An EKF-SLAM toolbox in Matlab

Joan Solà – LAAS-CNRS

April 24, 2013

Contents

1 Quick start

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

1. Visit www.joansola.eu and download the toolbox package.
2. Move **slamToolbox.zip** where you want the SLAM toolbox to be installed. Unzip it.
3. Rename the expanded directory if wanted (we'll call this directory **SLAMTB/**).
4. Open Matlab. Add all directories and subdirectories in **SLAMTB/** to the Matlab path.[1]
5. Execute **slamtb** from the Matlab prompt.

1.1 Quick curiosity

Or, if you want to get some more insight:

6. Edit **userData.m**. Read the help lines. Explore options and create, by copying and modifying, new robots and sensors. You can modify the robots' initial positions and motions and the sensors' positions and parameters. You can also modify the default set of landmarks or 'World'.
7. Edit and run **slamtb.m**. Explore its code by debugging step-by-step. Explore the Map figure by zooming and rotating with the mouse.
8. Read the help contents of the following 4 functions: **frame**, **fromFrame**, **q2R**, **pinHole**. Follow some of the **See also** links.
9. Set **FigOpt.createVideo** to **'true'** in **userData**. Obtain a series of images to create a video sequence – locate them at **SLAMTB/figures/simu/idpPnt/mono/images/**.
10. Read '**guidelines.pdf**' before contributing your own code.

[1] **File>Set Path>Add with subfolders>[select SLAMTB folder]**

1.2 Quick expert

Or, if you want to explore the full capacity of this toolbox:

11. **SLAM WITH POINTS:** Choose **userDataPnt** instead of the default **userData** (at the third line of code in **slamtb.m**). Scroll down to find the structure **Obs**. Try landmark types **'idpPnt'**, **'ahmPnt'**, **'hmgPnt'**, and **'fhmPnt'**[2] in entry **Opt.init.initType**, and compare performances of inverse-depth against anchored-homogeneous, framed-homogeneous and pure-homogeneous parametrizations for SLAM with 3D points. Read [3, 16].

12. **SLAM WITH LINES:** Choose **userDataLin** instead of the default **userData** (at the third line of code in **slamtb.m**). Scroll down to find the structure **Obs**. Try landmark types **'plkLin'**, **'aplLin'**, **'hmgLin'**, **'ahmLin'** and **'idpLin'** in entry **Opt.init.initType** and compare performances of Plucker, anchored Plucker, Homogeneous, Anchored-homogeneous and Inverse-depth parametrizations for 3D lines. Read [17, 16].

13. **SLAM WITH OMNIDIRECTIONAL CAMERA** Edit **userData.m** or **userDataPnt.m**, go to the **Sensor{1}** section, comment it, and uncomment a second **Sensor{1}** below with a model for Omnicam. Choose **'ahmPnt'** (see 11 above). Thanks to Grigory Abuladze for this contribution!

14. **SLAM WITH MULTIPLE SENSORS:** Choose **userData** as the user data file in **slamtb**. Uncomment the full **Sensor{2}** structure in **userData** and get bi-camera SLAM. Set **Sensor{2}.frameInMap** to **'true'** and get extrinsic self-calibration of the stereo rig. Read [15].

15. **SLAM WITH MULTIPLE ROBOTS:** Uncomment the full **Robot{2}** structure. Set **Sensor{2}.robot = 2** to assign sensor 2 to robot 2. Get multi-robot centralized SLAM. Read [15].

[2] Framed homogeneous points as described in [1]

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, carrying a different number of sensors of different kinds, which gather raw data and, by processing it, are capable of 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. See Table 2 to see the object types and options that are currently implemented.

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*.

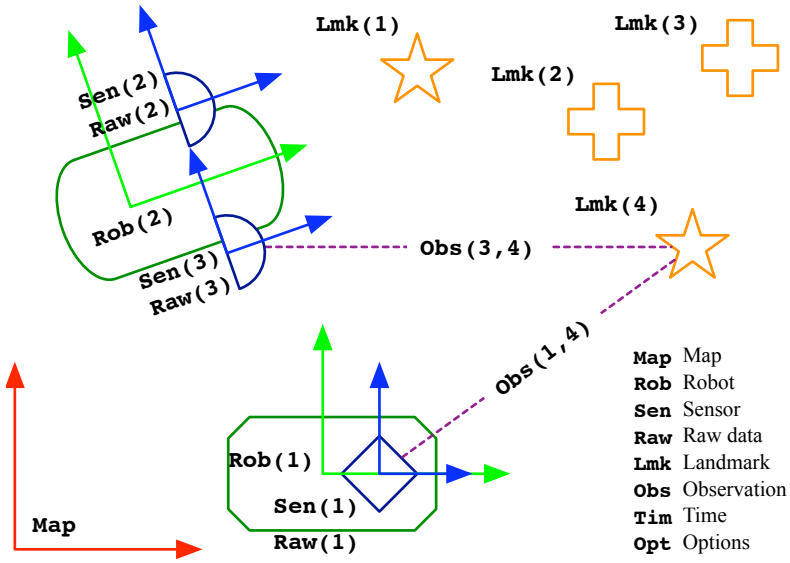

Figure 1: Overview of the SLAM problem with the principal data structures.

Table 1: All data structures.

Purpose	SLAM	Simulator	Graphics
Map	**Map**		**MapFig**
Robots	**Rob**	**SimRob**	
Sensors	**Sen**	**SimSen**	**SenFig**
Raw data	**Raw**		
Landmarks	**Lmk**	**SimLmk**	
Observations	**Obs**		
Time	**Tim**		
Options	**Opt**	**SimOpt**	**FigOpt**

Table 2: Supported object types

Object class	type	Variable	value
Robot motion	odometry	`Rob.motion`	`'odometry'`
Robot motion	constant velocity	`Rob.motion`	`'constVel'`
Sensor	pin-hole camera	`Sen.type`	`'pinHole'`
Sensor	omni-dir. camera	`Sen.type`	`'omniCam'`
Landmark	Euclidean point	`Lmk.type`	`'eucPnt'`
Landmark	Homogeneous point	`Lmk.type`	`'hmgPnt'`
Landmark	Anchored homog. point	`Lmk.type`	`'ahmPnt'`
Landmark	Inverse-depth point	`Lmk.type`	`'idpPnt'`
Landmark	Framed homog. point	`Lmk.type`	`'fhmPnt'`
Landmark	Plucker line	`Lmk.type`	`'plkLin'`
Landmark	Anchored Plucker line	`Lmk.type`	`'aplLin'`
Landmark	Homogeneous line	`Lmk.type`	`'hmgLin'`
Landmark	Anchored homogeneous line	`Lmk.type`	`'ahmLin'`
Landmark	Inverse-depth line	`Lmk.type`	`'idpLin'`

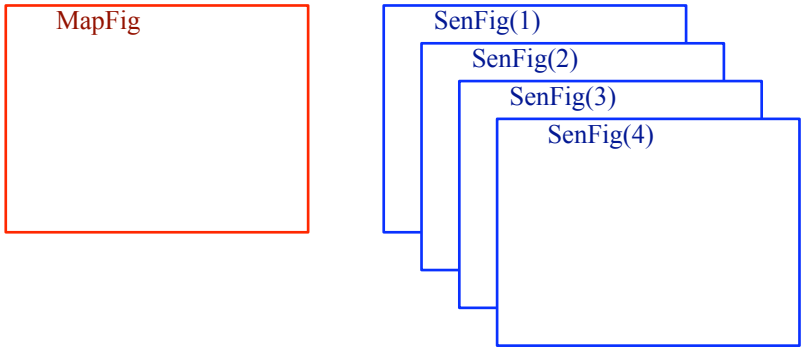

Figure 2: The set of figures. The structures **`MapFig`** and **`SenFig(s)`** contain the handles to all graphics objects drawn.

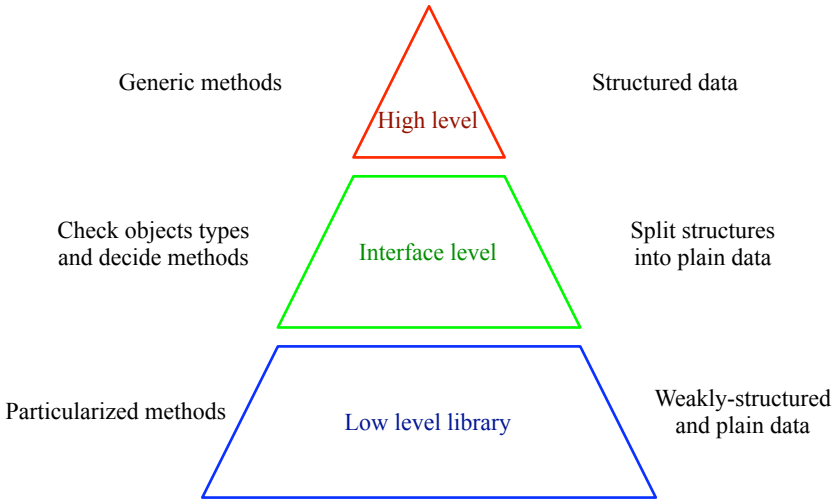

Figure 3: Overview of the levels of abstraction of the functions and their relation to data structuration. Functions and scripts in the High and Interface levels are in the **`HighLevel/`** and **`InterfaceLevel/`** directories. The Low Level library occupies all other directories.

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.

`Raw`: A set of raw data captures, one per sensor.

`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 processing **`Raw`** data.

`Tim`: A few time-related variables.

`Opt`: Algorithm options.

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 11 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`** (type **`help frame`** at the Matlab prompt for help on 3D reference frames). Observations require two indices because they relate sensors to landmarks. Thus, **`Obs(sen,lmk)`** stores the data associated to the observation of landmark **`lmk`** from sensor **`sen`**.

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(rob)       % 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
        .uStd      % standard deviation of u
        .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
        .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.

```
Sen(sen)       % Sensor structure, containing:
    .sen         % index in Sen() array
    .id          % sensor id
    .name        % sensor name
    .type        % sensor type
    .robot       % robot it is installed to
    .frameInMap  % flag: is frame in Map?
    .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
        .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
        .pixErr    % pixel error std.
        .pixCov    % pixel covariances matrix
    .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
    .imGrid      % image grid for active initialization
        .imSize    % image size - copy of par.imSize
        .numCells  % number of cells [H,V]
        .skipOuter % flag to skip outer cells
        .usedCell  % boolean matrix of flags indicating used cells
        .xticks    % x-coordinates of cell corners
        .yticks    % y-coordinates of cell corners
```

Figure 5: The **Sen** structure array.

```
Raw(sen)       % Raw data structure, containing:
  .type          % type of raw data
  .data          % raw data, containing
    .points        % 3D point landmarks (for simulated data)
      .coord         % a matrix of points
      .app           % a vector of appearances
    .segments      % 3D segment landmarks (for simulated data)
      .coord         % a matrix of segments (two endpoints, stacked)
      .app           % a vector of appearances
    .img           % a pixels image (for real images)
```

Figure 6: The **Raw** structure array.

```
Lmk(lmk)       % Landmark structure, containing:
    .lmk         % index in Lmk() array
    .id          % landmark id
    .type        % sensor type
    .sig         % landmark descriptor or signature
    .used        % flag: is landmark used in the map?
    .state       % state structure, containing
      .r           % range in the SLAM map Map
    .par         % other lmk parameters
      .endp()      % 2 endpoints for segments
        .t           % abscissa
        .e           % endpoints mean
    .nSearch     % number of times searched
    .nMatch      % number of times matched
    .nInlier     % number of times declared inlier
```

Figure 7: The **Lmk** structure array.

```
Map            % Map structure, containing:
    .used        % vector of flags indicating non-free positions
    .x           % state vector's mean
    .P           % covariances matrix
```

Figure 8: The **Map** structure.

```
Obs(sen,lmk)  % Observation structure, containing:
    .sen        % index to sensor in Sen() array
    .lmk        % index to landmark in Lmk() array
    .sid        % sensor id
    .lid        % landmark id
    .stype      % sensor type
    .ltype      % landmark type
    .meas       % measurement
        .y        % mean
        .R        % covariance
    .nom        % non-measurable degrees of freedom
        .n        % mean
        .N        % covariance
    .exp        % expectation
        .e        % mean
        .E        % covariance
        .um       % uncertainty measure, um = det(E)
    .inn        % innovation
        .z        % mean
        .Z        % covariance
        .iZ       % inverse covariance
        .MD2      % squared Mahalanobis distance, MD2 = z'*iZ*z
    .app        % appearance
        .pred     % predicted appearance
        .curr     % current appearance
        .sc       % matching quality score
    .par        % other parameters
        .endp()   % two segment endpoints
            .e      % mean
            .E      % covariance
    .vis        % flag: is lmk visible from sensor?
    .measured   % flag: has lmk been measured?
    .matched    % flag: has lmk been matched?
    .updated    % flag: has Map been updated?
    .Jac        % Jacobians
        .E_r      % expectation wrt robot frame vector
        .E_s      % expectation wrt sensor frame vector
        .E_l      % expectation wrt landmark parameters
        .Z_r      % innovation wrt robot frame vector
        .Z_s      % innovation wrt sensor frame vector
        .Z_l      % innovation wrt landmark parameters
```

Figure 9: The **Obs** structure array.

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

Figure 10: The **Tim** structure.

```
Opt                    % Options structure, containing:
    .map                 % Options for the map
        .numLmks           % map capacity: number of 3d landmarks
        .lmkSize           % nominal lmk size (for map size estimation)
    .correct             % Options for lmk correction
        .reprojectLmks     % reproject lmks after active search?
        .rearametrize      % reparametrize landmark?
        .nUpdates          % maximum simultaneus updates
        .MD2th             % threshold on Mahalanobis distance
        .linTestIdp        % threshold on IDP linearity test
        .lines             % line landmarks correction
            .innType         % innovation type for lines
            .extPolicy       % endpoints extension policy
            .extSwitch       % policy switching threshold
    .init                % Options for initialization
        .nbrInits          % Number of inits [first frame , other frames]
        .initType          % Type of landmark to initialize
        .idpPnt            % options for inverse-depth based landmarks
            .nonObsMean      % mean of non-observable prior
            .nonObsStd       % std dev. of non-observable prior
        .plkLin            % Plucker based lines
            .nonObsMean
            .nonObsStd
    .obs                 % Observation options
        .lines             % options for lines or segments
            .minLength       % minimum segment length
```

Figure 11: The **Opt** structure.

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.

`SimOpt`: Options for the simulator.

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(rob)    % 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
        .Pi        % PI matrix, Pi = q2Pi(q)
        .Pc        % conjugate PI matrix, Pc = pi2pc(Pi)
    .vel         % velocity stucture, containing
        .x         % 6-vector, linear and angular velocities
    .con         % Control vector
        .u         % control signals for the motion model
        .uStd      % standard deviation of u
        .U         % covariance of u
```

Figure 12: The **SimRob** structure array.

```
SimSen(sen)    % 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
        .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 13: The **SimSen** structure array.

```
SimLmk          % Simulated landmarks structure, containing:
    .points      % Point landmarks
        .id        % N-vector of point identifiers
        .coord     % 3-by-N array of 3D points
    .segments    % segment landmarks
        .id        % M-vector of segment identifiers
        .coord     % 6-by-M array of 3D segments
    .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 14: The **SimLmk** structure.

```
SimOpt            % Simulator options structure, containing:
    .random          % random generator options
        .newSeed     % use true random generator?
        .fixedSeed   % random seed for non-random runs
        .seed        % actual seed
    .obs           % options for simulated observations.
                   % (this is a hard-copy of Obs.obs)
```

Figure 15: The **SimOpt** structure

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. 16 and 17).

`SenFig`: A structure array of handles to graphics objects in the sensor figures. One figure per sensor, visualizing its *measurement space* (Figs. 18 and 19).

`FigOpt`: A structure with options for figures such as colors, views and projections.

It follows a reproduction of the arborescences of the principal graphics structures. See Section 5.7 for information about graphic functions.

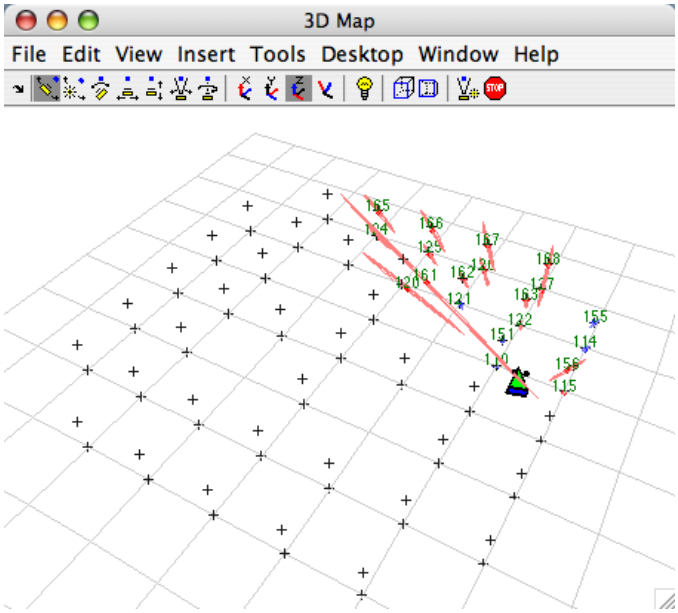

Figure 16: The 3D map figure. **`MapFig`** contains handles to all objects drawn.

```
MapFig          % Map figure structure, containing:
    .fig          % figure number and handle
    .axes         % axes handle
    .ground       % handle to floor object
    .Rob          % array of structures to SLAM robot handles
        .patch      % handle to robot graphics patch
        .ellipse    % handle to robot's uncertainty ellipsoid
    .Sen          % array of handles to SLAM sensors
    .Lmk          % handles to SLAM landmarks, containing:
        .drawn      % array of flags indicating drawn landmarks
        .mean       % array of handles to landmarks means
        .ellipse    % array of handles to landmarks ellipses
        .label      % array of handles to landmarks labels
    .simRob       % array of handles to simulated robots
    .simSen       % array of handles to simulated sensors
    .simLmk       % handle to simulated landmarks
```

Figure 17: The **`MapFig`** structure.

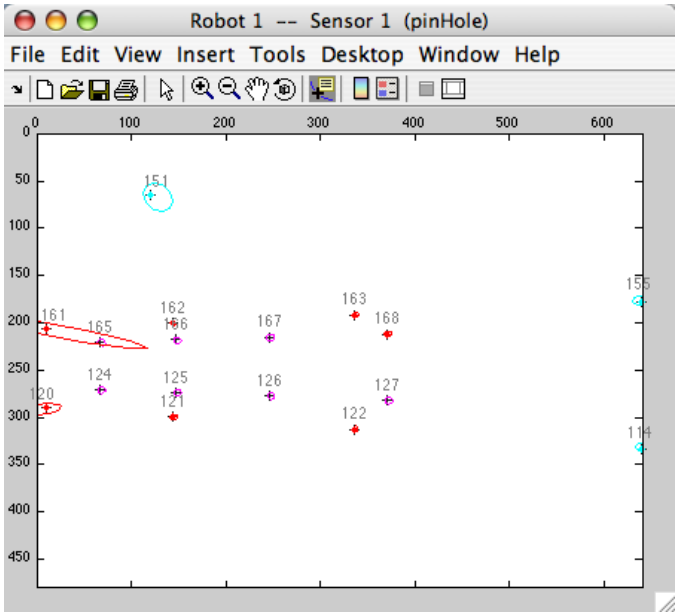

Figure 18: A pin-hole sensor view figure. **`SenFig(1)`** contains handles to all objects drawn.

```
SenFig(sen)     % Sensor figure structure, containing:
    .fig          % figure number and handle
    .axes         % axes handle
    .raw          % handles to raw data
        .points     % handle to one line object for all raw points
        .segments   % array of handles to line objects for raw segments
    .drawn        % vector of flags indicating drawn observations
    .measure      % array of handles to landmarks measurements
    .mean         % array of handles to predicted means
    .ellipse      % array of handles to predicted ellipses
    .label        % array of handles to landmarks labels
```

Figure 19: The **`SenFig`** structure array.

```
FigOpt              % Figure options structure, containing:
  .renderer           % renderer
  .rendPeriod         % rendering period in frames
  .createVideo        % flag: create video sequence?
  .map                % map figure options
    .proj               % projection of the 3d figure
    .view               % viewpoint of the 3d figure
    .orbit              % AZ and EL orbit angle increments
    .size               % map figure size
    .showSimLmk         % flag: show simulated landmarks?
    .showEllip          % flag: show uncertainty ellipsoids?
    .colors             % map figure colors
      .border           % border
      .axes             % axes, ticks and axes labels
      .bckgnd           % background
      .simLmk           % simulated landmarks
      .defPnt           % default point
        .mean             % mean dot
        .ellip            % ellipsoid
      .othPnt           % other point
        .mean             % mean dot
        .ellip            % ellipsoid
      .defLin           % default line
        .mean             % mean line
        .ellip            % endpoint ellipsoids
      .simu             % simulated robots and sensors
      .est              % estimated robots and sensors
      .ground           % ground
      .label            % landmark ID labels
  .sensor             % sensor figures options
    .size               % sensor figure size
    .showEllip          % flag: show uncertainty ellipses?
    .colors             % Sensor figure colors:
      .border             % border
      .axes               % axes, ticks and axes labels
      .bckgnd             % background
      .raw                % raw data
      .defPnt             % euclidean point
        .updated            % updated
        .predicted          % predicted
      .othPnt             % other point
        .updated            % updated
        .predicted          % predicted
      .defLin             % default line
        .meas               % measurement
        .mean               % mean line
        .ellip              % endpoint ellipses
      .othLin             % other line
        .meas               % measurement
        .mean               % mean line
        .ellip              % endpoint ellipses
      .label              % label
```

Figure 20: The **FigOpt** structure.

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/tmp/'`.

frames: Frames are Matlab structures that we created to store data belonging to 3D frames (see Fig. 21 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'
    .Pi     % PI matrix,              Pi = q2Pi(q)
    .Pc     % conjugate PI matrix,    Pc = q2Pi(iq)
```

Figure 21: The `frame` structure.

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, one for the code and one for the data:

`slamtb.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.

Here is a simplified version of this script:

```
% SLAMTB  EKF-SLAM simulator. Main script.

% User-defined data.
userData;

% Create all data structures
[Rob, Sen, Lmk, Obs, Tim] = createSlamStructures(...
                            Robot, Sensor, Landmark, Time, ...
                            Opt);
[SimRob, SimSen, SimLmk]  = createSimStructures(...
                            Robot, Sensor, World,...
                            SimOpt);
[MapFig, SenFig]          = createGraphicsStructures(...
                            Rob, Sen, Lmk, Obs, ...
                            SimRob, SimSen, SimLmk, ...
                            FigOpt);

% Main loop
for currentFrame = Tim.firstFrame : Tim.lastFrame

    % 1. SIMULATION
```

```
    for rob = [SimRob.rob]

        % Simulate robot motion
        SimRob(rob) = simMotion(SimRob(rob), Tim);

        for sen = SimRob(rob).sensors

            % Simulate measurements
            SimObs(sen) = simObservation(...
                SimRob(rob), SimSen(sen), SimLmk) ;
        end

    end

    % 2. SLAM
    for rob = [Rob.rob]

        % Robot motion
        Rob(rob) = motion(Rob(rob),Tim);

        for sen = Rob(rob).sensors

            % Correct known landmarks
            [Rob(rob), Sen(sen), Lmk, Obs(sen,:)] = ...
                correctKnownLmks(...
                Rob(rob),Sen(sen),Raw(sen),Lmk,Obs(sen,:));

            % Initialize new landmarks
            [Lmk, Obs(sen,:)] = initNewLmks(...
                Rob(rob),Sen(sen),Raw(sen),Lmk,Obs(sen,:));
        end

    end

    % 3. VISUALIZATION
    drawMapFig(MapFig, Rob, Sen, Lmk, SimRob, SimSen);
    for sen = [Sen.sen]
        drawSenFig(SenFig(sen),Sen(sen),Raw(sen),Obs(sen,:));
    end
    drawnow;
end
```

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

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

```
createSLAMstructures()          % Create SLAM structures
createSimStructures()           % Create simulation structures
createGraphicsStructures()      % Create graphics structures
```

The main purpose of these functions is to take the data from **userData**, which is just what the user needs to enter, and create with them the more complete structures that the program will use.

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.

Interface-level functions perform the different simulation, SLAM, and redraw operations. They are called inside the main loop:

```
% Simulator
simMotion()                     % Simulate motions
simObservation()                % Simulated observations
% SLAM
motion()                        % Robot motion
observeKnownLmks()              % EKF-update of known landmarks
initNewLmk()                    % Landmark initialization
% Visualization
drawMapFig()                    % Redraw 3D Map figure
drawSenFig()                    % Redraw sensors figures
```

Other intermediate-level functions create all graphics figures. They are called by **createGraphicsStructures.m**:

```
createMapFig()                  % Create 3D Map figure
createSenFig()                  % Create all sensors' figures
```

A good example of interface function is **simMotion.m**, whose code is reproduced in Fig. 22.

```
function Rob = simMotion(Rob, Tim)
% SIMMOTION  Simulated robot motion.

switch Rob.motion      % check robot's motion model

    case  'constVel'
        Rob.state.x  =  constVel(Rob.state.x,Rob.con.u,Tim.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,Rob.con.u);

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

Figure 22: The **simMotion.m** interface function. 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) in **case 'constVel'**, **frame.x** is modified by the low-level motion functions, and we need a call to **updateFrame()** 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
Lines/                   % Line landmarks
Math/                    % Some math functions
Observations/            % Observation models
Points/                  % Point landmarks
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.

To explore the contents of the library, start by typing `help DirectoryName` at the Matlab prompt.

5 Developing new observation models

This section describes the necessary steps for creating new observation models any time a new type of sensor and/or a new type of landmark is considered. Please read '**guidelines.pdf**' before contributing your own code.

Before you develop a new observation model, you must take care of the following facts:

1. You need a *direct observation model* for observing known landmarks and correcting the map, and an *inverse observation model* for landmark initialization.
2. The robot acts as a mere support for sensors. Normally, only its current frame is of any interest. In some (rare) special cases, the robot's velocity may be of interest if the measurements are sensitive to it (for example, when considering a sonar sensor with Doppler-effect capabilities).
3. The sensor's frame is specified in robot frame. It may be part of the SLAM state vector.
4. The sensor contains other parameters. The number and nature of these parameters depend on the type of sensor and cannot be generalized. We have not considered these parameters as part of the SLAM state vector, although this could be done. Most observation functions in the toolbox already return the Jacobians with respect to these parameters.
5. The landmark has the main parameters in the SLAM vector, but it may have some other parameters out of it.
6. The sensor may provide full or partial measurements of the landmark state. In case of partial measurements, you have to provide a Gaussian prior of the non-measured part for initialization.

5.1 Practical error-free procedure for the lazy-minded

To start the creation of a new model, simply edit **userData.m** and enter a new string in **Opt.init.initType** (for example, enter the string **'newLin'** to create a new line. When creating a new landmark model, remember to use strictly 6 characters, with the last 3 indicating either **'Pnt'** or **'Lin'**!. Then execute **slamtb**. The program will fail exactly in the place where you

need to enter new code. Let me call this an *entry point*. Read the error comment and check the existing code close to the entry point: get inspired on it and you will soon know what to do, and how to do it. After adding the code (and saving the corresponding file!), re-execute **slamtb** to advance to the next entry point.

For example, when entering `'newLin'` in **Opt.init.initType** in **userData.m**, the following error occurs:

```
??? Error using ==> initNewLmk at 39
??? Unknown landmark type 'newLin'.

Error in ==> slamtb at 130
            [Lmk,Obs(sen,:)] = initNewLmk(...
```

In this case, you just need to edit **initNewLmk.m**, go to line 39, and add a new **case** `'newLin'` entry within the **switch** **Opt.init.initType** statement. Within this **switch** statement, you observe that the size of the landmark parametrization is being defined. Enter a line of code with the appropriate size; for example, if your new line parametrization contains 10 states, write

```
case 'newLin'
    lmkSize = 10;
```

Then re-execute **slamtb**. You have advanced to the second entry point!!!

Unfortunately, some code you need to write is not as trivial as the example above. The following paragraphs explain in more detail some of the steps you need to perform.

5.2 Steps to incorporate new models

Once you decide to incorporate new models, follow these steps:

1. Write direct and inverse observation models for your landmark and sensor. See Sections 5.3 and 5.4.

2. Edit **userData.m**. Add a number of new landmarks in structure **World**. Type **help userData** and explore the comments within its code to learn how to achieve this.

3. Edit **simObservation.m** for simulating landmark measurements. Add new case lines **case 'my_Sen'** and **case 'my_Lmk'**, and write the necessary code that calls the functions in your direct model. These function calls do not request Jacobians.

4. Edit **initNewLmk.m** for landmark initialization. Add new case lines **case 'my_Sen'** and **case 'my_Lmk'**, and write the necessary code that calls the functions in your inverse model. See that these function calls request Jacobians.

5. Edit **correctKnownLmks.m** for landmark corrections. Add new case lines **case 'my_Sen'** and **case 'my_Lmk'**, and write the necessary code that calls the functions in your direct model. See that these function calls request Jacobians. Name the return variable and Jacobians exactly as in the other existing models: they are used later. See Sections 5.3, 5.5 and 5.6.

6. Edit **createMapFig.m** and **createSenFig.m** for map and sensor figures. Add new **switch**–**case** entries and the methods to create the desired graphics. See Section 5.7.

7. Edit **drawMapFig.m** and **drawSenFig.m** to redraw the new landmarks in the map and sensor figures. Add new **switch**–**case** entries and create the desired methods for showing the landmarks and associated observations. See Section 5.7.

5.3 Direct observation model for map corrections

The observation operations are split into three stages: transformation to robot frame, transformation to sensor frame, and projection into the sensor's measurement space. The model takes the general form **e = h(Rf, Sf, Sp, l)**, with **Rf** the robot frame, **Sf** the sensor frame, **Sp** the sensor parameters, **l** the landmark parameters, and **e** the expected measurement or projection (in the EKF argot, **e = h(x)**).

We have basically two options for its implementation: building from scratch or adapting an existing model.

5.3.1 Build model from scratch

Here is a simplified implementation:

```
function e = observationModel(Rf,Sf,Sp,l)
% IN - Rf: robot frame
%      Sf: sensor frame
%      Sp: sensor parameters
%      l : landmark in world frame
% OUT- e : projected magnitude

lr = toFrame(Rf,l);           % landmark in robot frame
ls = toFrame(Rs,lr);          % landmark in sensor frame
e  = projectToSensor(Sp,ls); % projection to sensor's space
```

This shows that we need to create three functions for a direct observation model: **toFrame**, **projectToSensor** and **observationModel**, whose names will be properly particularized for the types of sensor and landmark of the model.

This scheme must be enriched with two important capabilities, namely:

- Jacobian matrices computation.
- Vectorized operation for multiple landmarks.

The following code exemplifies the direct measurement model for a pinhole camera mounted on a robot and observing Euclidean 3D points. Use it as a guide for writing your own models. Notice the systematic use of the chain rule for computing the Jacobians (see '**guidelines.pdf**' for info on the chain rule).

```
function [u, s, U_r, U_s, U_k, U_d, U_l] = ...
    projEucPntIntoPinHoleOnRob(Rf, Sf, Spk, Spd, l)
% IN - Rf : robot frame
%      Sf : sensor frame
%      Spk: pin-hole intrinsic parameters
%      Spd: pin-hole distortion parameters
%      l  : landmark in world frame
% OUT- u  : projected pixel
%      s  : non-measurable depth
%      U_*: Jacobians

if nargout <= 2 % No Jacobians requested
    lr    = toFrame(Rf,l);        % lmk to robot frame
    ls    = toFrame(Sf,lr);       % lmk to sensor frame
    [u,s] = pinHole(ls,Spk,Spd);  % lmk into measurement space

else % Jacobians requested

    if size(l,2) == 1  % single point
```

```
        % Same functions with Jacobian output
        [lr, LR_r, LR_l]    = toFrame(Rf,l);
        [ls, LS_s, LS_lr]   = toFrame(Sf,lr);
        [u,s,U_ls,U_k,U_d] = pinHole(ls,Spk,Spd);

        % Apply the chain rule for Jacobians
        U_lr = U_ls*LS_lr;
        U_r  = U_lr*LR_r;
        U_s  = U_ls*LS_s;
        U_l  = U_lr*LR_l;
    else
        error('??? Jacobians not available for multiple points.')
    end
end
```

The model makes use of the functions **toFrame()** and **pinHole()**. The first function is specific to the landmark type, while the second depends on both the landmark type and the sensor type. We reproduce them here:

```
function [pf, PF_f, PF_p] = toFrame(F, pw)
% IN - F   : frame
%      pw  : point in world frame
% OUT- pf  : point in F-frame
%      PF_*: Jacobians

s = size(p_W,2); % number of points in input matrix

if s==1 % one point
    pf = F.Rt*(pw - F.t);

    if nargout > 1 % Jacobians.
        PF_t  = -F.Rt;
        sc    = 2*F.Pc*(pw - F.t);
        PF_q  = [...
            sc(2)  sc(1) -sc(4)  sc(3)
            sc(3)  sc(4)  sc(1) -sc(2)
            sc(4) -sc(3)  sc(2)  sc(1)];
        PF_p  = F.Rt;
        PF_f  = [PF_t PF_q];
    end

else % multiple points
    pf = F.Rt*(pw - repmat(F.t,1,s));
    if nargout > 1
        error('??? Jacobians not available for multiple points.');
    end
end
```

```
function [u, s, U_p, U_k, U_d] = pinHole(p, k, d)
% IN - p  : 3D point
%      k  : pin-hole intrinsic parameters
%      d  : pin-hole distortion parameters
% OUT- u  : projected pixel
%      s  : non-measurable depth
%      U_*: Jacobians

% Point's depths
s = p(3,:);

if nargin < 3, d = []; end  % Default is no distortion

if nargout <= 2 % no Jacobians requested
    u = pixellise(distort(project(p),d),k);

else % Jacobians

    if size(p,2) == 1    % p is a single 3D point
        [up, UP_p]        = project(p);
        [ud, UD_up, UD_d] = distort(up,d);
        [u, U_ud, U_k]    = pixellise(ud,k);
        U_d               = U_ud * UD_d;
        U_p               = U_ud * UD_up * UP_p;

    else    % p is a 3D points matrix - no Jacobians possible
        error('??? Jacobians not available for multiple points.')
    end
end
```

5.3.2 Adapting an existing model

Adapting an existing model is very easy and interesting. We avoid errors and save a lot of coding time. It is possible if we know the function that transforms one model into another one. For example, if we already have the observation model for Euclidean points, **projEucPntIntoPinHoleOnRob()**, we just need the conversion function **idp2euc()** (see pag. 37), transforming IDP points to Euclidean, and build the observation model as

```
function [u, s, U_r, U_s, U_k, U_d, U_l] = ...
    projIdpPntIntoPinHoleOnRob(Rf, Sf, Spk, Spd, l)

if nargout <= 2 % no Jacobians requested

    % first convert to the existing model's type
```

```
    p = idp2euc(l);

    % then apply the known model
    e = projEucPntIntoPinHoleOnRob(Rf, Sf, Sp, p);

else            % Jacobians requested

    % first convert to the existing model's type
    [p, P_l] = idp2euc(l);

    % second apply the known model
    [u, s, U_r, U_s, U_k, U_d, U_p] = ...
        projEucPntIntoPinHoleOnRob(Rf, Sf, Sp, p);

    % finally apply the chain rule
    U_l = U_p * P_l;
end
```

5.4 Inverse observation model for landmark initialization

The inverse model works inversely to the direct one, with one important detail: for sensors providing partial landmark measurements, a prior is needed in order to provide the inverse function with the full necessary information.

The model takes the general form **`l = g(Rf, Sf, Sp, y, n)`**, with **`Rf`** the robot frame, **`Sf`** the sensor frame, **`Sp`** the sensor parameters, **`y`** the measurement, **`n`** the non-measured prior, and **`l`** the retro-projected landmark parameters. Here is a simplified implementation:

```
function l = invObsModel(Rf, Sf, Sp, y, n)
% IN - Rf: robot frame
%      Sf: sensor frame
%      Sp: sensor parameters
%      y : measurement
%      n : non-measured prior
% OUT- l : obtained landmark

ls = retroProjectFromSensor(Sp, e, n);    % lmk in sensor frame
lr = fromFrame(Sf, ls);                   % lmk in robot frame
l  = fromFrame(Rf, lr);                   % lmk in world frame
```

In this case, only Jacobians computation need to be added, as it is not likely that we need to retro-project several points at a time (contrary to what happens with the direct models).

The following code exemplifies the inverse measurement model for a pin-

hole camera mounted on a robot and observing 3D points, rendering 3D landmarks parametrized as inverse-depth [3]. The inverse depth is precisely the non-measured part **n**, and is provided as a prior with a separate input.[3]

```
function [idp, IDP_rf, IDP_sf, IDP_sk, IDP_sc, IDP_u, IDP_n] = ...
    retroProjIdpPntFromPinHoleOnRob(Rf, Sf, Sk, Sc, u, n)
% IN - Rf : robot frame
%      Sf : sensor frame
%      Sk : sensor intrinsic parameters
%      Sc : sensor distortion correction parameters
%      y  : measurement
%      n  : non-measured prior
% OUT- idp: retro-projected inverse-depth point
%      IDP_*: Jacobians

if nargout == 1 % No Jacobians requested
    idps = invPinHoleIdp(u,n,Sk,Sc) ;
    idpr = fromFrameIdp(Sf, idps) ;
    idp  = fromFrameIdp(Rf, idpr) ;

else            % Jacobians requested
    % function calls
    [idps, IDPS_u, IDPS_n, IDPS_sk, IDPS_sc] = ...
        invPinHoleIdp(u, n, Sk, Sc) ;
    [idpr, IDPR_sf, IDPR_idps] = fromFrameIdp(Sf, idps) ;
    [idp,   IDP_rf,  IDP_idpr] = fromFrameIdp(Rf, idpr) ;

    % The chain rule
    IDP_idps = IDP_idpr * IDPR_idps; % intermediate result
    IDP_sk   = IDP_idps * IDPS_sk ;
    IDP_sc   = IDP_idps * IDPS_sc ;
    IDP_u    = IDP_idps * IDPS_u ;
    IDP_n    = IDP_idps * IDPS_n ;
end
```

5.5 Landmark reparametrization

In case you are using landmarks with a parametrization that is especialized for initialization, such as Inverse Depth points, you must consider reparametrizing them to more economical forms, such as Euclidean points, once they have converged to stable 3D positions. Read [2] if you do not know what I am talking about. In this case, write reparametrization functions and

[3] For speed reasons, the function `retroProjIdpPntFromPinHoleOnRob` is implemented somewhat differently in the toolbox.

include them in the code.

Here is an example of the reparametrization function:

```
function [p,P_idp] = idp2euc(idp)
% IDP2EUC  Inverse Depth to Euclidean point conversion.

x0 = idp(1:3,:);    % origin
py = idp(4:5,:);    % pitch and roll
r  = idp(6,:);      % inverse depth

if size(idp,2) == 1 % one only Idp

    [v,V_py] = py2vec(py);  % unity vector
    p        = x0 + v/r;

    if nargout > 1 % jacobians

        P_x   = eye(3);
        P_v   = eye(3)/r;
        P_r   = -v/r^2;

        P_py  = P_v*V_py;

        P_idp = [P_x P_py P_r];
    end

else  % A matrix of Idps

    v = py2vec(py);  % unity vector
    p = x0 + v./repmat(r,3,1);

    if nargout > 1
        error('??? Jacobians not available for multiple landmarks.')
    end
end
```

To include this function in the code, edit the function **`reparametrizeLmk()`**, create a new **`case`** in the landmark's type **`switch`**, and add your code as appropriated.

Another example of reparametrization is **`hmg2euc()`**. It transforms homogeneous points to the Euclidean space.

5.6 Landmark parameters out of the SLAM map

There may be some landmark parameters that are not part of the stochastic state vector estimated by SLAM. The number and nature of these parame-

ters depend on the landmark type and cannot be generalized.

The direct and inverse observation models must be complemented with the appropriate methods to initialize and update these parameters. In the toolbox, we use the functions **`initLmkParams`** and **`updateLmkParams`**.

Examples of such parameters are:

- The endpoints of segment landmarks. See [8, 17] for examples of setting and updating. Initialization and updating of segment endpoints for lines of the type Plucker are supported in this toolbox. Check functions **`retroProjPlkEndPnts`** and **`updatePlkLinEndPnts`**.
- The landmark's appearance information, used in appearance-based feature matching. See [4, 15] for examples on setting and using these parameters. See [9] for an example of updating them.

5.7 Graphics

Each new landmark needs its own drawing methods and, possibly, dedicated graphic data structures.

5.7.1 Graphic handles

Graphics are managed with Matlab *handles*. If you do not know about handles, we give here a basic approach. In brief, when you create a graphics object you assign it a handle that will allow you to manipulate the object.[4] After that, to redraw the object you only need to update the values in that handle. (You update the values that have changed, and leave the rest unchanged. The alternative to plot at each frame the whole graphic is not time efficient.) Here is an example of a moving graphic using handles:

```
% create a fancy figure for our demo
f1 = figure(1);                % figure handle 'f1'
set(f1, 'renderer', 'opengl') % we modify the figure's renderer
ax = gca ; cla ; axis equal ; % axes handle 'ax'
axis([-1.1 1.1 -1.1 1.1])

% we create here the graphics object: a single-point line
```

[4] Yes, handles are there always to manipulate objects. As well as *handle* comes from the English root '*hand*', the word *manipulate* comes from the Latin root '*manus*' for *hand*, and means "*handle or control (a mechanism, tool, etc.), typically in a skillful manner*" (Oxford). It is no language abuse to say that a cup's handle allows us to manipulate the cup without getting burned.

```
angle0       = 0;
objecthandle = line( ...
    cos(angle0),   sin(angle0), ...
    'parent',      ax,  ...
    'marker',      'o', ...
    'color',       'r', ...
    'markersize',   10);

for angle = angle0:0.01:(angle0+2*pi)
    % we change only its position
    set(objecthandle, ...
        'xdata', cos(angle), ...
        'ydata', sin(angle));
    drawnow
end
```

We basically use two types of graphics objects: **line** and **patch**. To know the properties of each object you can access to, simply create a default object with *e.g.* **h = line()**, obtaining the line's handle **h**, and then type **set(h)**. You will get a list of all possible properties with their default values. To know the properties's values, type **get(h)**. To modify a particular property, type **set(h,'propertyName',value)**. To read a particular value, use **get(h,'propertyName')**.

5.7.2 Graphic functions

In the toolbox, objects are crated in **createMapFig** and **createSenFig**, invoked by **createGraphicsStructures** before the main loop. They are updated in the third part of the loop, with **drawMapFig** and **drawSenFig**. Visit these functions, add new **switch**−**case** entries for your objects, and code the necessary methods.

Bear in mind that, while new landmark parametrizations require new drawing methods for the 3D part (the map figure), once they are projected into a particular sensor they may end up having the same **Obs** structure as other existing landmarks. For instance, Euclidean, IDP and homogeneous points all project into 2D points in the image. This means that we will find 3D-drawing functions **drawEucLmk**, **drawIdpLmk** and **drawHmgLmk**, but we will find only one function **drawObsPnt** for all of their 2D projections.

The following examples show how to redraw a 3D ellipsoid belonging to an Euclidean landmark, and how to redraw a projected 2D segment in a pin-hole image. Use them as templates for your own methods. The first function **drawEucPnt** draws a Euclidean 3D point in the Map figure:

```
function drawEucPnt(MapFig, Lmk, color)
% DRAWEUCPNT  Draw Euclidean point landmark in MapFig.

global Map
posOffset = [0;0;.2];

% Mean and covariance
x = Map.x(Lmk.state.r);
P = Map.P(Lmk.state.r,Lmk.state.r);

% draw
drawPnt    (MapFig.Lmk(Lmk.lmk).mean,    x,    color.mean)
drawEllipse(MapFig.Lmk(Lmk.lmk).ellipse, x, P, color.ellip)
drawLabel  (MapFig.Lmk(Lmk.lmk).label,   x+posOffset, num2str(Lmk.id))
```

It calls **drawPnt**, **drawEllipse** and **drawLabel** which are used to draw a labeled Gaussian 3D point. We show here **drawEllipse**:

```
function drawEllipse(h, x, P, c, ns, NP)
% DRAWELLIPSE Draw 2D ellipse or 3D ellipsoid.
% IN - h  : ellipse handle
%      x  : mean
%      P  : covariance
%      C  : color
%      ns : number of sigmas
%      NP : number of points to draw the ellipse

if nargin < 6,         NP = 10;
    if nargin < 5,     ns = 3;
    end
end

if numel(x) == 2        % 2D ellipse
    [X,Y] = cov2elli(x,P,ns,NP);
    set(h,'xdata',X,'ydata',Y,'vis','on')

elseif numel(x) == 3    % 3D ellipsoid
    [X,Y,Z] = cov3elli(x,P,ns,NP);
    set(h,'xdata',X,'ydata',Y,'zdata',Z,'vis','on')

else
    error('??? Size of vector ''x'' not correct.')
end

if nargin ≥ 3 && ~isempty(c)
    set(h,'color',c)
end
```

This other function draws an observed line in the pin-hole sensor figure:

```
function drawObsLin(SenFig, Obs, colors, showEllip)
% DRAWOBSLIN  Draw an observed line on the pinHole sensor figure.

posOffset = 8;

% the measurement:
if Obs.measured
    y = Obs.meas.y;
    drawSeg(SenFig.measure(Obs.lmk),y,colors.meas)

    % the label
    c = (y(1:2)+y(3:4))*0.5;      % segment's center
    v = (y(1:2)-y(3:4));          % segment's vector
    n = normvec([-v(2);v(1)]);    % segment's normal vector
    pos = c + n*posOffset;
    drawLabel(SenFig.label(Obs.lmk), pos, num2str(Obs.lid))
else
    set(SenFig.measure(Obs.lmk), 'vis', 'off');
    set(SenFig.label(Obs.lmk),   'vis', 'off');
end

% the expectation
xlim = get(SenFig.axes,'xlim');
ylim = get(SenFig.axes,'ylim');
imSize = [xlim(2);ylim(2)];
s = trimHmgLin(Obs.exp.e, imSize);
if ~isempty(s)
    % mean
    drawSeg(SenFig.mean(Obs.lmk),s,colors.mean)

    % ellipses
    if showEllip
        drawEllipse(SenFig.ellipse(Obs.lmk,1), ...
            Obs.par.endp(1).e, ...
            Obs.par.endp(1).E, ...
            colors.ellip)
        drawEllipse(SenFig.ellipse(Obs.lmk,2), ...
            Obs.par.endp(2).e, ...
            Obs.par.endp(2).E, ...
            colors.ellip)
    end
else % not visible
    set(SenFig.mean(Obs.lmk),      'vis', 'off');
    set(SenFig.ellipse(Obs.lmk,:), 'vis', 'off');
end
```

6 Extensions

You want to extend the capabilities of this toolbox. Here are some suggestions:

Combined points and lines: SLAM using both points and lines. This should be relatively easy to implement, but at this time the toolbox does not support it. If you want to try, check and modify the feature detection and landmark initialization sections (all in **`initNewLmks.m`**). Updates and rendering should work without problems.

Multi-map: Multi-map operation for consistent large-scale SLAM. Basically, you should surround the main loop in **`slamtb.m`** with another loop managing the closing and creation of maps and performing the multi-map loop closures.

Other sensors: Use of sensors other than the pin-Hole camera. This is a matter of building new observation models as explained in Section 5.

Real images: Use of real data, avoiding all the simulation part. Refer to the following section for general guidelines.

6.1 Extension with real images

The toolbox does not support working with real images; you need to implement it.

Basically, you should remove the whole SIMULATION section in **`slamtb.m`** and substitute it by a few lines of code loading an image into **`Raw(sen).data.img`**, setting **`Raw(sen).type`** to `'real'` or `'image'`. Then, you need to code the feature detection and matching functions for real images, inserting them in **`initNewLmk.m`** and **`matchFeature.m`** respectively.

6.1.1 Working with images

To load an image into the Raw structure, you can simply do:

```
Raw(sen).data.img = imread(imageFileName.ext);
```

However, it might be more efficient to declare the image global, like:

```
global currentImage
[...]
```

```
currentImage{sen} = imread(imgFileName.ext);
```

and ignore **Raw(sen).data.img**. This will speed up your algorithm. I advise you to use a cell array **currentImage{sen}** to be able to handle more than one camera automatically, even if the images from each camera have different sizes.

You may want to be sure that the image is grayscale, with just one channel, and with just 8 bits depth. You can convert the image to any format you want using standard Matlab functions.

6.1.2 Some image processing tools

For what concerns the image processing, the toolbox has a few functions, located in directory **DetectionMatching/**, that might help you in doing this without the need of the Matlab's Image Processing toolbox:

harris_strongest.m gives you the strongest Harris point in an image. You normally give it a fraction of the image where you know no landmarks are tracked, and ask the function to give you one point there. This is used for landmark detection just before initialization.

pix2patch.m extracts a rectangular image patch centered on a pixel. In order to speed up some computations of the ZNCC, it is useful to define the patch structure as follows: **patch.I** is the patch image (of 9×9 to 15×15 pixels as detailed below). **patch.SI** is the sum of all pixels. And **patch.SII** is the sum of the squares of the pixels. This function does the job for you.

zncc.m is the zero-mean normalized correlation coefficient, used for feature matching. It basically compares 2 patches and gives you a score of similarity in the range [0 1].

imresize2.m is used to modify the reference patch before correlation, so that the appearance of the reference patch resembles at maximum that of the feature to match on current image. You use the estimated rotation and distance variation to infer zoom and rotation parameters for patch resizing. This function may need some improvements to work fully satisfactorily, but it is good enough for a start. If you have the Image processing toolbox, you can try **imtransform.m** instead.

patchResize.m uses **imresize2.m** to create a modified version of the structure **patch**.

6.1.3 The active-search algorithm

The work of implementing SLAM with real images is long. I recommend you read the literature about "active search" techniques [4, 6, 15]. The active search algorithm takes care of the following:

1. At initialization time (inside **`initNewLmk.m`**)
   (a) Define a grid in the image plane, for example 5x5 cells dividing the image in equal subimages.
   (b) Project all mapped landmarks into the image. For each projected landmark, set the grid cell where it has been projected to **`true`** or **`'occupied'`**.
   (c) When done, randomly select one only cell which is not **`'occupied'`**. Extract the image of that cell.
   (d) Detect the best Harris point in this image cell, with **`harris_strongest.m`**. Set the resulting pixel **`pix`** as the landmark measurement with **`Obs.meas.y = pix`**.
   (e) Check if the detected point is good enough by putting a threshold on the harris score returned. This "feature quality threshold" needs to be created beforehand. A good place to store it is in **`Obs.init.featQualityTh`**, defined in **`userData.m`**.
   (f) If successful, store a 9×9 to 15×15 pixels patch around the detected point, and the current camera position and orientation, in a "signature" structure in **`Lmk.sig`**, that is, create **`Lmk.sig.patch`** and **`Lmk.sig.pose0`**.
   (g) Proceed to initialization (this is already coded in **`initNewLmk.m`**).

   basically, you enter code 1a—1f in the **`case 'real'`** within **`initNewLmk.m`**.

2. At correction time (inside **`correctKnownLmks.m`**, and then inside **`matchFeature.m`**)
   (a) Project all mapped landmarks (this is already done).
   (b) Select the set of landmarks to observe (done).
   (c) For each selected observation, proceed as follows.
   (d) Define a rectangular search region based on the projection mean **`Obs.exp.x`** and the 3-sigma ellipse **`Obs.inn.P`**. The mean is the center, and the square roots of the diagonals of the covariance are the sigmas, σ_u and σ_v. You need to build a rectangle that goes $\pm 3\sigma$ at each side of the center.

(e) Using the current camera position and orientation, and the stored position and orientation in **`Lmk.sig.pose0`**, compute a zooming factor and a rotation to be applied to the stored patch **`Lmk.sig.patch`** before scanning. Store this predicted appearance in **`Obs.app.pred`**.

(f) Scan the rectangular region for the modified patch, using **`zncc.m`** as the preferred ZNCC correlation score. Store the best match patch in **`Obs.app.curr`**, and the score in **`Obs.app.sc`**. Store the best pixel as the measurement, in **`Obs.meas.y`**.

(g) On completion, test if the ZNCC score is greater than a threshold (0.95 minimum). This "appearance score threshold" needs to be created beforehand. A good place to store it is in **`Obs.correct.appScTh`**, defined in **`userData.m`**.

(h) Proceed with landmark correction (already done).

basically, you need to code 2c—2g inside **`matchFeature.m`**. Use **`switch / case`** statements to select which kind of processing you will do depending on the **`Raw.type`** data being **`'real'`**, **`'simu'`**, etc.

It is long but doable. Good luck!

7 Bibliography selection

The following publications list is of mandatory reading for anyone wishing to understand/use/contribute to this toolbox.

- One article by myself and colleagues demonstrating performances of several landmark parametrizations for monocular EKF-SLAM: [16].
- Articles by myself and colleagues about monocular SLAM and the extensions to multi-camera: [14, 13, 15, 17].
- My thesis [12], but not in all its extension. Read Chapter 6, and particularly section 6.5 on Active Search.
- Articles by Andrew J. Davison, the most important contributor to monocular EKF-SLAM, and his colleagues at Oxford, London and Cambridge: [4, 5, 6, 9, 7].
- Articles from the University of Zaragoza, mostly on EKF-SLAM. Landmark initialization using inverse-depth parametrization, and multi-map SLAM for large environments: [10, 2, 3, 11].

If this list is too long for your available time or patience, try this reduced version of just 3 titles:

1. Davison on monocular SLAM [4].
2. Solà on landmark parametrizations for points and lines [16].
3. Solà on multi-camera SLAM [15].

References

[1] Simone Ceriani, Daniele Marzorati, Matteo Matteucci, Davide Migliore, and Domenico Giorgio Sorrenti. On feature parameterization for ekf-based monocular slam. In *IFAC WC*, 2011.

[2] Javier Civera, Andrew J. Davison, and José María Martínez Montiel. Inverse Depth to Depth Conversion for Monocular SLAM. In *IEEE Int. Conf. on Robotics and Automation*, pages 2778 –2783, April 2007.

[3] Javier Civera, Andrew J. Davison, and José María Martínez Montiel. Inverse depth parametrization for monocular SLAM. *IEEE Trans. on Robotics*, 24(5), 2008.

[4] Andrew J. Davison. Real-time simultaneous localisation and mapping with a single camera. In *Int. Conf. on Computer Vision*, volume 2, pages 1403–1410, Nice, October 2003.

[5] Andrew J. Davison. Active search for real-time vision. *Int. Conf. on Computer Vision*, 1:66–73, 2005.

[6] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. MonoSLAM: Real-time single camera SLAM. *Trans. on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, June 2007.

[7] Ethan Eade and Tom Drummond. Scalable monocular SLAM. *IEEE Int. Conf. on Computer Vision and Pattern Recognition*, 1:469–476, 2006.

[8] Thomas Lemaire and Simon Lacroix. Monocular-vision based SLAM using line segments. In *IEEE Int. Conf. on Robotics and Automation*, pages 2791–2796, Rome, Italy, 2007.

[9] Nicholas Molton, Andrew J. Davison, and Ian Reid. Locally planar patch features for real-time structure from motion. In *British Machine Vision Conference*, 2004.

[10] José María Martínez Montiel, Javier Civera, and Andrew J. Davison. Unified inverse depth parametrization for monocular SLAM. In *Robotics: Science and Systems*, Philadelphia, USA, August 2006.

[11] L. M. Paz, Pedro Piniés, Juan Domingo Tardós, and José Neira. Large scale 6DOF SLAM with stereo-in-hand. *IEEE Trans. on Robotics*, 24(5), 2008.

[12] 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.

[13] Joan Solà, André Monin, and Michel Devy. BiCamSLAM: Two times mono is more than stereo. In *IEEE Int. Conf. on Robotics and Automation*, pages 4795–4800, Rome, Italy, April 2007. IEEE.

[14] Joan Solà, André Monin, Michel Devy, and Thomas Lemaire. Undelayed initialization in bearing only SLAM. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2499–2504, Edmonton, Canada, 2005. IEEE.

[15] Joan Solà, André Monin, Michel Devy, and Teresa Vidal-Calleja. Fusing monocular information in multi-camera SLAM. *IEEE Trans. on Robotics*, 24(5):958–968, October 2008.

[16] Joan Solà, Teresa Vidal-Calleja, Javier Civera, and José María Martínez Montiel. Impact of landmark parametrization on monocular EKF-SLAM with points and lines. *Int. Journal of Computer Vision*, 97(3):339–368, September 2011. Available online at Springer's: http://www.springerlink.com/content/5u5176nj521kl3h0/.

[17] Joan Solà, Teresa Vidal-Calleja, and Michel Devy. Undelayed initialization of line segments in monocular SLAM. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1553–1558, Saint Louis, USA, October 2009. IEEE.