Simultaneous localization and mapping with the extended Kalman filter

'A very quick guide... with Matlab code!'

Joan Solà

September 21, 2017

Contents

1 Simultaneous Localization and Mapping (SLAM)

1.1 Introduction

Simultaneous localization and mapping (SLAM) is the problem of concurrently estimating in real time the structure of the surrounding world (the map), perceived by moving exteroceptive sensors, while simultaneously getting localized in it. The seminal solution to the problem by Smith and Cheeseman (1987) [2] employs an extended Kalman filter (EKF) as the central estimator, and has been used extensively.

This file is an accompanying document for a SLAM course I give at ISAE in Toulouse every winter. Please find all the Matlab code generated during the course at the end of this document.

1.2 Notes for the absolute beginners

SLAM is a simple and everyday problem: the problem of spatial exploration. You enter an unknown space, you observe it, you move inside it; you build a spatial model of it, and you know where in this model you are located. Then, you can plan how to reach this or that part of the space, how to leave it, etc. It is truly an everyday problem. I mean, you do it all the time without even noticing it. And each time you get disoriented, confused or lost is because you did not do it right. Yet its solution, if it wants to be automated and executed by a robot, is complex and tricky, and many naive approaches to solve it literally fail.

SLAM involves a moving agent (for example a robot), which embarks at least one sensor able to gather information about its surroundings (a camera, a laser scanner, a sonar: these are called *exteroceptive sensors*). Optionally, the moving agent can incorporate other sensors to measure its own movement (wheel encoders, accelerometers, gyrometers: these are known as *proprioceptive sensors*). The minimal SLAM system consists of one moving exteroceptive sensor (for example, a camera in your hand) connected to a computer. Thus, it can be all included in your smartphone.

SLAM consists of three basic operations, which are reiterated at each time step:

The robot moves, reaching a new point of view of the scene. Due to unavoidable noise and errors, this motion **increases the uncertainty on the robot's localization**. An automated solution requires a mathematical model for this motion. We call this the *motion model*.

The robot discovers interesting features in the environment, which need to be incorporated to the map. We call these features *landmarks*. Because of errors in the exteroceptive sensors, the location of these landmarks will be uncertain. Moreover, as the robot location is already uncertain, these two uncertainties need to be properly composed. An automated solution requires a mathematical model to determine the position of the landmarks in the scene from the data obtained by the sensors. We call this the *inverse observation model*.

The robot observes landmarks that had been previously mapped, and uses them to correct both its self-localization and the localization of all landmarks in space. In this case, therefore, both **localization and landmarks uncertainties decrease**. An automated solution requires a mathematical model to predict the values of the measurement from the predicted landmark location and the robot localization. We call this the *direct observation model*.

With these three models plus an estimator engine we are able to build an automated solution to SLAM. The estimator is responsible for the proper propagation of uncertainties each time one of the three situations above occur. In the case of this course, an extended Kalman filter (EKF) is used.

Other than that, a solution to SLAM needs to chain all these operations together and to keep all data healthy and organized, making the appropriate decisions at every step.

This document covers all these aspects.

1.3 SLAM entities: map, robot, sensor, landmarks, observations, estimator...

1.3.1 Entities and their relationship

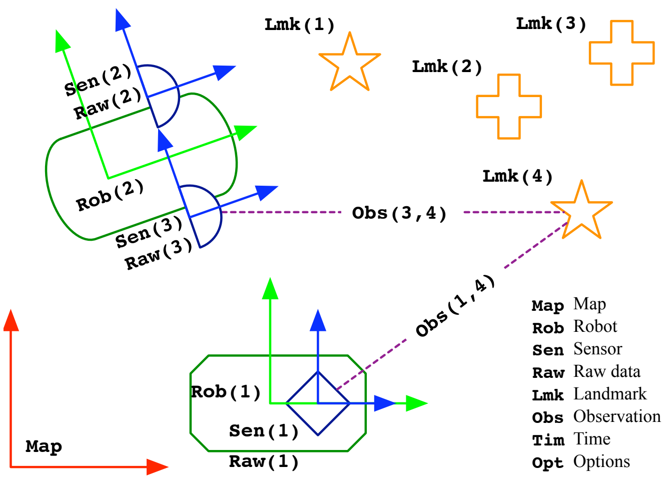


Figure 1: Typical SLAM entities

Fig. 1 is taken from the documentation of SLAMTB [3], a SLAM toolbox for Matlab that we built some years ago. In the figure we can see that

- The map has robots and landmarks.
- Robots have (exteroceptive) sensors.
- Each pair sensor-landmark defines an observation.

1.3.2 Class structure in RTSLAM

RTSLAM [1] is a C++ implementation of visual EKF-SLAM working in real-time at 60fps. Its structure of classes (Fig. 2) implements the scheme above, with the addition of two object managers. We can see that

- The map has robots and landmarks. Landmarks are maintained by *map managers* owned by the map.
- Robots have (exteroceptive) sensors.

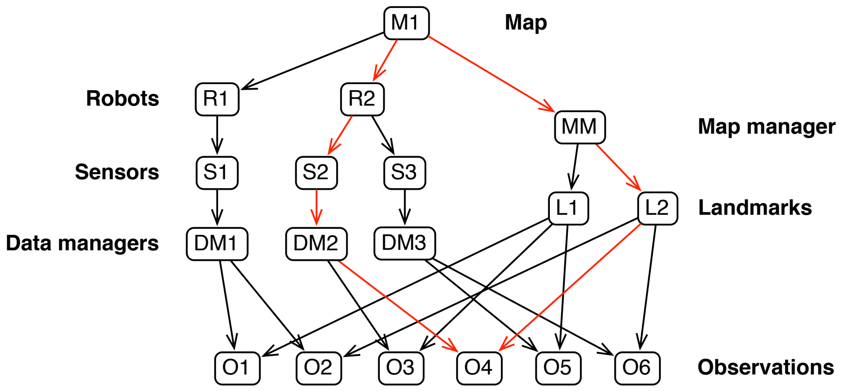


Figure 2: Ownship of classes in an object-oriented EKF-SLAM implementation

- Each pair sensor-landmark defines an observation. Observations are managed by the sensor with a *data manager*.

1.4 Motion and observation models

1.4.1 Motion model

The robot $\mathcal{R}$ moves according to a control signal $\mathbf{u}$ and a perturbation $\mathbf{n}$ and updates its state,

$$\mathcal{R} \leftarrow f(\mathcal{R}, \mathbf{u}, \mathbf{n}) \tag{1}$$

The control signal $\mathbf{u}$ is often the data from the proprioceptive sensors. It can also be the control data sent by the computer to the robot's wheels. And it can also be void, in case the motion model does not take any control input.

The perturbation $\mathbf{n}$ is considered a Gaussian random variable.

See App. A.3 for an example of motion model.

See App. C.2.1 for a Matlab implementation.

1.4.2 Direct observation model

The robot $\mathcal{R}$ observes a landmark $\mathcal{L}_i$ that was already mapped by means of one of its sensors $\mathcal{S}$. It obtains a measurement $\mathbf{y}_i$,

$$\mathbf{y}_i = h(\mathcal{R}, \mathcal{S}, \mathcal{L}_i) + \mathbf{v} \tag{2}$$

where $\mathbf{v}$ is an additive Gaussian noise that models sensor inaccuracies.

See App. A.5, Eq. (66) for an example of direct observation model.

See App. C.2.2 for a Matlab implementation.

1.4.3 Inverse observation model

The robot $\mathcal{R}$ observes a landmark $\mathcal{L}_j$ that was already mapped by means of one of its sensors $\mathcal{S}$, and obtains a measurement $\mathbf{y}_j$. With this measurement, the robot computes the state $\mathcal{L}_j$ of the newly discovered landmark,

$$\mathcal{L}_j = g(\mathcal{R}, \mathcal{S}, \mathbf{y}_j - \mathbf{v}) \tag{3}$$

where $\mathbf{v}$ is an additive Gaussian noise that models sensor inaccuracies.

See App. A.5, Eq. (67) for an example of inverse observation model.

See App. C.2.3 for a Matlab implementation.

Case of sensors offering partial observation Ideally, the function $g()$ is the inverse of $h()$ with respect to the measurement. In cases where the measurement is rank-deficient (that is, the measurement does not contain information on all the DOF of the landmark's state), we speak of *partial observation*; the observation function $h()$ is not invertible and $g()$ cannot be defined. This happens in *e.g.* monocular vision, where the images do not contain the distances to the perceived objects. The parameter $\mathbf{s}$ is then introduced as a *prior* of the lacking DOF in order to render $g()$ definible,

$$\mathcal{L}_j = g(\mathcal{R}, \mathcal{S}, \mathbf{y}_j - \mathbf{v}, \mathbf{s}) \tag{4}$$

2 EKF-SLAM

2.1 Setting up an EKF for SLAM

In EKF-SLAM, the map is a large vector stacking sensors and landmarks states, and it is modeled by a Gaussian variable. This map, usually called the stochastic map, is maintained by the EKF through the processes of prediction (the sensors move) and correction (the sensors observe the landmarks in the environment that had been previously mapped).

In order to achieve true exploration, the EKF machinery is enriched with an extra step of landmark initialization, where newly discovered landmarks are added to the map. Landmark initialization is performed by inverting the observation function and using it and its Jacobians to compute, from the sensor pose and the measurements, the observed landmark state and its necessary co- and cross-variances with the rest of the map. These relations are then appended to the state vector and the covariances matrix.

The following table resumes the similarities and differences between EKF and EKF-SLAM.

Table 1: EKF operations for achieving SLAM

Event	SLAM	EKF
Robot moves	Robot motion	EKF prediction
Sensor detects new landmark	Landmark initialization	State augmentation
Sensor observes known landmark	Map correction	EKF correction
Mapped landmark is corrupted	Landmark deletion	State reduction

2.2 The map

The map is a large state vector stacking robot and landmark states,[1]

$$\mathbf{x} = \begin{bmatrix} \mathcal{R} \\ \mathcal{M} \end{bmatrix} = \begin{bmatrix} \mathcal{R} \\ \mathcal{L}_1 \\ \vdots \\ \mathcal{L}_n \end{bmatrix} \tag{5}$$

where $\mathcal{R}$ is the robot state (typically its pose: position and orientation in space) and $\mathcal{M} = (\mathcal{L}_1, \cdots, \mathcal{L}_n)$ is the set of landmark states (typically for points, their Cartesian coordinates), with n the current number of mapped landmarks.

To illustrate the map at the time where the robot $\mathcal{R}$ has mapped two landmarks $\mathcal{L}_1$ and $\mathcal{L}_2$, the state vector is:

$$\mathcal{R} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \qquad \mathcal{L}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \qquad \mathcal{M} = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x \\ y \\ \theta \\ x_1 \\ y_1 \\ x_2 \\ y_2 \end{bmatrix} \tag{6}$$

When tens or hundreds of landmarks are mapped, the state vector becomes very large.

In EKF-SLAM, the map is modeled by a Gaussian variable using the mean and the covariances matrix of the state vector, denoted respectively by $\bar{\mathbf{x}}$ and $\mathbf{P}$,

$$\bar{\mathbf{x}} = \begin{bmatrix} \overline{\mathcal{R}} \\ \overline{\mathcal{M}} \end{bmatrix} = \begin{bmatrix} \overline{\mathcal{R}} \\ \overline{\mathcal{L}}_1 \\ \vdots \\ \overline{\mathcal{L}}_n \end{bmatrix} \qquad \mathbf{P} = \begin{bmatrix} \mathbf{P}_{\mathcal{R}\mathcal{R}} & \mathbf{P}_{\mathcal{R}\mathcal{M}} \\ \mathbf{P}_{\mathcal{M}\mathcal{R}} & \mathbf{P}_{\mathcal{M}\mathcal{M}} \end{bmatrix} = \begin{bmatrix} \mathbf{P}_{\mathcal{R}\mathcal{R}} & \mathbf{P}_{\mathcal{R}\mathcal{L}_1} & \cdots & \mathbf{P}_{\mathcal{R}\mathcal{L}_n} \\ \mathbf{P}_{\mathcal{L}_1\mathcal{R}} & \mathbf{P}_{\mathcal{L}_1\mathcal{L}_1} & \cdots & \mathbf{P}_{\mathcal{L}_1\mathcal{L}_n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{P}_{\mathcal{L}_n\mathcal{R}} & \mathbf{P}_{\mathcal{L}_n\mathcal{L}_1} & \cdots & \mathbf{P}_{\mathcal{L}_n\mathcal{L}_n} \end{bmatrix} \tag{7}$$

The goal of EKF-SLAM, therefore, is to keep the map $\{\bar{\mathbf{x}}, \mathbf{P}\}$ up to date at all times.

[1]The sensor state $\mathcal{S}$ appearing in the observation models is usually not part of the map because it is constituted of known constant parameters.

2.3 Operations of EKF-SLAM

2.3.1 Specification of noise and perturbation levels

For all motion and observation models, we need to specify their noise levels.

The perturbation noise $\mathbf{n} \sim \mathcal{N}\{0, \mathbf{N}\}$ is specified as a diagonal matrix containing the variances of each independent perturbation,

$$\mathbf{N} = \begin{bmatrix} \sigma_{n_1}^2 & & 0 \\ & \ddots & \\ 0 & & \sigma_{n_k}^2 \end{bmatrix} \tag{8}$$

The measurement noise $\mathbf{v} \sim \mathcal{N}\{0, \mathbf{R}\}$ is specified as a diagonal matrix containing the variances of each independent noise component,

$$\mathbf{R} = \begin{bmatrix} \sigma_{v_1}^2 & & 0 \\ & \ddots & \\ 0 & & \sigma_{v_m}^2 \end{bmatrix} \tag{9}$$

2.3.2 Map initialization

The map starts with no landmarks, therefore $n = 0$ and $\mathbf{x} = \mathcal{R}$. Also, the initial robot pose is usually considered the origin of the map that is going to be constructed, with absolute certainty (or absolutely no uncertainty!). Therefore,

$$\bar{\mathbf{x}} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{P} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{10}$$

2.3.3 Robot motion

In regular EKF, if $\mathbf{x}$ is our state vector, $\mathbf{u}$ is the control vector and $\mathbf{n}$ is the perturbation vector, then we have the generic time-update function

$$\mathbf{x} \leftarrow f(\mathbf{x}, \mathbf{u}, \mathbf{n}) \tag{11}$$

The EKF prediction step is classically writen as

$$\bar{\mathbf{x}} \leftarrow f(\bar{\mathbf{x}}, \mathbf{u}, 0) \tag{12}$$

$$\mathbf{P} \leftarrow \mathbf{F_x} \mathbf{P} \mathbf{F_x}^\top + \mathbf{F_n} \mathbf{N} \mathbf{F_n}^\top \tag{13}$$

with the Jacobian matrices $\mathbf{F_x} = \frac{\partial f(\bar{\mathbf{x}}, \mathbf{u})}{\partial \mathbf{x}}$ and $\mathbf{F_n} = \frac{\partial f(\bar{\mathbf{x}}, \mathbf{u})}{\partial \mathbf{n}}$, and where $\mathbf{N}$ is the covariances matrix of the perturbation $\mathbf{n}$.

In SLAM, only a part of the state is time-variant: the robot, which moves. Therefore we have a different behavior for each part of the state vector,

$$\mathcal{R} \leftarrow f_{\mathcal{R}}(\mathcal{R}, \mathbf{u}, \mathbf{n}) \tag{14}$$

$$\mathcal{M} \leftarrow \mathcal{M} \tag{15}$$

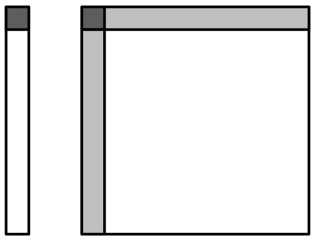


Figure 3: Updated parts of the map upon robot motion. The mean is represented by the bar on the left. The covariances matrix by the square on the right. The updated parts, in gray, correspond to the robot's state mean $\overline{\mathcal{R}}$ and covariance $\mathbf{P}_{\mathcal{R}\mathcal{R}}$ (dark gray), and the cross-variances $\mathbf{P}_{\mathcal{R}\mathcal{M}}$ and $\mathbf{P}_{\mathcal{M}\mathcal{R}}$ between the robot and the rest of the map (pale gray).

where the first equation is precisely the *motion model.* Then, because the largest part of the map is invariant upon robot motion, we have sparse Jacobian matrices with the following structure,

$$\mathbf{F}_{\mathbf{x}} = \begin{bmatrix} \frac{\partial f_{\mathcal{R}}}{\partial \mathcal{R}} & 0 \\ 0 & \mathbf{I} \end{bmatrix} \qquad \mathbf{F}_{\mathbf{n}} = \begin{bmatrix} \frac{\partial f_{\mathcal{R}}}{\partial \mathbf{n}} \\ 0 \end{bmatrix} \tag{16}$$

If we avoid performing in (13) all trivial operations such as 'multiply by one', 'multiply by zero' and 'add zero', we obtain the EKF sparse prediction equations which are used for robot motion (see Fig. 3),

$$\overline{\mathcal{R}} \leftarrow f_{\mathcal{R}}(\overline{\mathcal{R}}, \mathbf{u}, 0) \tag{17}$$

$$\mathbf{P}_{\mathcal{R}\mathcal{R}} \leftarrow \frac{\partial f_{\mathcal{R}}}{\partial \mathcal{R}} \mathbf{P}_{\mathcal{R}\mathcal{R}} \frac{\partial f_{\mathcal{R}}}{\partial \mathcal{R}}^{\top} + \frac{\partial f_{\mathcal{R}}}{\partial \mathbf{n}} \mathbf{N} \frac{\partial f_{\mathcal{R}}}{\partial \mathbf{n}}^{\top} \tag{18}$$

$$\mathbf{P}_{\mathcal{R}\mathcal{M}} \leftarrow \frac{\partial f_{\mathcal{R}}}{\partial \mathcal{R}} \mathbf{P}_{\mathcal{R}\mathcal{M}} \tag{19}$$

$$\mathbf{P}_{\mathcal{M}\mathcal{R}} \leftarrow \mathbf{P}_{\mathcal{R}\mathcal{M}}^{\top} \tag{20}$$

Moreover, if we take care to store the covariances matrix as a triangular matrix, which is possible because it is symmetric, the operation (20) does not need to be performed.

The algorithmic complexity of this set of equations is $O(n)$ due to (19).

We leave as an exercise to proof (18–20) from (13) and (16).

2.3.4 Observation of mapped landmarks

Similarly, we have in EKF the generic observation function

$$\mathbf{y} = h(\mathbf{x}) + \mathbf{v} \tag{21}$$

where $\mathbf{y}$ is the noisy measurement, $\mathbf{x}$ is the full state, $h()$ is the observation function and $\mathbf{v}$ is the measurement noise.

The EKF correction step is classically written as

$$\bar{\mathbf{z}} = \mathbf{y} - h(\bar{\mathbf{x}}) \tag{22}$$

$$\mathbf{Z} = \mathbf{H_x} \mathbf{P} \mathbf{H_x}^\top + \mathbf{R} \tag{23}$$

$$\mathbf{K} = \mathbf{P} \mathbf{H_x}^\top \mathbf{Z}^{-1} \tag{24}$$

$$\bar{\mathbf{x}} \leftarrow \bar{\mathbf{x}} + \mathbf{K}\bar{\mathbf{z}} \tag{25}$$

$$\mathbf{P} \leftarrow \mathbf{P} - \mathbf{K}\mathbf{Z}\mathbf{K}^\top \tag{26}$$

with the Jacobian $\mathbf{H_x} = \frac{\partial h(\bar{\mathbf{x}})}{\partial \mathbf{x}}$ and where $\mathbf{R}$ is the covariances matrix of the measurement noise. In these equations, the first two are the innovation's mean and covariances matrix $\{\bar{\mathbf{z}}; \mathbf{Z}\}$; the third one is the Kalman gain $\mathbf{K}$; and the last two constitute the filter update.

In SLAM, observations occur when a measure of a particular landmark is taken by any of the robot's embarked sensors. This usually requires a more or less significant amount of raw data processing (especially in vision and other high-bandwidth sensors), which is obviated by now. The outcome of this processing is a geometric parametrization of the landmark in the measurement space: the vector $\mathbf{y}_i$.

For example, in vision, a measurement of a point landmark corresponds to the coordinates of the pixel where this landmark is projected in the image: $\mathbf{y}_i = (u_i, v_i)$.

Landmark observations are processed in the EKF usually one-by-one. The observation depends only on the robot position or state $\mathcal{R}$, the sensor state $\mathcal{S}$ and the particular landmark's state $\mathcal{L}_i$. Assuming that landmark i is observed, we have the individual observation function (the *observation model*)

$$\mathbf{y}_i = h_i(\mathcal{R}, \mathcal{S}, \mathcal{L}_i) + \mathbf{v} \tag{27}$$

which does not depend on any other landmark than $\mathcal{L}_i$. Therefore the structure of the Jacobian $\mathbf{H_x}$ in EKF-SLAM is also sparse,

$$\mathbf{H_x} = \begin{bmatrix} \mathbf{H}_\mathcal{R} & 0 & \cdots & 0 & \mathbf{H}_{\mathcal{L}_i} & 0 & \cdots & 0 \end{bmatrix} \tag{28}$$

with $\mathbf{H}_\mathcal{R} = \frac{\partial h_i(\overline{\mathcal{R}}, \mathcal{S}, \overline{\mathcal{L}}_i)}{\partial \mathcal{R}}$ and $\mathbf{H}_{\mathcal{L}_i} = \frac{\partial h_i(\overline{\mathcal{R}}, \mathcal{S}, \overline{\mathcal{L}}_i)}{\partial \mathcal{L}_i}$. Thanks to this sparsity the equations (23) and (24) can be reduced to only the products involving the non-zero elements (see Fig. 4), and the set of correction equations becomes

$$\bar{\mathbf{z}} = \mathbf{y}_i - h_i(\overline{\mathcal{R}}, \mathcal{S}, \overline{\mathcal{L}}_i) \tag{29}$$

$$\mathbf{Z} = \begin{bmatrix} \mathbf{H}_\mathcal{R} & \mathbf{H}_{\mathcal{L}_i} \end{bmatrix} \begin{bmatrix} \mathbf{P}_{\mathcal{R}\mathcal{R}} & \mathbf{P}_{\mathcal{R}\mathcal{L}_i} \\ \mathbf{P}_{\mathcal{L}_i\mathcal{R}} & \mathbf{P}_{\mathcal{L}_i\mathcal{L}_i} \end{bmatrix} \begin{bmatrix} \mathbf{H}_\mathcal{R}^\top \\ \mathbf{H}_{\mathcal{L}_i}^\top \end{bmatrix} + \mathbf{R} \tag{30}$$

$$\mathbf{K} = \begin{bmatrix} \mathbf{P}_{\mathcal{R}\mathcal{R}} & \mathbf{P}_{\mathcal{R}\mathcal{L}_i} \\ \mathbf{P}_{\mathcal{M}\mathcal{R}} & \mathbf{P}_{\mathcal{M}\mathcal{L}_i} \end{bmatrix} \begin{bmatrix} \mathbf{H}_\mathcal{R}^\top \\ \mathbf{H}_{\mathcal{L}_i}^\top \end{bmatrix} \mathbf{Z}^{-1} \tag{31}$$

$$\bar{\mathbf{x}} \leftarrow \bar{\mathbf{x}} + \mathbf{K}\bar{\mathbf{z}} \tag{32}$$

$$\mathbf{P} \leftarrow \mathbf{P} - \mathbf{K}\mathbf{Z}\mathbf{K}^\top \tag{33}$$

The complexity of this set of equations is $O(n^2)$ due to (33). Such set of equations is applied each time a landmark is measured and updated. The total complexity for a

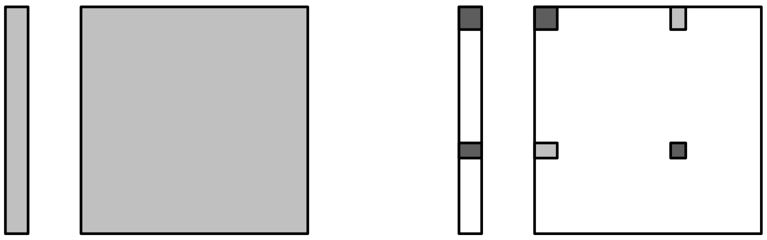


Figure 4: *Left:* Updated parts of the map upon landmark observation. The updated parts (in gray) correspond to the full map because the Kalman gain matrix $\mathbf{K}$ affects the full state. *Right:* However, the computation of the innovation (29, 30) is sparse: it only involves (in dark gray) the robot state $\overline{\mathcal{R}}$, the concerned landmark state $\overline{\mathcal{L}}_i$ and their covariances $\mathbf{P}_{\mathcal{R}\mathcal{R}}$ and $\mathbf{P}_{\mathcal{L}_i\mathcal{L}_i}$, and (in pale gray) their cross-variances $\mathbf{P}_{\mathcal{R}\mathcal{L}_i}$ and $\mathbf{P}_{\mathcal{L}_i\mathcal{R}}$.

total of k landmark updates is therefore $O(kn^2)$. It is worth noticing, for those who are used to standard EKF, that in our case the inversion of the innovation matrix $\mathbf{Z}$ is done in constant time $\mathcal{O}(1)$ (as opposed to cubic time $\mathcal{O}(n^3)$ in EKF!). Then, the Kalman gain $\mathbf{K}$ is computed in linear time $\mathcal{O}(n)$.

We leave as an exercise to proof (30–31) from (23–24) and (28).

2.3.5 Landmark initialization for full observations

Landmark initialization happens when the robot discovers landmarks that are not yet mapped and decides to incorporate them in the map. As such, this operation results in an increase of the state vector's size. The EKF becomes then a filter of a state of dynamic size. This is why this operation is not usually known by users of regular EKF.

Landmark initialization is simple in cases where the sensor provides information about all the degrees of freedom of the new landmark. When this happens, we only need to invert the observation function $h()$ to compute the new landmark's state $\mathcal{L}_{n+1}$ from the robot state $\mathcal{R}$, the sensor state $\mathcal{S}$ and the observation $\mathbf{y}_{n+1}$,

$$\mathcal{L}_{n+1} = g(\mathcal{R}, \mathcal{S}, \mathbf{y}_{n+1}), \tag{34}$$

which constitutes the *inverse observation model* of one landmark.

We proceed as follows. First compute the landmark's mean and the function's Jacobians[2]

$$\overline{\mathcal{L}}_{n+1} = g(\overline{\mathcal{R}}, \mathcal{S}, \mathbf{y}_{n+1}) \tag{35}$$

$$\mathbf{G}_{\mathcal{R}} = \frac{\partial g(\overline{\mathcal{R}}, \mathcal{S}, \mathbf{y}_{n+1})}{\partial \mathcal{R}} \tag{36}$$

$$\mathbf{G}_{\mathbf{y}_{n+1}} = \frac{\partial g(\overline{\mathcal{R}}, \mathcal{S}, \mathbf{y}_{n+1})}{\partial \mathbf{y}_{n+1}} \tag{37}$$

[2]See App. C.2.3 for a Matlab implementation.

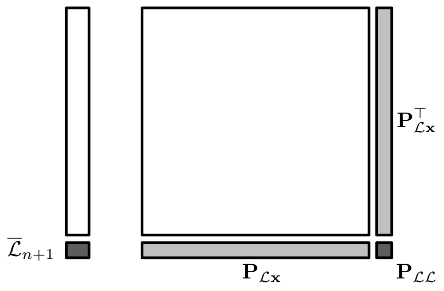

$\mathbf{P}_{\mathcal{L}\mathbf{x}}^\top$
$\bar{\mathcal{L}}_{n+1}$
$\mathbf{P}_{\mathcal{L}\mathbf{x}}$
$\mathbf{P}_{\mathcal{L}\mathcal{L}}$

Figure 5: Appended parts to the map upon landmark initialization. The appended parts, in gray, correspond to the landmark's mean and covariance (dark gray), and the cross-variances between the landmark and the rest of the map (pale gray). Each gray block in the figure is identified with the results of equations (35) and (38–39), and the update is as shown in (40–41).

Then compute the landmark's co-variance $\mathbf{P}_{\mathcal{L}\mathcal{L}}$, and its cross-variance with the rest of the map $\mathbf{P}_{\mathcal{L}\mathbf{x}}$,

$$\mathbf{P}_{\mathcal{L}\mathcal{L}} = \mathbf{G}_{\mathcal{R}}\mathbf{P}_{\mathcal{R}\mathcal{R}}\mathbf{G}_{\mathcal{R}}^\top + \mathbf{G}_{\mathbf{y}_{n+1}}\mathbf{R}\mathbf{G}_{\mathbf{y}_{n+1}}^\top \tag{38}$$

$$\mathbf{P}_{\mathcal{L}\mathbf{x}} = \mathbf{G}_{\mathcal{R}}\mathbf{P}_{\mathcal{R}\mathbf{x}} = \mathbf{G}_{\mathcal{R}}\begin{bmatrix}\mathbf{P}_{\mathcal{R}\mathcal{R}} & \mathbf{P}_{\mathcal{R}\mathcal{M}}\end{bmatrix} \tag{39}$$

Finally append these results to the state mean and covariances matrix (see Fig. 5)

$$\bar{\mathbf{x}} \leftarrow \begin{bmatrix}\bar{\mathbf{x}} \\ \bar{\mathcal{L}}_{n+1}\end{bmatrix} \tag{40}$$

$$\mathbf{P} \leftarrow \begin{bmatrix}\mathbf{P} & \mathbf{P}_{\mathcal{L}\mathbf{x}}^\top \\ \mathbf{P}_{\mathcal{L}\mathbf{x}} & \mathbf{P}_{\mathcal{L}\mathcal{L}}\end{bmatrix} \tag{41}$$

The complexity of this set of operations is $O(n)$.

2.3.6 Landmark initialization for partial observations

In cases where the sensor does not provide enough degrees of freedom for the function $h()$ to be invertible, we need to introduce this lacking information as a prior to the system [4]. This is the case when using bearing only sensors such as a monocular camera or range-only sensors such as a sonar.

With a prior, the inverse observation model is augmented to

$$\mathcal{L}_{n+1} = g(\mathcal{R}, \mathcal{S}, \mathbf{y}_{n+1}, \mathbf{s}) \tag{42}$$

where $\mathbf{s}$ is the prior. This prior is Gaussian with mean $\bar{\mathbf{s}}$ and covariances matrix $\mathbf{S}$.

From this on, the way to proceed is analogous to the fully observable case with just the addition of the prior term. First compute landmark's mean and all Jacobians

$$\overline{\mathcal{L}}_{n+1} = g(\overline{\mathcal{R}}, \mathcal{S}, \mathbf{y}_{n+1}, \bar{\mathbf{s}}) \tag{43}$$

$$\mathbf{G}_{\mathcal{R}} = \frac{\partial g(\overline{\mathcal{R}}, \mathcal{S}, \mathbf{y}_{n+1}, \bar{\mathbf{s}})}{\partial \mathcal{R}} \tag{44}$$

$$\mathbf{G}_{\mathbf{y}_{n+1}} = \frac{\partial g(\overline{\mathcal{R}}, \mathcal{S}, \mathbf{y}_{n+1}, \bar{\mathbf{s}})}{\partial \mathbf{y}_{n+1}} \tag{45}$$

$$\mathbf{G}_{\mathbf{s}} = \frac{\partial g(\overline{\mathcal{R}}, \mathcal{S}, \mathbf{y}_{n+1}, \bar{\mathbf{s}})}{\partial \mathbf{s}} \tag{46}$$

Then compute the landmark's co-variance and the cross-variance with the rest of the map

$$\mathbf{P}_{\mathcal{L}_{n+1}\mathcal{L}_{n+1}} = \mathbf{G}_{\mathcal{R}}\mathbf{P}_{\mathcal{R}\mathcal{R}}\mathbf{G}_{\mathcal{R}}^\top + \mathbf{G}_{\mathbf{y}_{n+1}}\mathbf{R}\mathbf{G}_{\mathbf{y}_{n+1}}^\top + \mathbf{G}_{\mathbf{s}}\mathbf{S}\mathbf{G}_{\mathbf{s}}^\top \tag{47}$$

$$\mathbf{P}_{\mathcal{L}\mathbf{x}} = \mathbf{G}_{\mathcal{R}}\mathbf{P}_{\mathcal{R}\mathbf{x}} = \mathbf{G}_{\mathcal{R}}\begin{bmatrix}\mathbf{P}_{\mathcal{R}\mathcal{R}} & \mathbf{P}_{\mathcal{R}\mathcal{M}}\end{bmatrix} \tag{48}$$

And finally augment the map as before.

Important note on Partial Landmark Initialization This trick of just introducing an invented prior seems trivial but it is not. Being $\mathbf{s}$ an unknown parameter, its associated covariance must ideally be infinite. And then this is what happens if we do not take important precautions:

1. EKF expects reasonable linearizations. This means that the function Jacobians must be valid approximations of the function derivatives inside all the probability concentration region (PCR) of the state variable.
2. If one DOF of our state has infinite uncertainty, it is required by the above rule that the functions manipulating it have a fairly linear behavior along the whole unbounded PCR of the prior. This is usually not the case.
3. As a consequence, setting up a naive system with a naive prior will most probably break the linearity condition and make EKF fail.

2.3.7 Partial landmark initialization from bearing-only measurements

The first key for a proper EKF performance is linearity. Linearity is defined as the opposite to non-linearity. Trivial. And non-linearity is defined as the change in the function derivatives inside the probability concentration region (PCR) of the input variables. Therefore non-linearity depends on both the function and the PCR (Fig. 6).

Then, if one of the input variables is completely unknown, its PCR is unbounded (it reaches the infinity) and the non-linearity should be small over this unbounded PCR.

In a bearing-only sensor such as a video camera, the unmeasured distance has an unbounded PCR, which reaches the infinity.

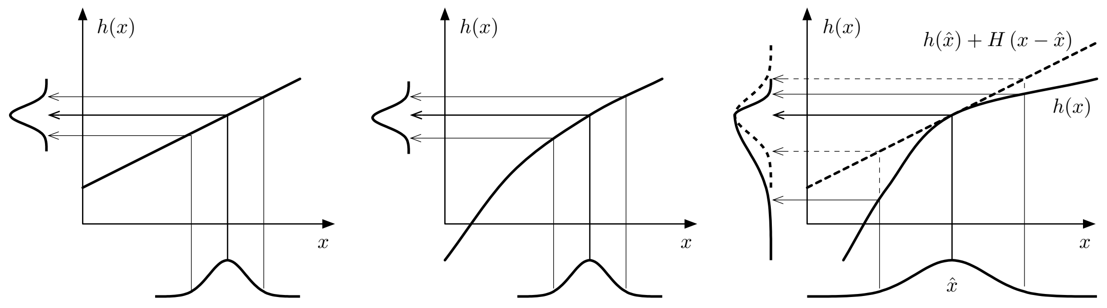


Figure 6: Linearization quality as a function of the probability concentration region (PCR). *Left*: linear case. *Center*: good linearization: the function is reasonably linear inside the PCR. *Right*: bad linearization: the derivatives vary very much within the PCR.

Because the observation function is nonlinear with respect to distance, we cannot assure a proper linearization inside the unbounded PCR. Can we do something about it? The answer is YES.

The first thing we can do is to define a new variable ρ as the inverse of the distance d

$$\rho \triangleq 1/d \tag{49}$$

and define the prior in this new variable. Assuming that the PCR of d spans from a certain minimum distance d_{min} to infinity, the PCR of ρ becomes bounded

$$d \in [d_{min}\ ,\ \infty] \quad \to \quad \rho \in [0\ ,\ 1/d_{min}] \tag{50}$$

We can define the (Cartesian) landmark position as a function of this new parameter, as follows

$$\mathbf{p} = \mathbf{p}_0 + 1/\rho \begin{bmatrix} \cos(\alpha) \\ \sin(\alpha) \end{bmatrix} \tag{51}$$

where $\mathbf{p}_0$ is the position of the sensor at the time of initialization, and α is an angle representing a direction in space. It is then convenient to parametrize the landmark as follows

$$\mathcal{L} = \begin{bmatrix} \mathbf{p}_0 \\ \alpha \\ \rho \end{bmatrix} \in \mathbb{R}^4 \tag{52}$$

The power of such construction becomes visible when we express the bearing of a new measurement taken from another robot position $\mathbf{t}$. Consider the vector $\mathbf{v} = \mathbf{p} - \mathbf{t}$ corresponding to the new line of sight,

$$\mathbf{v} \triangleq \mathbf{p} - \mathbf{t} = (\mathbf{p}_0 - \mathbf{t}) + 1/\rho \begin{bmatrix} \cos(\alpha) \\ \sin(\alpha) \end{bmatrix}. \tag{53}$$

Because the sensor cannot observe distances, the measurement of this landmark is insensitive to the magnitude of (or norm of) $\mathbf{v}$. Therefore rescaling the expression above by a factor ρ yields

$$\mathbf{v} \propto \rho(\mathbf{p}_0 - \mathbf{t}) + \begin{bmatrix} \cos(\alpha) \\ \sin(\alpha) \end{bmatrix}, \tag{54}$$

which is **linear in** ρ.

> KEY FACT: *The observation function in homogeneous coordinates is linear in ρ, and the PCR is bounded in ρ. This means that EKF is probably going to work!*

The direct observation function is the bearing of $\mathbf{v}$,

$$\mathbf{y} = h(\mathcal{R}, \mathcal{L}) = \arctan(v_2/v_1) - \theta \tag{55}$$

The inverse observation function for this landmark parametrization, knowing the robot state $\mathcal{R} = [\mathbf{t}, \theta]^\top$ at initialization time and the bearing-only measurement $\mathbf{y} = \phi$, is simply

$$\mathcal{L} = g(\mathcal{R}, \mathbf{y}, \rho) = \begin{bmatrix} \mathbf{t} \\ \theta + \phi \\ \rho \end{bmatrix} \tag{56}$$

2.4 Chaining the events

A basic but functioning algorithm performing SLAM needs to chain all these operations in a meaningful way. The following pseudocode is a valuable example,

```
% INITIALIZATION
initialize_map()
time = 0

% TIME LOOP
while (execution() == true) do

    % MOVE ROBOT
    control = acquire_control_signal()
    move_robot(robot, control)

    % ACQUIRE RAW SENSOR DATA
    raw = sensor->acquire_raw_data()

    % LOOP EXISTING LANDMARKS
    for each landmark in sensor->visible_landmarks()

        % MEASURE LANDMARK AND CORRECT MAP
        measurement = find_known_feature(raw)
        update_map(robot, sensor, landmark, measurement)
    end
    % DISCOVER NEW LANDMARKS
    measurement = detect_new_feature(raw)

    % INITIALIZE LANDMARK
    landmark = init_new_landmark(robot, sensor, measurement)

    time ++
end
```

This course is devoted to expand this pseudo code to a full functional algorithm that implements a 2-dimensional SLAM system. The full code is collected in App. C.

Appendices

A Geometry

A.1 Rotation matrix

$$\mathsf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{57}$$

A.2 Reference frames

Let $\mathcal{W}$ be the World frame. Let $\mathcal{F}$ be a cartesian frame defined with respect to the world frame by a translation vector $\mathbf{t}$ and a rotation angle θ.

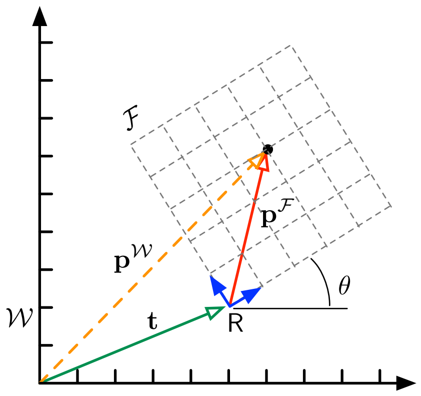


Figure 7: Frame transformation in the 2D plane. The two blue arrows are the two column vectors of the rotation matrix R, corresponding to the orientation θ of the local frame $\mathcal{F}$.

A point in space $\mathbf{p}$ can be expressed in World frame or in the local frame $\mathcal{F}$. Both expressions are related by the frame-transformation equations,

$$\mathbf{p}^{\mathcal{W}} = \mathsf{R}\mathbf{p}^{\mathcal{F}} + \mathbf{t} \quad \leftarrow \text{from frame } \mathcal{F} \tag{58}$$

$$\mathbf{p}^{\mathcal{F}} = \mathsf{R}^{\top}(\mathbf{p}^{\mathcal{W}} - \mathbf{t}) \leftarrow \text{to frame } \mathcal{F} \tag{59}$$

where R is the rotation matrix associated with the angle θ. The first expression is known as the '*from frame*' transformation. The second one is known as '*to frame*'.

A.3 Motion of a body in the plane

Let a Robot move in the plane. Let its state be the position and orientation, defined by

$$\mathcal{R} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} \mathbf{t} \\ \theta \end{bmatrix} \tag{60}$$

Let this robot receive a control signal $\mathbf{u}$ in the form of a vector specifying linear and angular pose increments,

$$\mathbf{u} = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{bmatrix} = \begin{bmatrix} \Delta\mathbf{t} \\ \Delta\theta \end{bmatrix} \tag{61}$$

After the motion step, the robot state $\mathcal{R}$ is updated according to

$$\mathbf{t} \leftarrow \mathsf{R}\Delta\mathbf{t} + \mathbf{t} \tag{62}$$

$$\theta \leftarrow \theta + \Delta\theta \tag{63}$$

where the first equation corresponds to a '*from frame*' transform, while the second one is trivial (well, one needs to take care to keep the angles in $\theta \in [-\pi, \pi]$ after each update).

A.4 Polar coordinates

Let a point in the plane be expressed by its two cartesian coordinates, $\mathbf{p} = [x, y]^\top$. Its polar representation is

$$\widehat{\mathbf{p}} = \begin{bmatrix} \rho \\ \phi \end{bmatrix} = \text{polar}(\mathbf{p}) = \begin{bmatrix} \sqrt{x^2 + y^2} \\ \arctan(y, x) \end{bmatrix}. \tag{64}$$

This operation can be inverted as follows

$$\mathbf{p} = \begin{bmatrix} x \\ y \end{bmatrix} = \text{rectangular}(\widehat{\mathbf{p}}) = \begin{bmatrix} \rho\cos\phi \\ \rho\sin\phi \end{bmatrix}. \tag{65}$$

A.5 Useful combinations

Suitable combinations of frame transforms and polar transforms are very handy. Many onboard sensors used in mobile robotics such as laser rangers, sonars, video cameras, etc., provide information of the external landmarks in the form of range and/or bearing with respect to the local sensor frame. Range-only sensors perform only the first row of (64). Bearing-only sensors perform only the second row. Range-and-bearing sensors perform both rows.

Let the point $\mathbf{p}$ above be expressed in World frame. The polar representation of this point in frame $\mathcal{F}$ is obtained by composing a '*to frame*' transformation with the polar transform above,

$$\widehat{\mathbf{p}}^{\mathcal{F}} = \text{polar}(\text{toFrame}(\mathcal{F}, \mathbf{p}^{\mathcal{W}})). \tag{66}$$

The opposite situation requires composing the inverse functions in reversed order,

$$\mathbf{p}^{\mathcal{W}} = \text{fromFrame}(\mathcal{F}, \text{rectangular}(\widehat{\mathbf{p}}^{\mathcal{R}})) \tag{67}$$

Equations (66) and (67) are used as the direct and inverse range-and-bearing observation functions in SLAM. We can call **`observe()`** the first function, in which the robot is obtaining a range-and-bearing measurement of a point, and **`invObserve()`** the second one, with the operation of obtaining a point from a range-and-bearing measurement. See Appendices C.2.2 and C.2.3 for their Matlab implementation.

Table 2: Range and/or bearing information provided by popular sensors

sensor	range	bearing
Laser range finder	YES	YES
Sonar	YES	poor
Camera	NO	YES
RGBD (*e.g.* Kinect)	YES	YES
ARVA	poor	poor
RFID antenna	poor	poor

B Probability

B.1 Generalities

B.1.1 Probability density function

$$p_X(\mathbf{x}) \triangleq \lim_{d\mathbf{x}\to 0} \frac{P(\mathbf{x} \leq X < \mathbf{x} + d\mathbf{x})}{d\mathbf{x}} \tag{68}$$

B.1.2 Expectation operator

$$\mathbb{E}[f(\mathbf{x})] \triangleq \int_{-\infty}^{\infty} f(\mathbf{x})p(\mathbf{x})d\mathbf{x} \tag{69}$$

B.1.3 Very useful examples

Mean and covariances matrix

$$\overline{\mathbf{x}} \triangleq \mathbb{E}[\mathbf{x}] \tag{70}$$
$$\mathbf{P} \triangleq \mathbb{E}[(\mathbf{x}-\overline{\mathbf{x}})(\mathbf{x}-\overline{\mathbf{x}})^\top] \tag{71}$$

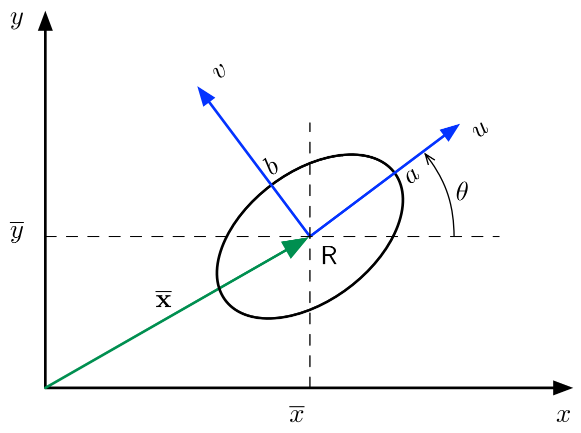
B.2 Gaussian variables

B.2.1 Introduction and definitions

$$\mathcal{N}(\mathbf{x},\overline{\mathbf{x}},\mathbf{P}) = \frac{1}{\sqrt{(2\pi)^n|\mathbf{P}|}} \exp\Big(-\frac{1}{2}(\mathbf{x}-\overline{\mathbf{x}})^\top\mathbf{P}^{-1}(\mathbf{x}-\overline{\mathbf{x}})\Big) \tag{72}$$
$$\tag{73}$$

B.2.2 Linear propagation

$$\mathbf{y} = \mathbf{F}\mathbf{x} \tag{74}$$
$$\overline{\mathbf{y}} = \mathbf{F}\overline{\mathbf{x}} \tag{75}$$
$$\mathbf{Y} = \mathbf{F}\mathbf{P}\mathbf{F}^\top \tag{76}$$



$$(\mathbf{x}-\overline{\mathbf{x}})^\top \mathsf{R} \begin{bmatrix} 1/a^2 & 0 \\ 0 & 1/b^2 \end{bmatrix} \mathsf{R}^\top (\mathbf{x}-\overline{\mathbf{x}}) = 1$$

Figure 8: Ellipsoidal representation of multivariate Gaussian variables (2D). Ellipse dimensions, position and orientation are governed by the SVD decomposition of the covariances matrix $\mathbf{P}$.

B.2.3 Nonlinear propagation and linear approximation

We make use of the first-order Taylor approximation of the function, with $\mathbf{x}_0 = \overline{\mathbf{x}}$ as the linearization point,

$$f(\mathbf{x}) = f(\overline{\mathbf{x}}) + \mathbf{F}_\mathbf{x}(\mathbf{x}-\overline{\mathbf{x}}) + O(\|\mathbf{x}-\overline{\mathbf{x}}\|^2) \tag{77}$$

with $\mathbf{F}_\mathbf{x} = \frac{\partial f(\overline{\mathbf{x}})}{\partial \mathbf{x}}$ the Jacobian of $f(\mathbf{x})$ with respect to $\mathbf{x}$ around $\overline{\mathbf{x}}$. Then,

$$\mathbf{y} = f(\mathbf{x}) \tag{78}$$

$$\overline{\mathbf{y}} \approx f(\overline{\mathbf{x}}) \tag{79}$$

$$\mathbf{Y} \approx \mathbf{F}_\mathbf{x}\mathbf{P}\mathbf{F}_\mathbf{x}^\top \tag{80}$$

B.3 Graphical representation of Gaussian uncertainties

From the multivariate Gaussian definition, we have that the part depending on $\mathbf{x}$ is only the exponent (at the right of the exp!). A curve of constant probability density can therefore be found as the locus of points $\mathbf{x}$ satisfying

$$(\mathbf{x}-\overline{\mathbf{x}})^\top \mathbf{P}^{-1}(\mathbf{x}-\overline{\mathbf{x}}) = const \tag{81}$$

When $const = 1$, this corresponds (see Fig. 8) to an ellipsoid centered at $\mathbf{x} = \overline{\mathbf{x}}$, with semiaxes oriented as the eigenvectors of $\mathbf{P}$ and of equal length as the square root of the singular values of $\mathbf{P}$. [3]

[3] **Proof:** Consider the ellipse in axes (u, v) in Fig. 8, with semiaxes a and b. Express it as $\frac{u^2}{a^2} + \frac{v^2}{b^2} = 1$. Write this expression in matrix form as

$$\mathbf{v}^\top \mathbf{D}^{-1}\mathbf{v} = 1 \tag{82}$$

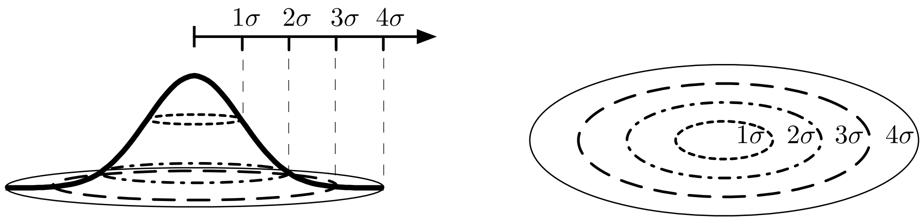


Figure 9: Ellipsoidal representation of multivariate Gaussian variables (2D). Different sigma-value ellipses can be defined for the same covariances matrix. The most useful ones are 2-sigma and 3-sigma.

Table 3: Percent probabilities of a random variable being inside its n-sigma ellipsoid.

	1σ	2σ	3σ	4σ
2D	$39,4\%$	$86,5\%$	$98,9\%$	$99,97\%$
3D	$19,9\%$	$73,9\%$	$97,1\%$	$99,89\%$

B.3.1 The Mahalanobis distance and the n-sigma ellipsoid

We can define the Mahalanobis distance as the normalized quadratic distance operator

$$\|\mathbf{x}-\overline{\mathbf{x}}\|_{\mathbf{P}} \triangleq \sqrt{(\mathbf{x}-\overline{\mathbf{x}})^\top \mathbf{P}^{-1}(\mathbf{x}-\overline{\mathbf{x}})} \tag{86}$$

With this, the ellipsoid above is the locus of points at a Mahalanobis distance of 1 from the point $\overline{\mathbf{x}}$.

We can also draw the ellipsoids at Mahalanobis distances other than one. These are called the n-sigma ellipsoids and are defined as a function of n by the locus $\|\mathbf{x}-\overline{\mathbf{x}}\|_{\mathbf{P}} = n$, or more explicitly

$$(\mathbf{x}-\overline{\mathbf{x}})^\top \mathbf{P}^{-1}(\mathbf{x}-\overline{\mathbf{x}}) = n^2 \tag{87}$$

In SLAM we make extensive use of the 2- and 3-sigma ellipsoids because they enclose probability concentrations of 97% to 99% (see Fig. 9 and Table 3).

with $\mathbf{v} = [u\ v]^\top$ and $\mathbf{D} = \mathrm{diag}(a^2, b^2)$. Then define the local reference frame $\mathbf{v}$ with respect to the global frame $\mathbf{x}$ by means of a rotation matrix R and a translation vector $\overline{\mathbf{x}}$. The following *to-frame* relation holds,

$$\mathbf{v} = \mathsf{R}^\top(\mathbf{x}-\overline{\mathbf{x}}). \tag{83}$$

Now inserting (83) into (82),

$$(\mathbf{x}-\overline{\mathbf{x}})^\top \mathsf{R}\mathbf{D}^{-1}\mathsf{R}^\top(\mathbf{x}-\overline{\mathbf{x}}) = 1 \tag{84}$$

and identifying terms in (81) we have that

$$\mathbf{P} = (\mathsf{R}\mathbf{D}^{-1}\mathsf{R}^\top)^{-1} = \mathsf{R}^{-\top}\mathbf{D}\mathsf{R}^{-1} = \mathsf{R}\mathbf{D}\mathsf{R}^\top \tag{85}$$

which corresponds to the singular value decomposition of $\mathbf{P}$.

Drawing the n-sigma ellipsoid is easy. Start by performing the SVD of $\mathbf{P}$,

$$[\mathsf{R}, \mathbf{D}] = \mathrm{svd}(\mathbf{P}) \tag{88}$$

Also, build the 2-by-$(n+1)$ matrix containing a set of points representing the unit circle in (u, v) axes,

$$\mathcal{C}_{uv} = \begin{bmatrix} u_0 & \cdots & u_n \\ v_0 & \cdots & v_n \end{bmatrix} \tag{89}$$

with $u_i = \cos(2\pi i/n)$ and $v_i = \sin(2\pi i/n)$, for $i = \{0, \dots, n\}$. Then build the ellipse with semi axes na and nb, and rotate and translate it according to R and $\overline{\mathbf{x}} = [\overline{x}, \overline{y}]^\top$

$$\mathcal{E}_{xy} = n\mathsf{R}\sqrt{\mathbf{D}}\mathcal{C}_{uv} + \begin{bmatrix} \overline{x} & \cdots & \overline{x} \\ \overline{y} & \cdots & \overline{y} \end{bmatrix} \tag{90}$$

A plot of the contents of $\mathcal{E}_{xy}$ completes the drawing process.

B.3.2 MATLAB implementation

To draw the 2-sigma ellipsoid of a 2D Gaussian with mean $\overline{\mathbf{x}}$ and covariances matrix $\mathbf{P}$, type the code

```
x = [1;2];                          % for example
P = [2 3;3 2];                       % for example
[xx,yy] = cov2elli(x, P, 3);         % 3-sigma ellipse's coordinates
plot(xx,yy);
axis equal
```

The key in this code is the function **`cov2elli()`** which returns two sets of coordinate values to be plotted as a line. This line draws the 2-sigma ellipse.

The code for **`cov2elli`** follows

```
function [X,Y] = cov2elli(x,P,n,NP)

% COV2ELLI  Ellipse contour from Gaussian mean and covariances matrix.
%    [X,Y] = COV2ELLI(X0,P) returns X and Y coordinates of the contour of
%    the 1-sigma ellipse of the Gaussian defined by mean X0 and covariances
%    matrix P. The contour is defined by 16 points, thus both X and Y are
%    16-vectors.
%
%    [X,Y] = COV2ELLI(X0,P,n,NP) returns the n-sigma ellipse and defines the
%    contour with NP points instead of the default 16 points.
%
%    The ellipse can be plotted in a 2D graphic by just creating a line
%    with 'line(X,Y)' or 'plot(X,Y)'.

%    Copyright 2008-2009 Joan Sola @ LAAS-CNRS.

if nargin < 4
    NP = 16;
    if nargin < 3
        n = 1;
    end
end

alpha  = 2*pi/NP*(0:NP);            % NP angle intervals for one turn
circle = [cos(alpha);sin(alpha)];   % the unit circle

% SVD method, P = R*D*R' = R*d*d*R'
[R,D]=svd(P);
d = sqrt(D);
% n-sigma ellipse <- rotated 1-sigma ellipse <- aligned 1-sigma ellipse <- unit circle
ellip = n * R * d * circle;

% output ready for plotting (X and Y line vectors)
X = x(1)+ellip(1,:);
Y = x(2)+ellip(2,:);
```

C Matlab code

Proof-ready functions to perform 2D EKF-SLAM with a range-and-bearing sensor are given below. They implement most of the material presented in this brief guide to EKF-SLAM. You can directly copy-paste them into your Matlab editor.

We differentiate between elementary function blocks and the functions SLAM really needs. The SLAM functions are often compositions of elementary functions.

All functions are able to return the Jacobian matrices of the output variables with respect to each one of the input variables.

Also, some of the functions here include a foot section with Matlab symbolic code for either constructing Jacobian matrices or testing if the function's code for the Jacobians is correct. To execute this code, put Matlab in cell mode and execute the foot cell.

Finally, we give a simple but sufficient example of a fully working SLAM algorithm, with simulation, estimation and graphics output. The program utilizes only 60 lines of code. With the functions it adds up to around 200 lines of code.

C.1 Elementary geometric functions

C.1.1 Frame transformations

Express a global point in a local frame:

```
function [pf, PF_f, PF_p] = toFrame(F , p)
%   TOFRAME transform point P from global frame to frame F
%
%   In:
%       F :     reference frame         F = [f_x ; f_y ; f_alpha]
%       p :     point in global frame   p = [p_x ; p_y]
%   Out:
```

```
%       pf:     point in frame F
%       PF_f:   Jacobian wrt F
%       PF_p:   Jacobian wrt p

%   (c) 2010, 2011, 2012 Joan Sola

t = F(1:2);
a = F(3);

R = [cos(a) -sin(a) ; sin(a) cos(a)];

pf = R' * (p - t);

if nargout > 1 %  Jacobians requested
    px = p(1);
    py = p(2);
    x = t(1);
    y = t(2);

    PF_f = [...
        [ -cos(a), -sin(a),   cos(a)*(py - y) - sin(a)*(px - x)]
        [  sin(a), -cos(a), - cos(a)*(px - x) - sin(a)*(py - y)]];

    PF_p = R';
end
end

function f()
%% Symbolic code below -- Generation and/or test of Jacobians
% - Enable 'cell mode' to use this section
% - Left-click once on the code below - the cell should turn yellow
% - Type ctrl+enter (Windows, Linux) or Cmd+enter (MacOSX) to execute
% - Check the Jacobian results in the Command Window.
syms x y a px py real
F = [x y a]';
p = [px py]';
pf = toFrame(F, p);
PF_f = jacobian(pf, F)
end
```

Express a local point in the global frame:

```
function [pw, PW_f, PW_pf] = fromFrame(F, pf)
%   FROMFRAME Transform a point PF from local frame F to the global frame.
%
%   In:
%       F :     reference frame     F  = [f_x ; f_y ; f_alpha]
%       pf:     point in frame F    pf = [pf_x ; pf_y]
%   Out:
%       pw:     point in global frame
%       PW_f:   Jacobian wrt F
%       PW_pf:  Jacobian wrt pf
%   (c) 2010, 2011, 2012 Joan Sola

t = F(1:2);
a = F(3);

R = [cos(a) -sin(a) ; sin(a) cos(a)];

pw = R*pf + repmat(t,1,size(pf,2));  % Allow for multiple points

if nargout > 1 %  Jacobians requested

    px = pf(1);
    py = pf(2);

    PW_f =  [...
        [ 1, 0, - py*cos(a) - px*sin(a)]
        [ 0, 1,   px*cos(a) - py*sin(a)]];

    PW_pf = R;

end
end

function f()
%% Symbolic code below -- Generation and/or test of Jacobians
% - Enable 'cell mode' to use this section
% - Left-click once on the code below - the cell should turn yellow
% - Type ctrl+enter (Windows, Linux) or Cmd+enter (MacOSX) to execute
% - Check the Jacobian results in the Command Window.
syms x y a px py real
F = [x;y;a];
pf = [px;py];
pw = fromFrame(F,pf);
PW_f = jacobian(pw,F)
PW_pf = jacobian(pw,pf)
end
```

C.1.2 Project to sensor

```
function [y, Y_p] = scan (p)
%   SCAN perform a range-and-bearing measure of a 2D point.
%
%   In:
%       p :     point in sensor frame   p = [p_x ; p_y]
%   Out:
%       y :     measurement             y = [range ; bearing]
%       Y_p:    Jacobian wrt p

%   (c) 2010, 2011, 2012 Joan Sola
```

```
px = p(1);
py = p(2);

d = sqrt(px^2+py^2);
a = atan2(py,px);
% a = atan(py/px);  % use this line if you are in symbolic mode.

y = [d;a];

if nargout > 1 %  Jacobians requested

    Y_p = [...
        px/sqrt(px^2+py^2)          , py/sqrt(px^2+py^2)
        -py/(px^2*(py^2/px^2 + 1)), 1/(px*(py^2/px^2 + 1)) ];

end
end

function f()
%% Symbolic code below -- Generation and/or test of Jacobians
% - Enable 'cell mode' to use this section
% - Left-click once on the code below - the cell should turn yellow
% - Type ctrl+enter (Windows, Linux) or Cmd+enter (MacOSX) to execute
% - Check the Jacobian results in the Command Window.
syms px py  real
p = [px;py];
y = scan(p);
Y_p = jacobian(y,p)
[y,Y_p] = scan(p);
simplify(Y_p - jacobian(y,p))
end
```

C.1.3 Back project from sensor

```
function [p, P_y] = invScan(y)
%   INVSCAN Backproject a range-and-bearing measure into a 2D point.
%
%   In:
%       y :     range-and-bearing measurement y = [range ; bearing]
%   Out:
%       p :     point in sensor frame   p = [p_x ; p_y]
%       P_y:    Jacobian wrt y

%   (c) 2010, 2011, 2012 Joan Sola

d = y(1);
a = y(2);

px = d*cos(a);
py = d*sin(a);
```

```
p = [px;py];

if nargout > 1 %  Jacobians requested

   P_y = [...
       cos(a) , -d*sin(a)
       sin(a) , d*cos(a)];

end
```

C.2 SLAM level operations

C.2.1 Robot motion

```
function [ro, RO_r, RO_n] = move(r, u, n)
%   MOVE Robot motion, with separated control and perturbation inputs.
%
%   In:
%       r: robot pose       r = [x ; y ; alpha]
%       u: control signal   u = [d_x ; d_alpha]
%       n: perturbation, additive to control signal
%   Out:
%       ro: updated robot pose
%       RO_r: Jacobian   d(ro) / d(r)
%       RO_n: Jacobian   d(ro) / d(n)

a = r(3);

if nargin > 2
    dx = u(1) + n(1);
    da = u(2) + n(2);
else
    dx = u(1);
    da = u(2);
end

ao = a + da;

if ao > pi
    ao = ao - 2*pi;
end
if ao < -pi
    ao = ao + 2*pi;
end

% build position increment dp=[dx;dy], from control signal dx
dp = [dx;0];

if nargout == 1 % No Jacobians requested

    to = fromFrame(r, dp);
```

```
else %  Jacobians requested

    [to, TO_r, TO_dt] = fromFrame(r, dp);
    AO_a = 1;
    AO_da = 1;

    RO_r = [TO_r ; 0 0 AO_a];
    RO_n = [TO_dt(:,1) zeros(2,1) ; 0 AO_da];

end

ro = [to;ao];
```

C.2.2 Direct observation model

Note: The function **observe.m** has been purposely left out. You need to write it !

C.2.3 Inverse observation model

```
function [p, P_r, P_y] = invObserve(r, y)
%   INVOBSERVE Backproject a range-and-bearing measurement and transform
%   to map frame.
%
%   In:
%       r :     robot frame     r = [r_x ; r_y ; r_alpha]
%       y :     measurement     y = [range ; bearing]
%   Out:
%       p :     point in global frame
%       P_r:    Jacobian wrt r
%       P_y:    Jacobian wrt y

%   (c) 2010, 2011, 2012 Joan Sola

if nargout == 1 % No Jacobians requested

    p   = fromFrame(r, invScan(y));

else %  Jacobians requested

    [p_r, PR_y]    = invScan(y);
    [p, P_r, P_pr] = fromFrame(r, p_r);

    % here the chain rule !
    P_y = P_pr * PR_y;

end
end

function f()
```

```
%% Symbolic code below -- Generation and/or test of Jacobians
% - Enable 'cell mode' to use this section
% - Left-click once on the code below - the cell should turn yellow
% - Type ctrl+enter (Windows, Linux) or Cmd+enter (MacOSX) to execute
% - Check the Jacobian results in the Command Window.
syms rx ry ra yd ya real
r = [rx;ry;ra];
y = [yd;ya];
[p, P_r, P_y] = invObserve(r, y); % We extract also the coded Jacobians P_r and P_y
% We use the symbolic result to test the coded Jacobians
simplify(P_r - jacobian(p,r))  % zero-matrix if coded Jacobian is correct
simplify(P_y - jacobian(p,y))  % zero-matrix if coded Jacobian is correct
end
```

C.3 EKF-SLAM code

It follows a m-file performing SLAM. This code uses all the files above, including the function **observe.m** that you have to code yourself.

Please read all help notes in this file carefully. They explain a number of important things not covered (because they relate to implementation issues) in the main body of this document.

NOTE: This script has been purposely corrupted. You need to complete it, by writing all the SLAM code (you have only the simulation and graphics code). You have also important guidelines as comments. Have fun!

```

% SLAM2D  A 2D EKF-SLAM algorithm with simulation and graphics.
%
%  HELP NOTES:
%  1. The robot state is defined by [xr;yr;ar] with [xr;yr] the position and
%     [ar] the orientation angle in the plane.
%  2. The landmark states are simply Li=[xi;yi]. They are stored in a internal
%     variable of the simulator.
%  3. The control signal for the robot is u=[dx;da] where [dx] is a forward
%     motion and [da] is the angle of rotation. It is fixed in the simulator
%     and accessible using the function sim_get_control_signal()
%  4. The motion perturbation is additive Gaussian noise n=[nx;na] with
%     covariance Q, which adds to the control signal. It is computed from the
%     standard deviation vector q.
%  5. The measurements are range-and-bearing Yi=[di;ai], with [di] the
%     distance from the robot to landmark Li, and [ai] the bearing angle from
%     the robot's x-axis. The measurements are accessible through the function
%     sim_get_lmk_measurement(i)
%  6. The true map is a unknown vector of the form
%     [xr;yr;ar;x1;y1;x2;y2;x3;y3; ... ;xN;yN]
%  7. The estimated map is Gaussian, defined by
%       x: mean of the map
%       P: covariances matrix of the map
%  8. The estimated entities (robot and landmarks) are extracted from {x,P}
%     via "pointers", which are just indices in the matrices x,P that define
%     the Gaussian. We suggest you use the following nomenclature, denoted in
%     small letters:
%       r:  pointer to robot state. Usually the first three elements:
%           r=[1,2,3]
%       l:  pointer to a landmark. For example, if we have several landmarks
%           in the map, then l=[4,5] is the pointer for the first landmark,
%           l=[6,7] for the second, and so on.
%       m:  pointers to all used landmarks. You should build it based on the
%           current known landmarks. For example, for 3 landmarks you would
%           have m=[4,5,6,7,8,9]. See lm_all_lmk_pointers helper function
%           further below.
%       rl: pointers to robot and one landmark. Build it from r and l, as rl =
%           [r;l]
%       rm: pointers to robot and all landmarks (the currently used map).
%           Build it using r and m vectors with rm=[r;m]
%     Therefore:  x(r)     is the robot state,
%                 x(l)     is the state of one landmark
%                 P(r,r)   is the covariance of the robot
%                 P(l,l)   is the covariance of one landmark
%                 P(r,l)   is the cross-variance between robot and a lmk
%                 P(rm,rm) is the current full covariance -- the rest is
%                 unused.
%     NOTE: Pointers are always row-vectors of integers.
%  9. Managing the map space is done through map management helper functions,
%     named with prefix mm_*. Use it as follows:
%       * mm_query_space(n)  -> returns pointer fs to n free spaces. It will
%                               return a pointer fs to the space available.
%                               length(fs) will be n if there is at least n
%                               spaces available, and less than n if there are
%                               less than n spaces available.
%       * mm_block_space(fs) -> block positions indicated in vector fs
%       * mm_free_space(fs)  -> liberate positions indicated in vector fs
%     NOTE: Don't forget to check if the number of elements in fs matches the
%           number of queried spaces.
% 10. Managing the existing landmarks is done through the use of landmark
%     management helper functions, named with prefix lm_*. Use it as follows:
%       * lm_find_non_mapped_lmk()           -> look for one non-mapped
%                                               landmark
%       * lm_associate_pointer_to_lmk(fs,i) -> associate free space pointer fs
%                                               to landmark i
%       * lm_lmk_pointer(i)                  -> recover a landmark pointer l
%       * lm_all_lmk_pointers()              -> recover pointers to all known
%                                               landmarks
%       * lm_forget_lmk(i)                   -> forget landmark i
%       * lm_all_lmk_ids()                   -> recover the id of all known
%                                               landmarks
% 11. The simulation can be accessed using sim_* functions. The available
%     functions are:
%       * sim_get_control_signal()     -> recover the current control signal
%       * sim_get_lmk_measurement(i)   -> recover measurement to landmark i
%       * sim_get_initial_robot_pose() -> recover the initial pose of the
%                                         robot
%       * sim_simulate_one_step()      -> perform one step of simulation. It
%                                         is already used where it should be
%                                         called. Do NOT use it again.
% 12. The init_* functions  are used to initialize the simulator. You should
%     not use nor change them in the code below.
%
%   (c) 2010, 2011, 2012, 2013, 2014, 2015, 2016 Joan Sola.
%   (c) 2016, 2017 Ellon Paiva Mendes.


% I. INITIALIZE ===========================================================

% I.1 SIMULATOR ------------------------------------------------------------

init_simulator

% I.2 ESTIMATOR ------------------------------------------------------------

% a. Define Map: Gaussian {x,P}
%   mapsize: size of the map
mapsize = 100;
%   x: state vector's mean
x = zeros(mapsize,1);
%   P: state vector's covariances matrix
P = zeros(mapsize,mapsize);

% b. Define system noise: Gaussian {0,Q}
% q = <...>;      % amplitude or standard deviation
% Q = diag(q.^2); % covariance matrix, built from the standard deviation

% c. Define measurement noise: Gaussian {0,S}
% s = <...>;      % amplitude or standard deviation
% S = diag(s.^2); % covariance matrix, built from the standard deviation

% d. Map management
init_map_management(mapsize); % See Help Note #12 above.

% e. Landmarks management
init_landmark_management;     % See Help Note #12 above.

% f. Initialize robot in map
% query for map space (see note 9)
r = mm_query_space(3); % get pointers to robot space in the map.
mm_block_space(r);     % block map positions

% init mean and covariance
% x(r)   = <...>;    % initialize robot state. See Help Note #11 above.
% P(r,r) = <...>;    % initialize robot covariance

% I.3 GRAPHICS ------------------------------------------------------------

init_graphics
```

```
% II. TEMPORAL LOOP =======================================================

for t = 1:200


    % II.1 SIMULATOR ------------------------------------------------------

    sim_simulate_one_step();

    % II.2 ESTIMATOR ------------------------------------------------------

    % NOTE: YOU MUST COMPLETE THIS WHOLE SECTION BY ENTERING YOUR CODE.
    %       After coding the function observe.m, uncoment all the lines
    %       below _once_ and follow the instructions in the remaining
    %       comments.
    %       Tip: Select all the lines in this section and press ctrl+t once
    %       to remove one comment level.

%     % a. create useful map pointers to be used hereafter
%     %-----------------------------------------------------
%     % vector with all pointers to known landmarks (see note 10)
%     % NOTE: once you defined m pointer, add it to the draw_graphics
%     %       function in the end of this file.
%     m  = <...>;
%     % vector with pointers to all used states: robot and known landmarks
%     % (see note 8)
%     rm = <...>;
%
%
%     % b. Prediction -- robot motion
%     %-------------------------------
%
%     % Update {x(r), P}: using the function move(),
%     %   compute new robot position x(r),
%     %   and obtain Jacobians R_r, R_n.
%     %   (see note 11)
%     [x(r), R_r, R_n] = <...>
%     % Then update covariances matrix P --> use sparse formulation, e.g.:
%     P(r,m)  = <...>
%     P(m,r)  = P(r,m)';
%     P(r,r)  = <...>
%
%
%     % c. Landmark correction -- known landmarks
%     %-------------------------------------------
%
%     % c1. obtain all indices of existing landmarks (see note 10)
%     lids = <...>;
%
%     % c2. loop for all known landmarks
%     for i = lids
%
%         % c3. Obtain pointers to landmark 'i' (see note 10)
%         l = <...>; % landmark's pointers vector
%
%         % c4. compute expectation Gaussian {e,E}  --- this is h(x) in your
%         % notes
%         [e, E_r, E_l] = <...>;
%         E = <...>;
%
%         % c5. get measurement Yi for landmark 'i', as Yi=[dist,angle]' (see
%         % note 11)
%         Yi = <...>
%
%         % c6. compute innovation --- this is y-h(x), or y-e as we have
%         % e=h(x)
%         z = <...>;
%         % remember to bring angular innovations z(2) around zero, not
%         % multiples of 2pi !
%         <...>
%
%         % c7. compute innovation covariance
%         Z = <...>
%
%         % Mahalanobis test (Ask BE responsible for justification)
%         if z' * Z^-1 * z < 9
%
%             % c8. compute Kalman gain
%             K = <...>;
%
%             % c9. Perform Kalman update
%             x(rm)    = <...>;
%             P(rm,rm) = <...>;
%
%         end % Mahalanobis test
%
%     end % loop for all known landmarks
%
%
%
%     % d. Landmark Initialization -- one new landmark only at each iteration
%     %------------------------------------------------------------------------
%
%     % d1. Get ID of a non-mapped landmark (see note 10)
%     i = <...>;
%     if ~isempty(i)
%         % d2. Get pointers vector of the new landmark in the map (see
%         % note 9)
%         l = <...>;
%
%         % if there is still space for a new landmark in the map
%         if numel(l) ≥ 2
%
```

```
%               % d3. block map space (see note 9)
%               <...>;
%
%               % d4. store landmark pointers (see note 10)
%               <...>;
%
%               % d5. measurement Yi of landmark 'i'  (see note 11)
%               Yi = <...>;
%
%               % d6. Landmark initialization in the map. Update x and P:
%               %   get x(l), landmark state
%               %   get Jacobians L_r and L_y
%               [x(l), L_r, L_y] = <...>
%               P(l,rm)          = <...>;
%               P(rm,l)          = <...>;
%               P(l,l)           = <...>;
%
%           end % if numel(l) ≥ 2
%       end % if ~isempty(i)

    % II.3 GRAPHICS ----------------------------------------------------------

    % a. Add the pointer m as the last argument to the draw_graphics
    draw_graphics(x,P,r)

    % If needed, use next line to slow down the loop
    % pause(1)
end
```

References

[1] Cyril Roussillon, Aurélien Gonzalez, Joan Solà, Jean Marie Codol, Nicolas Mansard, Simon Lacroix, and Michel Devy. RT-SLAM: a generic and real-time visual SLAM implementation. In *Int. Conf. on Computer Vision Systems (ICVS)*, Sophia Antipolis, France, September 2011.

[2] R. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. *Int. Journal of Robotics Research*, 5(4):56–68, 1987.

[3] Joan Solà, David Marquez, Jean Marie Codol, and Teresa Vidal-Calleja. An EKF-SLAM toolbox for MATLAB, 2009.

[4] 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/.