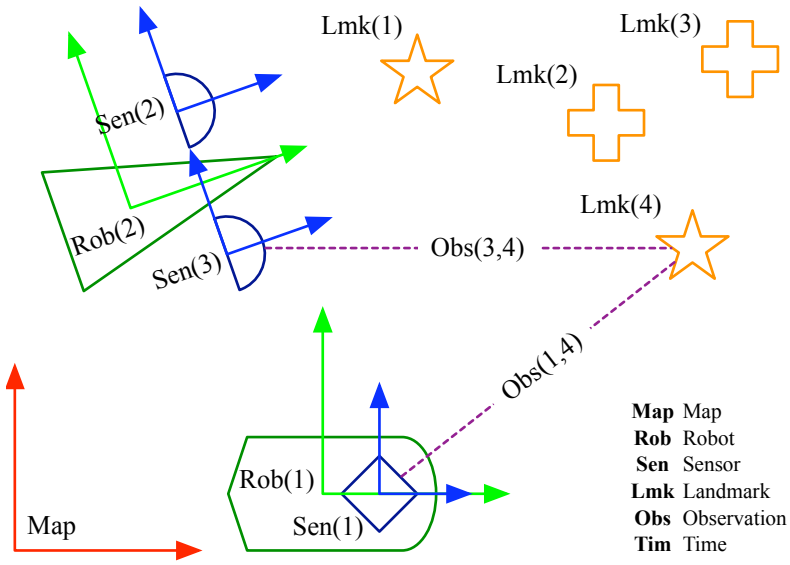

Figure 1: Overview of the SLAM problem with the principal data structures.

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`
Time	`Tim`		

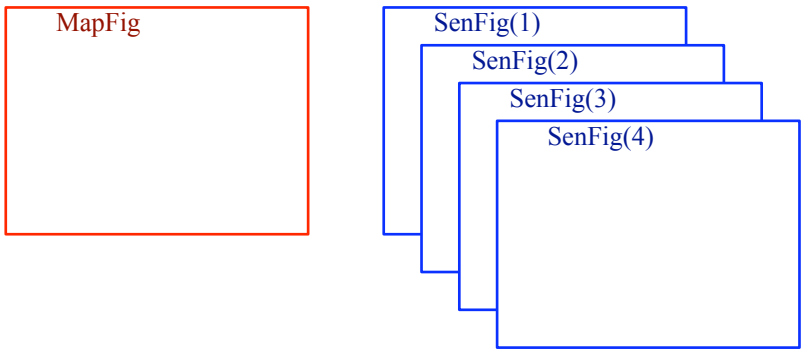

Figure 2: The set of figures. The structures `MapFig` and `SenFig(s)` contain the handles to all graphics objects drawn.

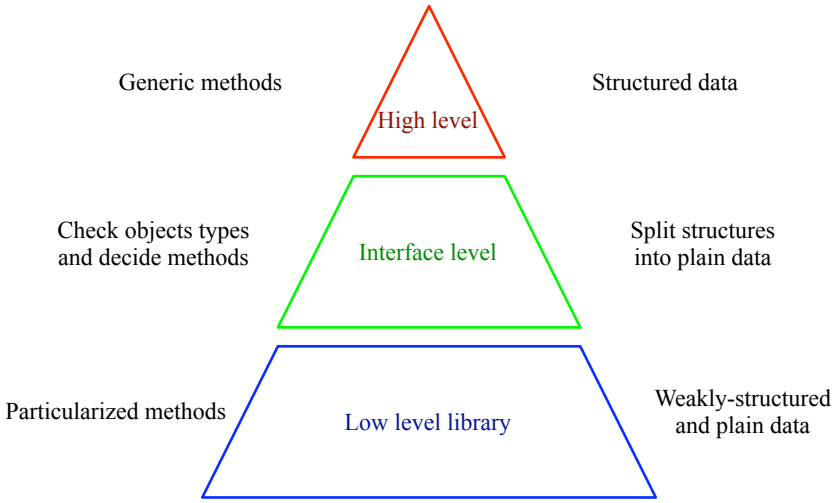

Figure 3: Overview of the levels of abstraction of the functions and their relation to data structuration. Functions and scripts in the High and Interface levels are in the `%SLAMtoolbox/HighLevel/` and `%SLAMtoolbox/InterfaceLevel/` directories. The Low Level library occupies all other directories.

3 Data organization

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

3.1 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.

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

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

`Tim:` A few time-related variables.

This toolbox considers these objects as the only existing data for SLAM. They are defined as structures holding a variety of fields (see Figs. 4 to 9 for reference). Structure arrays hold any number of such objects. 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

```
Rob(2).frame.R
```

A special case I want to mention here is `Obs`, because observations relate sensors to landmarks, therefore requiring two indices: the data associated to the observation of landmark `lmk` from sensor `sen` is stored in `Obs(sen,lmk)`.

It would be wise, before reading on, to revisit Fig. 1 and 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:
    .rob        * index in Rob() array
    .id         * robot id
    .name       * robot name
    .type       * robot type
    .sensors    * list of installed sensors
    .motion     * motion model
    .con        * control structure
        .u      - control signals for the motion model
        .U      - covariance of u
    .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 4: The `Rob` structure array with its arborescense.

```
Sen(s)          Sensor structure, containing:
    .sen        * index in Sen() array
    .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 5: The `Sen` structure array with its arborescense.

```
Lmk(l)          Landmark structure, containing:
    .lmk        * index in Lmk() array
    .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
    .nom        * prior of non-measurable degrees of freedom
        .n      - mean
        .N      - covariance
    .par        * other lmk parameters
```

Figure 6: 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 7: The `Map` structure with its arborescense.

```
Tim             Time structure, containing:
    .firstFrame * first frame to evaluate
    .lastFrame  * last frame to evaluate
    .dt         * Sampling period
```

Figure 8: The `Tim` structure with its arborescense.

```
Obs(s,l)        Observation structure, containing:
    .sen        * index to sensor in Sen() array
    .lmk        * index to landmark in Lmk() array
    .sid        * sensor id
    .lid        * landmark id
    .meas       * measurement
        .y      - mean
        .R      - covariance
    .nom        * non-measurable degrees of freedom
        .n      - mean
        .N      - covariance
    .exp        * expectation
        .e      - mean
        .E      - covariance
    .inn        * innovation
        .z      - mean
        .Z      - covariance
        .iZ     - inverse covariance
        .MD2    - squared Mahalanobis distance
    .app        * appearance
        .pred   - predicted appearance
        .curr   - current appearance
        .sc     - matching quality score
    .vis        * flag: lmk is visible from sensor
    .measured   * flag: lmk has been measured
    .matched    * flag: lmk has been matched
    .updated    * flag: lmk has been updated
    .Jac        * Jacobians of observation function e = h(Rob,Sen,Lmk)
        .E_r    - wrt robot pose
        .E_s    - wrt sensor
        .E_l    - wrt landmark
```

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

3.2 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:
    .rob        * index in SimRob() array
    .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 10: The `SimRob` structure array with its arborescense.

```
SimSen(s)       Simulated Sensor structure, containing:
    .sen        * index in SimSen() array
    .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 11: 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 12: 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 13: The `SimObs` structure array with its arborescense. `m` is the dimension of the measurement space. For vision, we have `m = 2`.

3.3 Graphics data

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

`MapFig:` A structure of handles to graphics objects in the 3D map figure. One Map figure showing the world, the robots, the sensors, and the current state of the estimated SLAM map (Figs. 14 and 15).

`SenFig:` A structure array of handles to graphics objects in the sensor figures. One figure per sensor, visualizing its *measurement space* (Figs. 16 and 17).

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

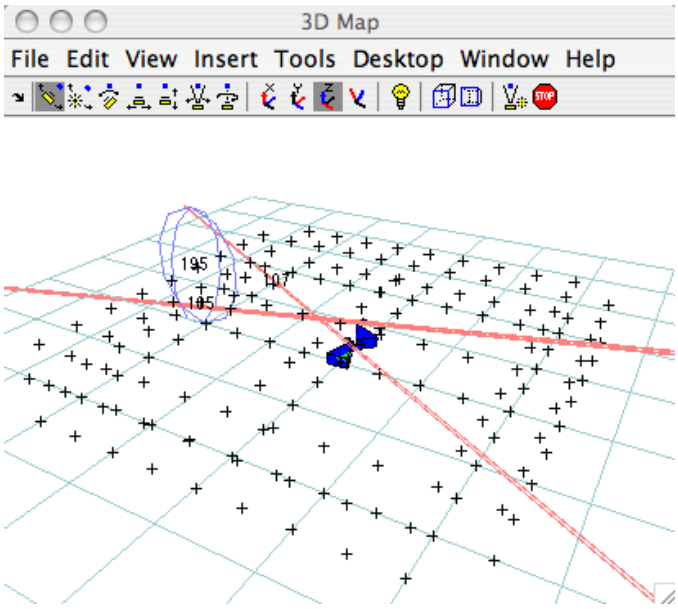

Figure 14: The 3D map figure. `MapFig` contains handles to all objects drawn.

```
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 15: The `MapFig` structure with its arborescense.

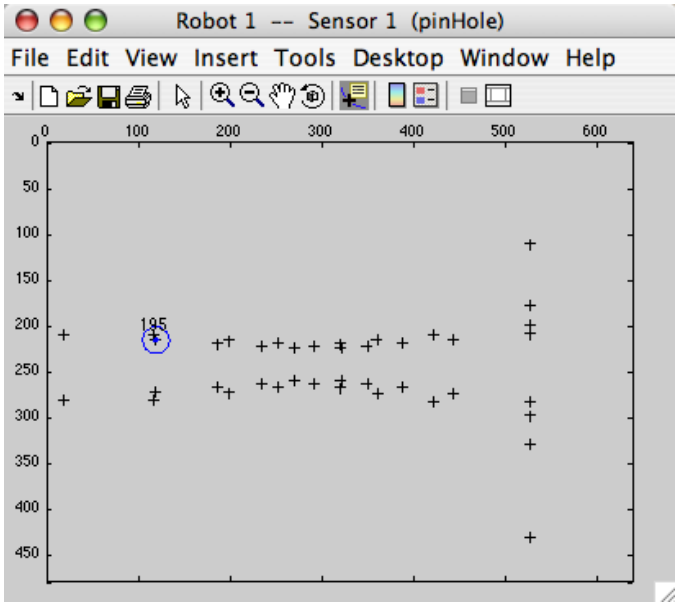

Figure 16: A pin-hole sensor view figure. `SenFig(1)` contains handles to all objects drawn.

```
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 17: The `SenFig` structure array with its arborescense.

3.4 Plain data

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

For plain 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. 18 for an instance of the `frame` structure; type `help frame` at the Matlab prompt). We do this to avoid having to compute multiple times rotation matrices and other frame-related constructions.

A frame is specified by a 7-vector `frame.x` containing translation vector and an orientation quaternion (type `help quaternion` at the Matlab prompt). This is the essential frame information. After each setting or modification of the state `frame.x`, call the `updateFrame()` function to create/update the rest of the frame structure.

```
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 18: The `frame` structure and its fields.

4 Functions

The SLAM toolbox is composed of functions of different importance, defining three levels of abstraction (Fig. 3). They are stored in subdirectories according to their field of utility. There are two particular directories: `HighLevel`, with two scripts and a limited set of high-level functions; and `InterfaceLevel`, with a number of functions interfacing the high level data with the low-level library. All other directories contain low-level functions.

4.1 High level

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

There are two main scripts that constitute the highest level:

`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.

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

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

```
createSLAMstructures()
createSimStructures()
createGraphicsStructures()
```

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

```
createMapFig()
createSenFig()
```

4.2 Interface level

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

The interface level functions interface the high-level scripts and structured data with the low-level functions and the plain 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 plain data.

3. Call the low-level functions with the plain data (see Section 3.4), and assign the outputs to the appropriate fields of structured data.

A good example of interface function is `SimMotion.m`, whose code is reproduced in Fig. 19.

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

% SIMMOTION  Simulated robot motion.
%   Rob = SIMMOTION(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, but it has
%   to be provided because SIMMOTION is a generic method.
%
%   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);

    otherwise
        error('??? Unknown motion model ''%s''.',Rob.motion);
end
```

Figure 19: The interface function `SimMotion.m`. Observe that (1) the interface function checks data types and selects different low-level functions accordingly; (2) the structures are split into chunks of plain data before entering the low-level functions; (3) only `frame.x` is modified by the low-level motion functions in `case 'constVel'`, and `updateFrame()` is called afterwards; (4) the low-level odometry function `odo3()` already performs frame update; (5) there is an error message for unknown motion models.

4.3 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
FrameTransforms/         Frame transformations
    rotations/           Rotations (inside FrameTransforms/)
Graphics/                Graphics creation and redrawing
Kinematics/              Motion models
Math/                    Some math functions
Observations/            Observation models
Simulation/              Methods exclusive to simulation
Slam/                    Low-level functions for EKF-SLAM
```

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

A Programming guidelines

A.1 Matlab Help

Prepare your help headers to look really Matlab-like!

```
% FUN One line description with one space between % and FUN.
%    FUN(X,Y) Longer description, with explanation of function
%    inputs X and Y and the output. There are 4 spaces between
%    % and FUN(). The function name is in CAPITAL LETTERS.
%    Preferably, the input variables X and Y are also in
%    capital letters.
%
%    If the paragraph above is too complex, brake it into
%    different paragraphs.
%
%    If the list of input arguments is too complex, make a
%    list here. Explain ALL input arguments. The list is
%    indented another 4 spaces:
%        X:   one Bourbon
%        Y:   one Scotch
%
%    FUN(X,Y,Z) explain extra inputs Z here and what they do.
%    Explain if they have a default value. If you need to
%    make a new list, remember the 4 spaces!
%        Z:   one beer.
%
%    [out, OUT_x, OUT_y] = FUN(...) returns the Jacobians
%    wrt X and Y. Maybe you have to explain something else.
%    You do not need to repeat the input parameters so you
%    can use the form [out, OUT_x] = FUN(...), with the (...).
%
%    Before saving, select entire paragraphs and do RIGHT
%    CLICK, "Wrap selected comments". This equals all line
%    lengths to approximately the page width.
%
%    See also FUN2, FUN3. Use it exactly like this "See also "
%    + func. names in CAPITAL LETTERS. Matlab parses this line
%    and will create links to the functions' helps ONLY IF YOU
%    FOLLOW THESE GUIDELINE STRICTLY.

%    (c) 2009 You @ LAAS-CNRS. Make yourself famous. See that
%    this comment line is disconnected from the Help body (the
%    previous line has no % sign).
```

A.2 Names of variables and Jacobians.

For convention, we are going to do the following:

1. Variables inside functions have short names in small letters normally.
2. Jacobians are `BIG_small`, where `X_y = dx/dy`.
3. Jacobians are not `Xy`, better `X_y`.
4. Robot, landmark, sensor etc INDICES are always rob, sen, lmk, obs.
5. Robot, landmark, sensor etc IDENTIFIERS are rid, sid, lid, etc.

A.3 Using logical vectors to select desired elements in structure arrays.

1. Use the logical forms. Examples:

```
used  = [Lmk.used];      % a vector of logicals
vis   = [Obs.vis];
drawn = (strcmp((get([MapFig.estLmk.ellipse],'visible')),'on'))';
```

2. Compose the logicals to get new logicals. Example:

```
erase = ~vis & drawn;
```

3. When setting logicals, always use true/false, not 1/0:

```
Obs(1).vis = true;     % Do not use 1 instead of true, otherwise
Obs(2).vis = false;    % you turn the whole vector to numeric.
```

4. You can access an array with the logical vector

```
Lmk(used)   % all the Lmk's that are used
```

5. You can get the indices with FIND()

```
usedIdx = find(used);
```

6. You can also access an array with indices, of course:

```
Lmk(usedIdx)    % this is equivalent to Lmk(used)
```

7. If you want the first N unused Lmk's, do for example

   ```
   Lmk(find(~used,N,'first'))
   ```

   or, easier to read:

   ```
   notUsed = find(~[Lmk.used]);
   Lmk(notUsed(1:N));
   ```

A.4 Function readability I: aligned code reads well!

1. When using consecutive lines of code, try to vertically align all EQUAL signs. Examples:

   ```
   % GOOD: code reads easy
   x        = f(y);
   variable = fun(z);
   JAC_x    = JAC_y*Y_x;

   % BAD: code is a pack
   x = f(y);
   variable = fun(z);
   JAC_x = JAC_y*Y_x;
   ```

2. Similarly, when commenting multiple lines on the right margin, align comments. Examples:

   ```
   % GOOD: comments read well
   x        = f(y);         % these lines
   variable = fun(z);       % are all easy
   JAC_x    = JAC_y*Y_x;    % to read

   % BAD: comments are packed within the code
   x        = f(y); %these lines
   variable = fun(z); % are not easy
   JAC_x    = JAC_y*Y_x; % to read
   ```

3. Exceptions are accepted, but use common sense. Examples

```
% GOOD: all possible alignments coincide
x        = f(y);             % these comments are aligned
variable = g(z);             % with the fourth line.
JAC_x    = JAC_y*Y_x + Z_a*A_variable*VARIABLE_x; % Oops!
output   = JAC_x*P*JAC_x';   % this defines the alignment above.
extra    = I*dont*know;      % over all it is easy to read.

% NOT SO GOOD, BUT OK: alignments come in groups
x        = f(y);      % these comments are NOT aligned
variable = g(z);      % with the fourth and fifth lines.
JAC_x    = JAC_y*Y_x + Z_a*A_variable*VARIABLE_x; % Oops!
output   = JAC_x*P*JAC_x';   % this margin is new
extra    = I*dont*know;      % over all it is easy to read.
```

4. Still, you can try to align different groups of lines. Example

```
x        = f(y);             % these comments aligned,
variable = g(z);             % and the alignment
output   = JAC_x*P*JAC_x';   % continues in next group

y        = 4;                % this follows the same alignment
extra    = 5*eye(3);         % over all it is easy to read.
```

A.5 Functions readability II. Grouping and commenting.

1. Comment every group of lines performing a coherent action before the group. Example:

```
% get idps to delete
used    = [Lmk.used];
idps    = strcmp({Lmk.type},'idpPnt');
drawn   = (strcmp((get([MapFig.estLmk.ellipse],'visible')),'on'))';
delIdps = drawn & idps & ~used;
```

2. Comment individual lines on the right if more info is needed. Example:

```
% get idps to delete
used    = [Lmk.used];                  % used lmks
idps    = strcmp({Lmk.type},'idpPnt'); % lmks that are inverse-depth
drawn   = (strcmp((get([MapFig.estLmk.ellipse],'visible')),'on'))'; % previously drawn
delIdps = drawn & idps & ~used;        % these need to be deleted
```

3. Separate all groups of lines with an empty line so that the code does not look packed. As a rule, no more than 4 lines should go together.

4. Before saving the function, do CNTRL+A, CNTRL+I to make all the indents look nice.

A.6 Make exceptional use of line breaking "..."

Particularly when functions have long names or many long parameters:

```
[out, OUT_x, OUT_y, OUT_z, OUT_par, OUT_calibration] = ...
    functionNameThatMightBeVeryLong(...
        Lmk.state.x,...          % you can put
        Sen(4).par.y,...         % comments here
        Obs(sen,lmk).nom.N,...   % if necessary
        Sen(4).par.k,...         % to explain the
        Sen(4).par.cal);         % input data
```

See `USERDATA.M`, `CREATEMAPFIG.M` to see examples of this.

A.7 Error messages.

Stick to Matlab standards:

```
error('??? Unknown sensor type ''%s''.',Sen(sen).type);
```

gives a 'nice' Matlab error message (the second line is ours!):

```
??? Error using ==> createSensors at 46
??? Unknown sensor type 'pinPole'.

Error in ==> createSLAMstructures at 10
Sen = createSensors(Sensor);

Error in ==> universalSlam at 36
[Rob,Sen,Lmk,Obs,Tim] = createSLAMstructures(...
```

This error information is enough. Matlab has debugging mechanisms to go find further info for the error.