An EKF-SLAM toolbox in Matlab

Joan Solà
LAAS-CNRS

April 3, 2009

Contents

1 Quick start

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

1. Move **`slamToolbox.zip`** where you want the SLAM toolbox to be installed. Unzip it.
2. Rename the expanded directory if wanted (we'll call this directory **`SLAMtoolbox/`**).
3. Open Matlab. Add all directories and subdirectories in **`SLAMtoolbox/`** to the Matlab path.
4. Execute **`universalSlam`** from the Matlab prompt.

Or, if you want to get some more insight:

5. 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'.
6. Edit and run **`universalSlam.m`**. Explore its code by debugging step-by-step. Explore the Map figure by zooming and rotating with the mouse.
7. Read the help contents of the following 4 functions: **`frame`**, **`fromFrame`**, **`q2R`**, **`pinHole`**. Follow some of the **`See also`** links.
8. Read '**`guidelines.pdf`**' before contributing your own code.

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*.

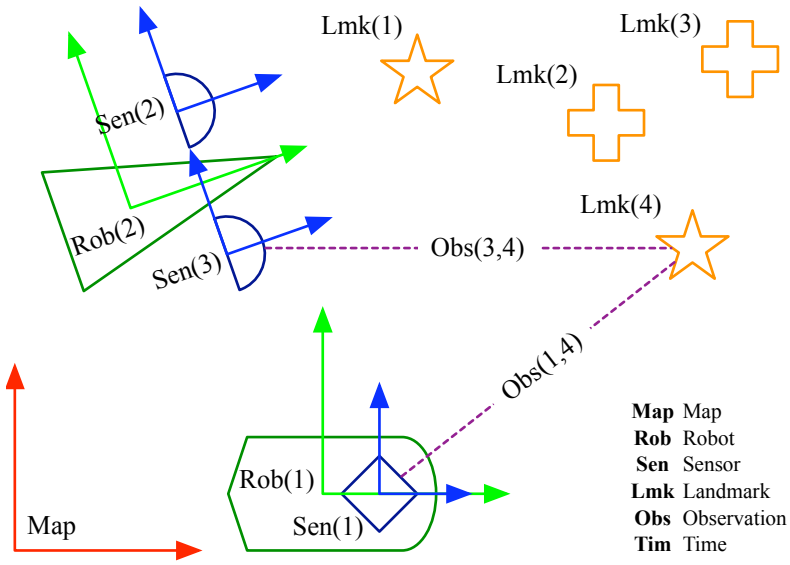

Figure 1: Overview of the SLAM problem with the principal data structures.

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`		

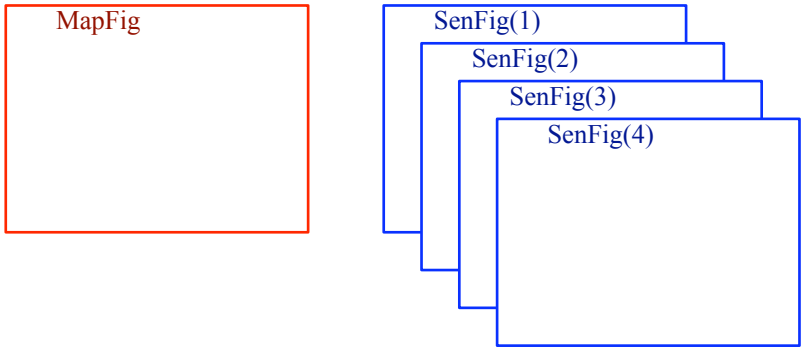

Figure 2: The set of figures. The structures **`MapFig`** and **`SenFig(s)`** contain the handles to all graphics objects drawn.

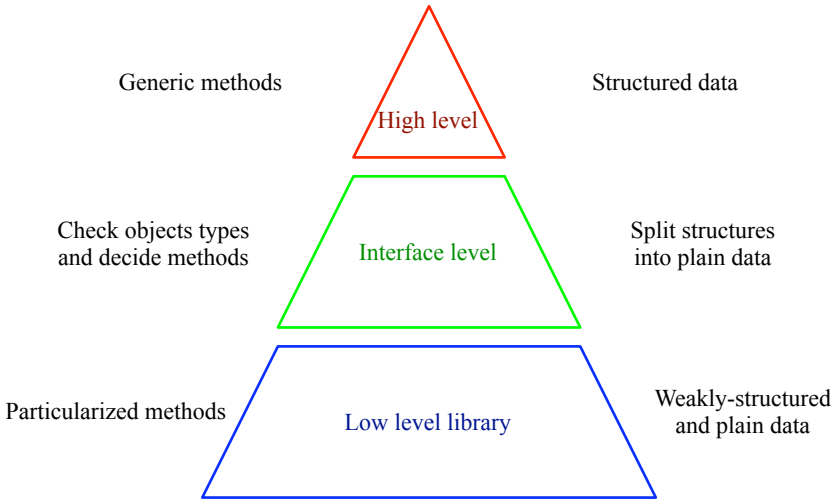

Figure 3: Overview of the levels of abstraction of the functions and their relation to data structuration. Functions and scripts in the High and Interface levels are in the **`HighLevel/`** and **`InterfaceLevel/`** directories. The Low Level library occupies all other directories.

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.

`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`** (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
        .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.

```
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
    .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.

```
Lmk(lmk)       % 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.

```
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.

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

Figure 8: The **Tim** 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
    .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_r      % wrt robot pose
        .E_s      % wrt sensor
        .E_l      % wrt landmark
```

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

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(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
        .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.

```
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
        .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.

```
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.

```
SimObs(sen)   % 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. **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.

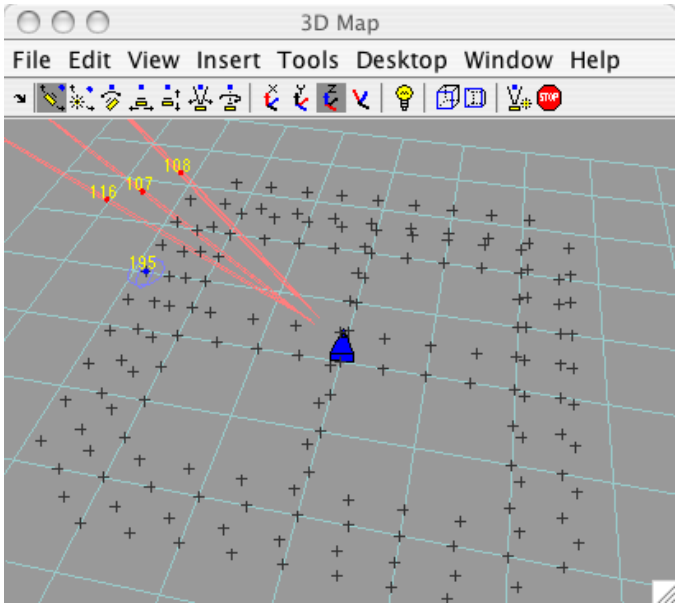

Figure 14: The 3D map figure. **MapFig** contains handles to all objects drawn.

```
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.

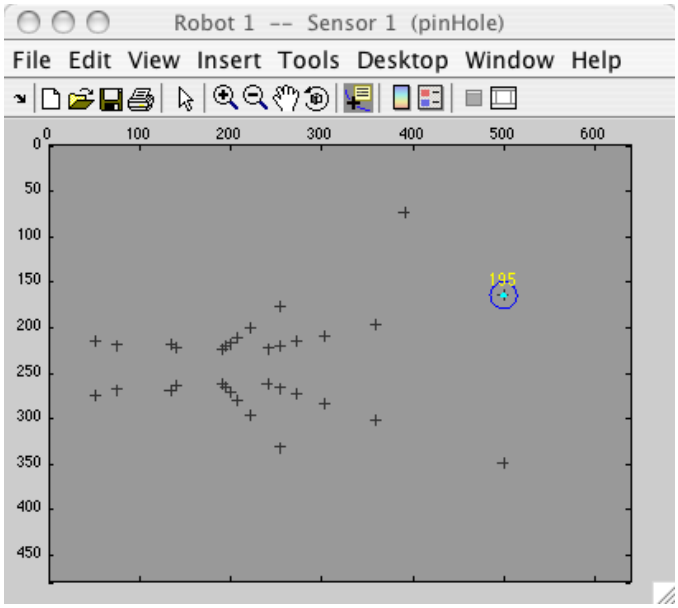

Figure 16: A pin-hole sensor view figure. **`SenFig(1)`** contains handles to all objects drawn.

```
SenFig(sen)     % Sensor figure structure, containing:
    .fig          % figure number and handle
    .axes         % axes handle
    .raw          % 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.

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.

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

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

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

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, Tim)

% SIMMOTION  Simulated robot motion.
%   Rob = SIMMOTION(Rob, Tim) performs one motion step to robot
%   Rob, following the controls in Rob.con according to the motion
%   model in Rob.motion. The time information Tim 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,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 19: 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
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.

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

1. We 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 depends on the type of sensor and cannot be generalized. We have not considered these parameters as part of the SLAM state vector.

5. The landmark has some 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, we have to provide a Gaussian prior of the non-measured part for initialization.

5.1 Direct observation model for map corrections

The observation operations are split in 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)`**). Here is a simplified implementation:

```
function e = observationModel(Rf,Sf,Sp,l)
% Rf: robot frame
% Sf: sensor frame
% Sp: sensor parameters
% l : landmark in world frame
% 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)
% PROJEUCPNTINTOPINHOLEONROB Project Euc pnt into pinhole on robot.
%    [U,S] = PROJEUCPNTINTOPINHOLEONROB(RF, SF, SPK, SPD, L)
%    projects 3D Euclidean points into a pin-hole camera mounted
%    on a robot, providing also the non-measurable depth. The input
%    parameters are:
%       RF  : robot frame
%       SF  : pin-hole sensor frame in robot
%       SPK : pin-hole intrinsic parameters [u0 v0 au av]'
%       SPD : radial distortion parameters [K2 K4 K6 ...]'
%       L   : 3D point [x y z]'
%    The output parameters are:
%       U   : 2D pixel [u v]'
%       S   : nom-measurable depth
%
%    The function accepts a points matrix L = [L1 ... Ln] as input.
%    In this case, it returns a pixels matrix U = [U1 ... Un] and
%    a depths row-vector S = [S1 ... Sn].
```

```
%
%    [U,S,U_R,U_S,U_K,U_D,U_L] = ... gives also the jacobians of
%    the observation U wrt all input parameters. Note that this
%    only works for single points.
%
%    See also PINHOLE, TOFRAME.

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)
% TOFRAME  Express in local frame a set of points from global frame.
%   TOFRAME(F,PW)  takes the W-referenced points matrix PW and
%   returns it in frame F. PW is a points matrix defined as
%   PW = [P1 P2 ... PN], where Pi = [xi;yi;zi].
%   F is a frame structure (see FRAME) containing at least:
%     t : frame position
%     q : frame orientation quaternion
%     Rt: transposed rotation matrix
%     Pc: Conjugated Pi matrix
%
%   [pf,PF_f,PF_p] = TOFRAME(...) returns the Jacobians wrt the
%   frame vector F.x and the point PW. Note that this is only
```

```
%   available for single points P = [x;y;z].
%
%   See also FRAME, FROMFRAME, Q2PI, PI2PC, QUATERNION,
%   UPDATEFRAME, SPLITFRAME.

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);   % Conjugated s
        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)
% PINHOLE Pin-hole camera model, with radial distortion.
%   U = PINHOLE(P,K) gives the projected pixel U of a point P in a
%   pin-hole camera with intrinsic parameters
%     K = [u0 v0 au av]'
%
%   The reference frames are {RDF,RD} (right-down-front for the 3D
%   world points and right-down for the pixel), according to this
%   scheme:
%
%         / z (forward)
%        /
%       +-------- x                 +-------- u
%       |                           |
%       |       3D : P=[x;y;z]      |      image : U=[u;v]
%       | y                         | v
%
%   U = PINHOLE(P,K,D) allows the introduction of the camera's
%   radial distortion parameters:
```

```
%     D = [K2 K4 K6 ...]'
%   so that the new pixel is distorted following the distortion
%   equation:
%     UD = UP * (1 + K2*R^2 + K4*R^4 + ...)
%   with R^2 = sum(UP.^2), being UP the projected point in the
%   image plane for a camera with unit focal length.
%
%   [U,S] = PINHOLE(...) returns the vector S of depths from the
%   camera center.
%
%   If P is a points matrix, PINHOLE(P,...) returns a pixel matrix
%   U and a depths row-vector S. P, U and S are defined as
%     P = [P1 ... Pn];   Pi = [xi;yi;zi]
%     U = [U1 ... Un];   Ui = [ui;vi]
%     S = [S1 ... Sn]
%
%   [U,S,U_p,U_k,U_d] = PINHOLE(...) returns the Jacobians of U
%   wrt P, K and D. It only works for single points P=[x;y;z], and
%   for distortion vectors D of up to 3 parameters D=[d2;d4;d6].
%   See DISTORT for information on longer distortion vectors.
%
%   See also PROJECT, DISTORT, PIXELLISE, INVPINHOLE, PINHOLEIDP.

% 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
```

5.2 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, e, n)`, with `Rf` the robot frame, `Sf` the sensor frame, `Sp` the sensor parameters, `e` 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, e, n)
% Rf: robot frame
% Sf: sensor frame
% Sp: sensor parameters
% e : measurement
% n : non-measured prior
% 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
```

The following code exemplifies the inverse measurement model for a pinhole camera mounted on a robot and observing 3D points, rendering 3D landmarks parametrized as inverse-depth. The inverse depth is precisely the non-measured part `n`, and is provided as a prior with a separate input. In this case, only Jacobians computation is added, as it is not likely that we need to retro-project several points at a time.[1]

```
function [idp, IDP_rf, IDP_sf, IDP_sk, IDP_sd, IDP_u, IDP_n] = ...
    retroProjIdpPntFromPinHoleOnRob(Rf, Sf, Sk, Sc, u, n)
% RETROPROJIDPPNTFROMPINHOLEONROB Retro-pr idp from pinhole on rob.
%   IDP = RETROPROJIDPPNTFROMPINHOLEONROB(RF, SF, SK, SC, U, N)
%   gives the retroprojected IDP in World Frame from an observed
%   pixel U. RF and SF are Robot and Sensor Frames, SK and SD are
%   camera calibration and distortion correction parameters. U is
%   the pixel coordinate and N is the non-observable inverse depth.
%   IDP is a 6-vector :
%     IDP = [X Y Z Pitch Yaw IDepth]'
%
%   [IDP, IDP_rf, IDP_sf, IDP_k, IDP_c, IDP_u, IDP_n] = ...
%   returns the Jacobians wrt RF.x, SF.x, SK, SC, U and N.
%
```

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

```
%   See also INVPINHOLEIDP, FROMFRAMEIDP.

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;
    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
```