J Braz Comput Soc (2013) 19:181–192
DOI 10.1007/s13173-012-0092-4

ORIGINAL PAPER

A method to convert floating to fixed-point EKF-SLAM for embedded robotics

Leandro de Souza Rosa · Vanderlei Bonato

Received: 20 February 2012 / Accepted: 15 October 2012 / Published online: 8 November 2012

Abstract The Extended Kalman Filter (EKF) is one of the most efficient algorithms to address the problem of Simultaneous Localization And Mapping (SLAM) in the area of autonomous mobile robots. The EKF simultaneously estimates a model of the environment (map) and the position of a robot based on sensor information. The EKF for SLAM is usually implemented using floating-point data representation demanding high computational processing power, mainly when the processing is performed online during the environment exploration. In this paper, we propose a method to automatically estimate the bit-range of the EKF variables to mitigate its implementation using only fixed-point representation. In this method is presented a model to monitor the algorithm stability, a procedure to compute the bit range of each variable and a first effort to analyze the maximum acceptable system error. The proposed system can be applied to reduce the overall system cost and power consumption, specially in SLAM applications for embedded mobile robots.

Keywords EKF-SLAM · Fixed-point · Bit-range analysis · Embedded mobile robots

1 Introduction

The Extended Kalman Filter (EKF) [1] is one of the most important algorithms in the area of mobile robotics for the localization and mapping tasks due to its ability to deal with noise and its high degree of accuracy [2]. Derived from Bayesian filters [3], EKF has high computational complexity and performs operations over a large volume of data. Applying such a solution in the development of embedded mobile robots is highly desired, since it allows one to build intelligent machines capable of acting autonomously in complex environments.

Implementing this algorithm in a customized device is usually desired when a typical solution of a general purpose processor is unsatisfactory, such as when the robotic system has limited cost, performance and power requirements. Developing an optimized processing system, such as the one presented in Bonato et al. [4], where an EKF floating-point based system on an FPGA (Field-Programmable Gate Array) was implemented, is a feasible way to achieve the requirements via specific customizations of software and hardware. Customized system operating over fixed-point format is another way to obtain further optimizations for embedded robotic applications.

There are different ways to convert algorithms from floating-point to fixed-point format. Most solutions are completely automatic and orientated to DSP applications [5,6]. The main task in converting an algorithm is to estimate the bit-range of each single variable in such way to avoid an error bigger than the one allowed by the algorithm without compromising the overall system accuracy, which can be caused by underflows and overflows of variables.

Recent work has used fixed-point representation to implement the EKF algorithm to solve the SLAM problem. In Moyers et al. [7], we have a fixed-point implementation, but without bit length optimization, and in, Mingas et al. [8], there exists another fixed-point implementation, with bit lengths of all variables defined by physical constraints of a robot. In this paper is the first work that presents an approach to estimate a

L. de Souza Rosa (✉)
Institute of Mathematical and Computing Sciences,
The University of São Paulo, São Carlos, Brazil
e-mail: le.souza.rosa@gmail.com; leandrors@usp.br

V. Bonato
São Carlos School of Engineering,
The University of São Paulo, São Carlos, Brazil
e-mail: vbonato@icmc.usp.br

particular bit length for each according to a maximum error defined by the user. The main contributions of this paper are:

- a method to monitor the EKF error during the conversion process;
- an optimized way to reduce the time needed for the conversion process;
- the bit range of each EKF variable for a given maximum error.

The paper is organized as follows. Section 2 briefly introduces the EKF algorithm for SLAM along with a table describing each of its variables. Section 3 presents the floating to fixed-point conversion algorithm steps, which was adapted from Roy and Banerjee [9]. The error analysis method is demonstrated in Sect. 4 followed by Sect. 5, where the conversion results are presented. Finally, Sect. 6 concludes the paper.

2 Extended Kalman Filter

This section briefly describes the EKF algorithm and a complete description, with the derivation of each equation, can be found in [2]. The source code used as base in our work can be founded in Newman [10].

2.1 EKF description

EKF is composed of two phases: prediction and update. In the SLAM context, the prediction phase estimates the robot position $\mu_v^{(t)}$, at time t, based on a prior believed position $\mu_v^{(t-1)}$ and on a movement control $u^{(t)}$, while the update phase integrates robot sensor observations $z^{(t)}$ in order to update a map of the robot environment and to, again, estimate the robot position. These two steps are repeated for each EKF iteration, where the data estimated at one iteration are used as input to the next one.

For this paper, we are considering the robot position comprising of two-dimensional planar coordinates (x, y) relative to some external coordinate frame, along with its angular orientation θ. The map generated by the EKF-SLAM consists of a set of feature positions $(\mu_{f_1}^{(t)}, \mu_{f_2}^{(t)}, \ldots, \mu_{f_n}^{(t)})$ detected from the robot navigation environment by a sensor, where each feature is represented in the same way by its (x, y) coordinates. Thus the robot state size is three and the feature state size is two; these parameters are represented in this paper by r and s, respectively, for generalization. As these robot and feature states are estimated, they have an associated covariance matrix in order to represent their uncertainty, which is represented by $\Sigma^{(t)}$ at time t. In the EKF algorithmthese data are organized as in (1), where $\mu^{(t)}$ is composed of the estimated robot position and the feature set at time t.

$$\mu^{(t)} = \begin{bmatrix} \mu_v^{(t)} \\ \mu_{f_1}^{(t)} \\ \mu_{f_2}^{(t)} \\ \vdots \\ \mu_{f_n}^{(t)} \end{bmatrix}, \quad \Sigma^{(t)} = \begin{bmatrix} \Sigma_{vv}^{(t)} & \Sigma_{vf_1}^{(t)} & \ldots & \Sigma_{vf_n}^{(t)} \\ \Sigma_{f_1v}^{(t)} & \Sigma_{f_1f_1}^{(t)} & \ldots & \Sigma_{f_1f_n}^{(t)} \\ \Sigma_{f_2v}^{(t)} & \Sigma_{f_2f_1}^{(t)} & \ldots & \Sigma_{f_2f_n}^{(t)} \\ \vdots & \vdots & \ddots & \\ \Sigma_{f_nv}^{(t)} & \Sigma_{f_nf_1}^{(t)} & \ldots & \Sigma_{f_nf_n}^{(t)} \end{bmatrix} \tag{1}$$

The uncertainty cames from the odometry noise and the observation noise simulated. If we consider a perfect odometry and observation, the matrix $\Sigma^{(t)}$ would have all elements equal to zero forever (noise free), however, for real odometry and observation the noise needs to be considered in order to have a more robust system.

Table 1 describes the variables used in the prediction and update EKF equations along with their dimensions. Equations (2) and (3) are used to estimate the new robot position given the belief vector $\mu_v^{(t-1)}$ and the covariance matrix that correspond to the robot position $\Sigma_{vv}^{(t-1)}$ and the current motion control $u^{(t)}$. $F^{(t)}$ and $G^{(t)}$ are Jacobian matrices containing derivatives of the prediction function α with respect to the motion command variables at time t. Equation (4) estimates the covariance between the robot and feature position Σ_{vf} given the corresponding covariance from time $(t-1)$ and the matrix $F^{(t)}$.

After computing the prediction equations, the update step starts by predicting the sensor measurement through the measurement function equation (9) using the estimated robot position $\mu_v^{(t)}$ and the detected feature $\mu_{fi}^{(t-1)}$. Then, Eqs. (10) and (11) calculate the innovation related to the measurement $\nu^{(t)}$ and covariance $S^{(t)}$, respectively. $\nu^{(t)}$ is the difference between the real and the estimated sensor measurement and $S^{(t)}$ is the new information added to the system covariance given the current matrix $H^{(t)}$ and the covariance from the previous prediction phase. $H^{(t)}$ is a matrix that compounds two Jacobian matrices $H_v^{(t)}$ and $H_{fi}^{(t)}$ from (8) which are derivatives of the prediction function γ with respect to the estimated robot position and the detected feature at time t. Then, Eq. (12) computes the filter weight. Finally, Eqs. (5) and (6) update all data that corresponds to the estimated map.

The integer and fraction bit-range estimated for each variable presented in Table 1 is shown in Sect. 5, which can be used as a reference for the development of a fixed-point hardware architecture customized for the EKF algorithm.

Prediction:

$$\mu_v^{(t)} = \alpha(\mu_v^{(t-1)}, u^{(t)}) \tag{2}$$

$$\Sigma_{vv}^{(t)} = F^{(t)}\Sigma_{vv}^{(t-1)}F^{(t)T} + G^{(t)}QG^{(t)T} \tag{3}$$

$$\Sigma_{vf}^{(t)} = F^{(t)}\Sigma_{vf}^{(t-1)} \tag{4}$$

Table 1 The description and dimension of the EKF symbols, where s and r represent the feature and robot state size, v and f the robot and feature position, i the feature number and n the total number of features

Sym.	Dimension	Description
μ	$(r+sn)\times 1$	Both robot and feature positions
μ_v	$r\times 1$	Elements of μ related to robot position
μ_f	$sn\times 1$	Elements of μ related to feature position
Σ_{vv}	$r\times r$	Robot position covariance
Σ_{vf}	$r\times(sn)$	Cross robot-feature covariance
Σ_{ff}	$(sn)\times(sn)$	Cross feature-feature covariance
Σ	$(r+sn)\times$ $(r+sn)$	Cross robot-feature and feature-feature covariance
α	–	Prediction function
γ	–	Measurement function
u	$r\times 1$	Robot motion command
F	$r\times r$	Robot motion Jacobian
G	$r\times r$	Robot motion noise Jacobian
Q	$r\times r$	Permanent motion noise
H_v	$s\times r$	Measurement Jacobian with respect to v
H_{fi}	$s\times s$	Measurement Jacobian with respect to f_i
H	$s\times(r+sn)$	Compounded measurement Jacobian
R	$s\times s$	Permanent measurement noise
W	$(r+sn)\times s$	Filter gain
ν	$s\times 1$	Mean innovation
z	$s\times 1$	Sensor measurement
z_{pred}	$s\times 1$	Sensor measurement prediction
S	$s\times s$	Covariance innovation
Z_1	$s\times(s(i-1))$	Zero matrix
Z_2	$s\times(s(n-i))$	Zero matrix

Update:

$$\mu^{(t)} = \bar{\mu}^{(t)} + W^{(t)}\nu^{(t)} \tag{5}$$

$$\Sigma^{(t)} = \bar{\Sigma}^{(t)} - W^{(t)}S^{(t)}W^{(t)T} \tag{6}$$

where:

$$\bar{\mu}^{(t)} = \begin{bmatrix} \mu_v^{(t)} \\ \mu_f^{(t-1)} \end{bmatrix}, \bar{\Sigma}^{(t)} = \begin{bmatrix} \Sigma_{vv}^{(t)} & \Sigma_{vf}^{(t)} \\ \Sigma_{vf}^{(t)T} & \Sigma_{ff}^{(t-1)} \end{bmatrix} \tag{7}$$

$$H^{(t)} = \begin{bmatrix} H_v^{(t)} & Z_1 & H_{fi}^{(t)} & Z_2 \end{bmatrix} \tag{8}$$

$$z_{\text{pred}}^{(t)} = \gamma(\mu_v^{(t)}, \mu_{fi}^{(t-1)}) \tag{9}$$

$$\nu^{(t)} = z^{(t)} - z_{\text{pred}}^{(t)} \tag{10}$$

$$S^{(t)} = H^{(t)}\bar{\Sigma}^{(t)}H^{(t)T} + R \tag{11}$$

$$W^{(t)} = \bar{\Sigma}^{(t)}H^{(t)T}S^{(t)^{-1}} \tag{12}$$

$$\mathbf{F} = \frac{\partial f}{\partial x_v} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \tag{13}$$

$$\mathbf{G} = \frac{\partial f}{\partial u} = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \cdots & \frac{\partial f_1}{\partial u_m} \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial u_1} & \cdots & \frac{\partial f_n}{\partial u_m} \end{bmatrix} \tag{14}$$

$$\mathbf{H_v} = \frac{\partial h}{\partial x_v} = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \cdots & \frac{\partial h_1}{\partial x_m} \\ \vdots & & \vdots \\ \frac{\partial h_n}{\partial x_1} & \cdots & \frac{\partial h_n}{\partial x_m} \end{bmatrix} \tag{15}$$

$$\mathbf{H_f} = \frac{\partial h}{\partial x_f} = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \cdots & \frac{\partial h_1}{\partial x_m} \\ \vdots & & \vdots \\ \frac{\partial h_n}{\partial x_1} & \cdots & \frac{\partial h_n}{\partial x_m} \end{bmatrix} \tag{16}$$

3 Conversion algorithm

In this section, we present the conversion algorithm adapted from Roy and Banerjee [9] to estimate the bit-range of each fixed-point EKF variable. The algorithm is divided in eight steps. The first two conversion steps, *Levelization* and *Scalarization*, are used to modify the algorithm source code to make the process more systematic so as to obtain better results. The third step, *Computation of Ranges of Variables*, computes the maximum values of each algorithm variable. Then, the fourth step, *Evaluate Integer Variables*, figures out the variables of the source code that are integers, followed by the fifth and sixth steps, where the source code is converted to fixed-point representation and the integer range of the fixed point representation of each variable is evaluated, respectively. Finally, the last two steps, *Coarse Optimization and Fine Optimization*, optimize the final bit-range of each variable.

Comparing this with the conversion algorithm given in Roy and Banerjee [9], the *Computation of Ranges of Variables* and *Fine Optimization* were modified, and, the step *Evaluate Integer Variables*, was added, keeping the other steps as they were.

As input to this conversion procedure, we used a floating-point code of the EKF algorithm implemented in Matlab, adapted from Newman [10]. The use of Matlab code is appropriated, since it provides a set of facilities needed by the conversion algorithm. The following subsection starts presenting an introduction to the terms adopted by the conversion procedure. Then, the next subsections describe each conversion step presented along with examples.

3.1 Introduction and nomenclature

The conversion algorithm basically uses the objects 'quantizer()' and 'quantize()' from the Filter Design Analysis (FDA) toolbox available in the MATLAB [11]. Here, we will use `m` for the integer and `p` for the fractional length of any variable. The 'quantizer()' function is a constructor to 'quantize' objects which allocates the bit-widths and the information about the representation, as if the representation

is signed or unsigned, the kind of rounding and the overflow treatment. Furthermore, the 'quantize()' applies the 'quantize' object in a number, and gives as a return the value of the number represented by an integer data type defined by the 'quantize'.

The error measure is defined as the difference, in percentual, between the result of the floating-point and the fixed-point execution, and can be written as (17), where $\text{outdata}_{\text{float}}$ and $\text{outdata}_{\text{fixed}}$ are the output of the floating-point and fixed-point filter versions, respectively.

$$E = \frac{\text{norm}(\text{outdata}_{\text{float}} - \text{outdata}_{\text{fixed}}) \times 100}{\text{norm}(\text{outdata}_{\text{float}})} \tag{17}$$

Once the simulated noise is fixed, this error measures the impact of the conversion between floating point to fixed point. As seen in Sect. 3.6, the error due to conversion is incorporated by the covariance matrix, and then, propagated to all variables.

For all cases, n is the number of variables that the algorithm to be converted has after the *Scalarization step*. As each variable has a different number of bits lengths, we represent m, p and q as vectors, of n positions, representing the integer-width, the fractional width and 'quantizers', respectively, for each variable.

3.2 Levelization

The first step of conversion expands each complex code statement to simple statements containing only one-operation each. As an example, we present the code fragment 3.1 from our code where the complex statement is commented on (line 1) and the following statements are its levelization (lines 2 to 4). To perform such a transformation it was necessary to add temporary variables to the code (variables 'temp1' and 'temp2').

```
1 %PPredvm = J1f(xVehicle,u)*PEst(1:3,4:end);
2 temp1 = J1f(xVehicle,u)
3 temp2 = PEst(1:3,4:end)
4 PPredvm = temp1*temp2;
```

Code Fragment 3.1: Example of Levelization.

3.3 Scalarization

This step converts the vectorized MATLAB statements for scalar ones using FOR loops. The code fragment 3.2 exemplify the scalarization step.

```
1  %var38 = var27*var17(1:3,4:end);
2  var34 = length(var17)-3;
3  var38 = zeros(3, var34);
4  for var29=1:3
5      for var30=1:var34
6          for var31=1:3
7          var38(var29, var30) = var38(var29,
               var30) + var27(var29, var31)*
               var17(var31, var30+3);
8          end
9      end
10 end
```

Code Fragment 3.2: Example of Scalarization.

3.4 Computation of ranges of variables

This is the core step for calculating the m values (integer width). First, it is necessary to take the maximum and minimum values of the algorithm's inputs and then propagate them to the forward direction calculating the maximum and minimum values for each variable. To exemplify this step we are going to use a generic algorithm; the code fragment 3.3 shows an example and Table 2 demonstrates the result of propagation for this code.

As stated in Table 2, there is a problem regarding the division by zero. When it happens, it is necessary to consider a tiny number, say a unit, for the variables with the zero division problem, making the roll back calculation of maximum e minimum values. To finish, we must ensure that the entries are within the limits obtained.

```
1 function output = function(a, b)
2     x = a*b;
3     y = a/b;
4     z = x+y;
5     output = z+1;
6 end
```

Code Fragment 3.3: Generic code to exemplify the calculation of maximum e minimum values.

Another problem can occur when the algorithm to be converted stays running in FOR loops with feedbacks (some output values are used as entry values in the next loop); in that case, the maximum or minimum values of some variables may grow to infinity. This is the case for our EKF algorithm,

Table 2 Maximum and Minimum values propagated for the code fragment 3.3

Variable	*a*	*b*	*x*	*y*	*z*	Output
Maximum	100	5	500	Error/Inf	Error/Inf	Error/Inf
Minimum	0	0	0	0	0	1

When a division by zero occurs we get $\pm\infty$ (Inf), which is treated as an error

leading to a situation where the maximum and minimum values need to be estimated instead of being calculated. To perform such an estimate, we can simply define a training set and evaluate these values while executing the algorithm over this training set. The code fragment 3.4 demonstrates how to calculate the maximum value for a matrix variable. Note that using the absolute value of each variable we don't have to care about the minimum values of the variables.

```
function maximums = Max(var, maximums, i)
temp = max(max(abs(var)));

if temp > maximums(i,1)
    maximums(i,1) = temp;
end
```

Code Fragment 3.4: Code of the function used to calculate the absolute maximum values of variables.

The code fragment 3.5 exemplifies the usage of 'max()' function from code fragment 3.4, where maximums is a vector that stores the maximum values for each variable.

```
var14 = var13*var4;
maximums = Max(var14, maximums, 14);
```

Code Fragment 3.5: Example of usage of maximums function.

The problem of this procedure is that we cannot guarantee any overflows during the executions for the cases that do not belong to the training set. For this situation, we need to use a larger training set.

3.5 Evaluate integer variables

The reasoning of this step is to figure out which variables of the algorithm to be converted represent integers values, which usually appear as counters or indices variables. The step described in Sect. 3.8 gives a unique value for all p, including the integer variables, and the step described in 3.9 have to reduce the p values of the integer variables to zero, what may takes a long time. Since the integer variables should be converted too, we can save time in step 3.9 if we know which variable represents integer numbers by simply setting zero to the p values of integer variables.

The idea is to define a $n \times 1$ boolean vector, say D, where each index corresponds to a variable of the algorithm to be converted after the step 3.3. In this vector, 0 means that the correspondent variable is not an integer, and 1 means that the variable is an integer.

To evaluate which variables are integer we should know exactly which operations (statements, functions and methods) return integer values given their entries.

Table 3 Classification of some comum operations in MATLAB since their return values

AIR	Linear	FR
Length()	+	/
Size()	−	rand()
Numel()	∗	sqrt()
Ceil()	$\wedge x(x \geq 0)$	$\wedge x(x < 0)$
Floor()	det()	norm()
Zeros()		inv()
Ones()		sin()
Eye()		cos()
		tan()

We define the class "absolute integer return" (AIR) as the class of all operations that returns integers given any kind of entries, and we define the class "linear" as the class of all operations that, given integer values for all entries, returns integer values, finally, we define the class "float return" (FR) as the class of operations that is impossible to guarantee if the return is integer or non-integer values given any kind of entry.

In most cases, the classification is based on basic mathematical operations. For example, since addition, subtraction, and multiplication are linear operations and division is a FR operation, we can say that the operation y^x (raise a number, y, to the power x) is linear if $x \geq 0$, but the operation is FR if $x < 0$. Table 3 shows some common operations used in MATLAB and in our EKF-SLAM algorithm.

The Algorithm 3.1 shows the steps that should be applied in the straight forward data flow direction, for what ensures that all values in D receive the proper values, different than **NaN** (not a number in MATLAB). It is also important, before applying the Algorithm 3.1 in the main code, to apply it in the functions used in the main code to define which kind of function it is (AIR, linear, FR).

- *Start setting **NaN** for all D values.*
- *Analise each entry of the algorithm to be converted, if it is integer, set 1 in D vector, else, set 0.*
- *For each statement between the variables v_k, assigned to a variable v_r; $k, r \in \{1, ..., n\}$ $(v[r] = operation(v_{k1}, v_{k2}, ...))$, if the operation is a FR type set $i[r] = 0$; if $i[r] = NaN$ and the operation is from AIR or linear types, set $i[r] = \prod_k i[k]$; or, if $i[r] \neq NaN$ and the operation is from AIR or linear types, set $i[r] = i[r].\prod_k i[k]$.*

Algorithm 3.1: Evaluate integer variables algorithm.

3.6 Generation of fixed-point MATLAB code

This step converts the algorithm by replacing each arithmetic and assignment operation with a quantized computation as shown in code fragment 3.6.

```
1 %var14 = var13*var4;
2 [var14] = quantize(q(14), var13*var4);
3
4 %var19 = GetOdometry(1);
5 [var19, q] = GetOdometry(1, q);
6 [var19] = quantize(q(19), var19);
```

Code Fragment 3.6: Exemple of code conversion

At this point, we see that, in each statement, we have associated an error due to the conversion from floating-point to fixed-point. The EKF equations 2 to 6 show that these errors are incorporated by the Σ matrix and propagated to all other variables. Although these errors are incorporated with the system noise, if we fix the simulated odometry and observation noises, the conversion will generate an error measured by Eq. 17.

When generating the fixed-point MATLAB code for matrix multiplications we see that scalarizations results in a non-levelizated statement (as showed in code fragment 3.2 line 7). At first sight, we would have to re-apply the levelization step in this statement before the conversion to fixed-point code, however, with the following Observations 1 and 2 we can see that this "re-levelization" is irrelevant and can be skipped. In this way, the converted code for matrix multiplications is exemplified by the code fragment 3.7.

Observation 1 *0 = quantize(q, 0) ever!*

Observation 2 *If we have two variables a and b, and a 'quantizer' q, and apply the 'quantize()' method over one variable, say a, it will get your value modified, call it as qa. In this way, to add qa with b is different than to add b with a, and as consequence, apply the 'quantize()' method over a+b and qa+b will lead us to different results.*

However, if one of the variables were already quantized with q, the method 'quantize()' will present linear behavior. Say that qa = quantize(q, a) and qb = quantize(q, b), the Fig. 1 shows this behavior.

```
>> qab = quantize(q, a+b)

qab =

    0.6875

>> qab = quantize(q, qa+qb)

qab =

    0.6563
```

```
>> qab = quantize(q, qa+b)

qab =

    0.6563

>> qab = qa+qb

qab =

    0.6563
```

Fig. 1 Example of 'quantize()' method's pseudo-linearity in MATLAB

```
1  %var28 = var27*var17(1:3,1:3);
2  var28 = zeros(3);
3  for var29=1:3
4      for var30=1:3
5          for var31=1:3
6          %var28(var29, var30) = var28(var29,
               var30) + var27(var29, var31)*
               var17(var31, var30);
7          var28(var29, var30) = quantize(q(28)
               , var28(var29,var30) + var27(
               var29, var31)*var17(var31, var30
               ));
8          end
9      end
10 end
```

Code Fragment 3.7: Example of the code fragment 3.2 converted to fixed-point representation.

3.7 Fixed integral range

With the maximums vector defined, we can calculate the m values simply by using the $\log_2$ operation. The code fragment 3.8 shows the procedure where we used +2 bits to compensate the floor() round and to represent the sign bit.

```
1  %calculating the size of the integer part of
       the variables
2  m = NaN(107,1);
3  for i =1:107
4      m(i,1) = floor(log2(maximums(i)))+2;
5      if(m(i,1)<=0)
6          m(i,1) = 1;
7      end
8  end
```

Code Fragment 3.8: Function to calculate the absolute maximum value from a matrix

3.8 Coarse optimization

This step estimates a value for every p. The Algorithm 3.2 used is exactly the same presented by Roy and Banerjee [9] and it is resumed in Algorithm 3.2.

- *Define L=0, H=32.*
- *Set M = (l+H)/2.*
- *Calculate E values for all p(i)=L, p(i)=H and p(i)=M.*
- *If $E_M < E_{max}$, replace M by H, if $E_M \geq E_{max}$, replace M by L and repeat from the second step.*
- *We repeat the above three steps until we reach the* ***coarse optimal quantizer*** *value(s) p satisfying $E_m < E_{max}$ and $0 \leq (M - L) \leq 1$ & $0 \leq H - M \leq 1$. We call the E at this point E_{coarse}.*

Algorithm 3.2: Resumed coarse optimization algorithm given by Roy [9]

3.9 Fine optimization

Given the p by the step in Sect. 3.8, this step estimates the optimized value for each p by analyzing and reducing the p value for each variable. The Algorithm 3.3 resumes

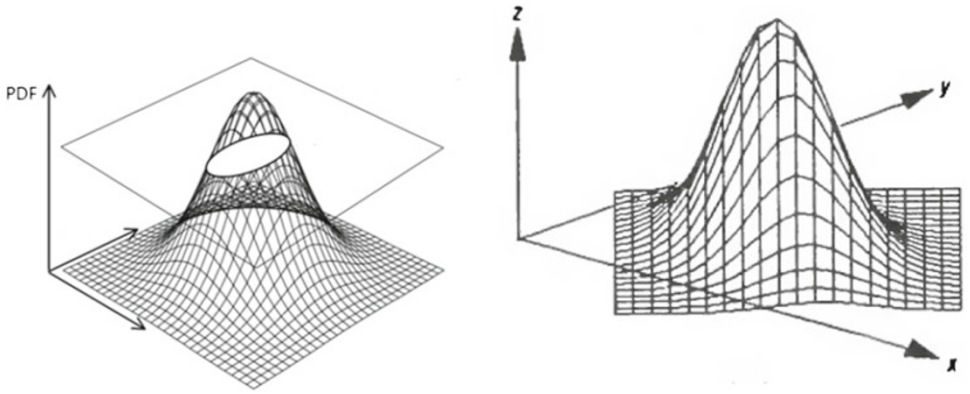


Fig. 2 The figures show two generic PDF where we can see that, given a probability, we can find a plane that intercepts the PDF forming an ellipse

the algorithm presented in Roy and Banerjee [9] and adds one more step in it. The first step in Algorithm 3.3 uses the vector D to set zero to the p values corresponding to integer variables, the following steps correspond to the algorithm given by Roy and Banerjee [9] resumed. Setting zero to some p values might impact significantly the *Fine Optimization* step, as can be seen in Sect. 5.

4 Analysis of the covariance matrix

The covariance matrix Σ encompasses the errors generated by the system noise and by the conversion from floating to fixed point. The larger is the matrix elements, the greater is the errors in the measures [1]. Each pair of elements of μ, which represents the feature and the robot locations, is associated to elements of Σ. These Σ elements define an ellipse in a two-dimensional space representing the area where the real feature should be located, given a fixed probability as shown in Fig. 2.

- *Set* $p = p. * not(In)$.
- *Set* $E_{fine} = E_{coarse}$.
- *Let* $E(i) = E$ *with* $p_i + 1$ *instead* p_i *for all quantizers* $q_1, q_2, ..., q_i, ..., q_n$.
- *Calculate* $DE(i) = E_{fine} - E(i)$, *for* $1 \leq i \leq n$.
- *Choose the smallest element (say index* $\boldsymbol{j}$*) of array DE for which* $p_j > 0$*; if* $p_i = 0$ *for all i, we terminate our algorithm.*
- *Calculate* $E(j) = E$ *with* $p_j - 1$ *instead* p_j*; if* $E(j) \leq E_{max}$ *then* $E_{fine} = E(j)$ *and set* $p_j = p_j - 1$ *and repeat the above three steps. If* $E(j) > E_{max}$ *we move to next step.*
- *Calculate DE using the same definition above.*
- *Find* $\boldsymbol{j}$ *for which DM(j) is maximum and the smallest element DE(*$\boldsymbol{k}$*) for which* $p_k > 1$*, if no such element is found we terminate the algorithm.*
- *Calculate* $E_{(j,k)} = E$ *with* $p_j + 1$ *and* $p_k - 2$ *instead* p_j, p_k.
- *If* $E_{(j,k)} < E_{max}$ *then* $E_{fine} = E_{(j,k)}$ *and* p_j, p_k *are changed to* $p_j + 1, p_k - 2$. *Repeat the above two steps. If* $E_{(j,k)} > E_{max}$ *we have found our* ***fine optimal quantizer values****.*

Algorithm 3.3: Adapted fine optimization algorithm. The first step sets zero for p values of integer variables defined by the vector D, from the second step forward, is the same algorithm adapted from Roy [9].

The conversion algorithm tries to define values for p such that the error, using a training set, remains lower than the error $E_{\max}$ given in step 3.2. If the chosen error is small, the p values will be bigger than necessary, on the other hand, if the chosen error is big, the system might diverge.

To define a better $E_{\max}$ value, we can analyze the covariance matrices since they incorporate the conversion errors. To compare the covariance matrices is easier than to define a metric that aggregate the information about the covariance ellipses in one curve, and then, compare two curves instead of a couple of matrices entries. Our idea is to observe the lengths of the semi-axis of the ellipses [12] by simply averaging the size of the semi-axis of each ellipse.

In this way, if any ellipse gets a large covariance component in one direction and a small component in another direction, our measure will have a considerable increase. This sensibility could not be reached if we would have considered other metrics, like area, this would not be interesting since the uncertainty about the feature position is actually high.

The ellipses axis are directly associated with elements of the main diagonal of the covariance matrix. This association is not linear, which means it has no direct meaning. Figure 3 represents a generic covariance ellipse of matrix (18) in $x \times y$ plane, the σ_{uu} and σ_{vv} are the values of the semi-axis of the ellipse.

$$\Sigma = \begin{bmatrix} \sigma_{xx}^2 & \sigma_{xy}^2 \\ \sigma_{xy}^2 & \sigma_{yy}^2 \end{bmatrix} \tag{18}$$

We can obtain the real values of the semi-axis of the ellipses, by the general law of covariances propagation [13], using the Eqs. 19 and 20.

$$\tan 2\alpha = \frac{2\sigma_{xy}^2}{\sigma_{xx}^2 - \sigma_{yy}^2} \tag{19}$$

$$\begin{cases} \sigma_{uu}^2 = \sigma_{xx}^2 \cos^2\alpha + \sigma_{yy}^2 \sin^2\alpha - 2\sigma_{xy}^2 \sin\alpha\cos\alpha \\ \sigma_{vv}^2 = \sigma_{xx}^2 \sin^2\alpha + \sigma_{yy}^2 \cos^2\alpha + 2\sigma_{xy}^2 \sin\alpha\cos\alpha \end{cases} \tag{20}$$

To exemplify the analysis, Fig. 4 shows some steps of an execution of our EKF-SLAM algorithm, and Fig. 5 shows the elements of Σ referents to the vehicle, and the size of

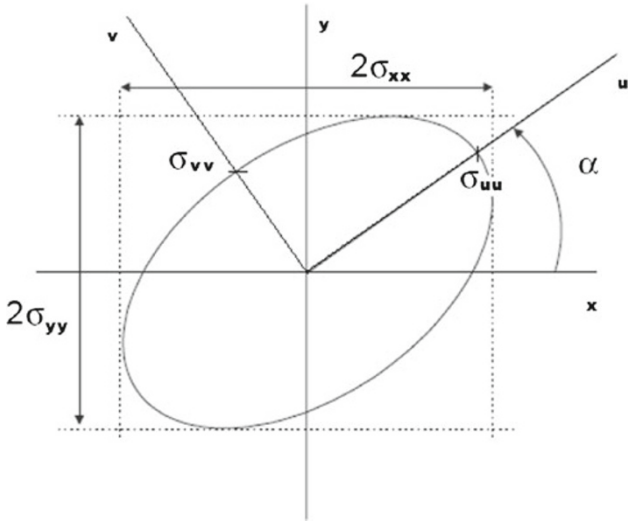


Fig. 3 Example of covariance ellipse representation in the plane $x \times y$ (Figure adapted from [10])

the semi-axis of vehicles covariance ellipse by the execution time (Fig. 5a, b, respectively). In Fig. 5, we can see the values of the main diagonal (of matrix Σ) elements for each feature and their average in time (Fig. 5c, d, respectively). Here, we can see that the values do not have a physical meaning, so, it is not a proper way for comparison (the values reach values near 300, which is bigger than the map dimension). Figure 5 also shows the values of the semi-axis of each diagonal and their average in time (Fig. 5e, f, respectively), where we can see that the covariance axis became stable between 2 and 3 units, for such configurations.

After the conversion is done, we are able to decrease p values and make another evaluation like the one shown in Fig. 5 and compare them to decide if the $E_{\max}$ chosen fits well. In our tests we decided to decrease one bit of each variable between comparisons to increase the error. Figure 6 shows the evaluations obtained. Observing the graphics in Fig. 6 a new error is chosen and the conversion algorithm is applied again to find new suitable values for m and p.

5 Preliminary results

In this section we present the main results of this work. The first one is the algorithm, which was adapted from Roy and

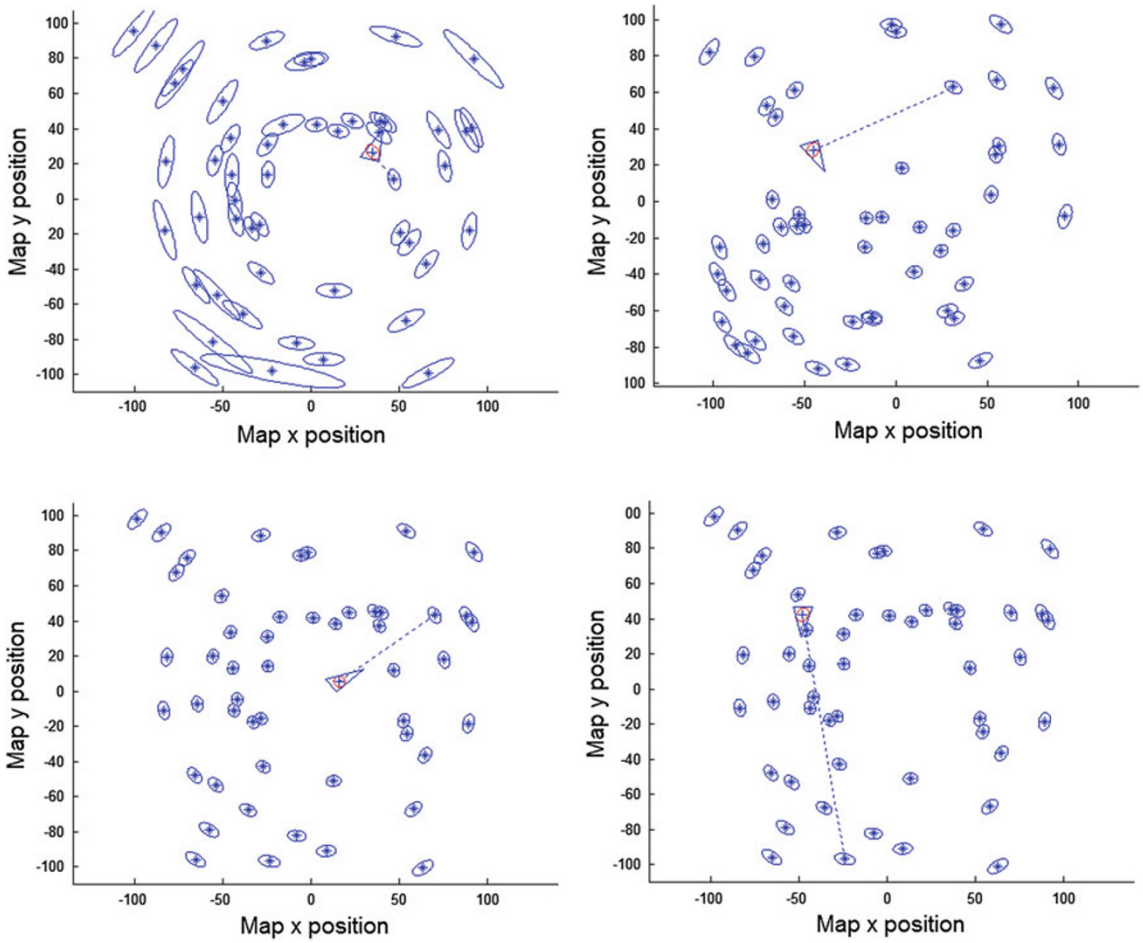
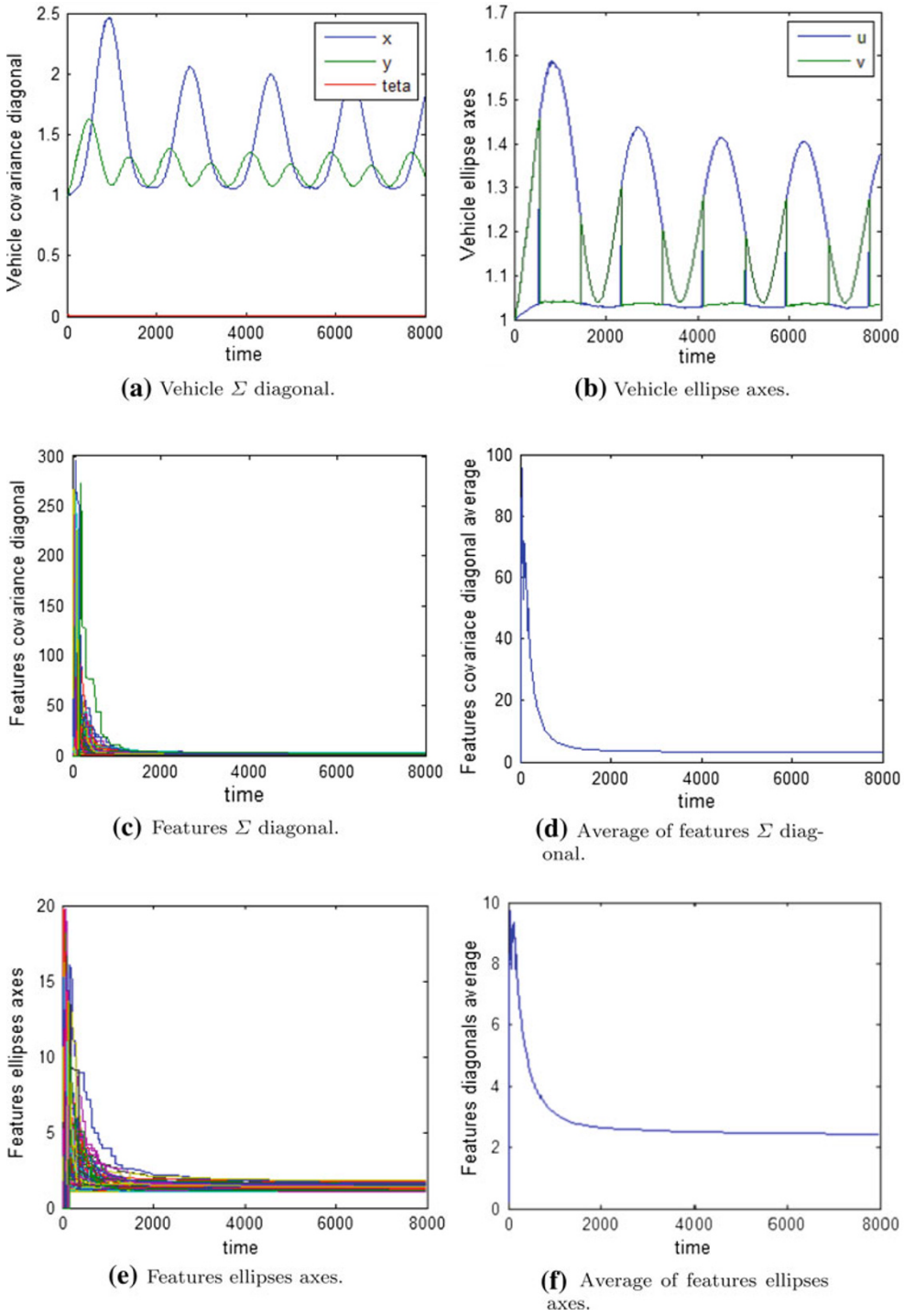


Fig. 4 Execution sequence for "full range" test in a two dimensional environment



(a) Vehicle Σ diagonal. **(b)** Vehicle ellipse axes.

(c) Features Σ diagonal. **(d)** Average of features Σ diagonal.

(e) Features ellipses axes. **(f)** Average of features ellipses axes.

Fig. 5 Covariance ellipses behavior for "full range" test

Banerjee [9] to convert the EKF-SLAM; the second one is the time that the conversion takes to finish; and the third one is the bit range for each of the EKF-SLAM variables. We used a simple configuration with a small training set in order to make the conversion faster. Real applications may need a bigger training set in order to have a better tuning for the bit range of the EKF variables.

5.1 Processing time

The error is calculated taking the maximum error between the execution of the floating-point code and the fixed-point code over the entire training set. Given x elements in the training set, for each error calculation the fixed-code and the floating-code are executed x times. If we say that the fixed-code takes

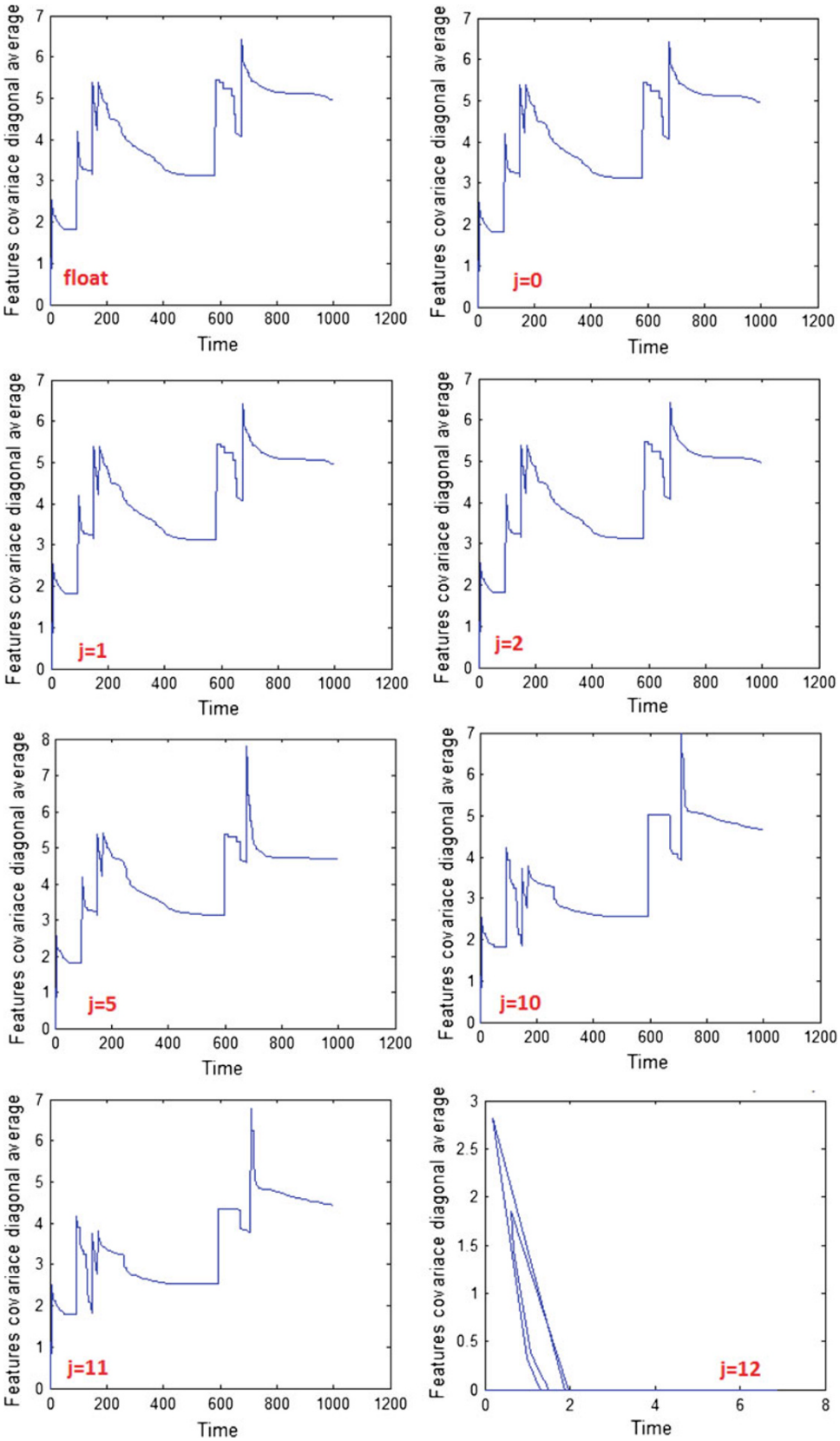


Fig. 6 Graphics of the average of the semi-axis sizes of the floating-point implementation, of the fixed-point implementation with $E_{\max} = 0.01\ \%$ and for the same fixed point implementation with j bits reduced

t_{fixed} and the floating t_{float} seconds, we have, approximately $t_{\text{error}} = x(t_{\text{fixed}} + t_{\text{float}})$ seconds for calculating the error.

The maximums values are calculated once running once the floating-code over the training set, so we can say that $t_{\text{maximums}} \approx x t_{\text{float}}$. We are going do disregard the time for calculating the p values because it is faster than the error calculation. Since the coarse optimization performs a binary search between [0, 32] we have $t_{\text{coarse}} = 3t_{\text{error}} \log_2 32 \approx 15t_{\text{error}}$.

The t_{maximuns} have no meaning in the original algorithm presented by Roy and Banerjee [9], the t_{error} can be very different in the original and in the modified conversions, which depends on the training set size. However, $t_{\text{coarse}} \approx 15t_{\text{error}}$ in both implementations.

Supposing that the coarse optimization gave a p_0 value for p, that we have n_i integer variables found in step (Sect. 3.5), $(0 \leq n_i \leq n)$, and that we have the final values of p. Then, the number of bits reduced in the fine optimization step is $np_0 - \sum_{i=1}^{n} p[i]$. For each bit reduced, the algorithm calculates the error n times, so, in the original algorithm we have $t^o_{\text{fine}} = n(np_0 - \sum_{i=1}^{n} p[i])t^o_{\text{error}}$. In the modified version we have that n_i variables have their p values set to zero instead of leaving it decreased to zero by the conversion algorithm. This saves $p_0.n_i$ to be reduced, therefore, in the modified algorithm we have $t^m_{\text{fine}} = n(p_0(n - n_i) - \sum_{i=1}^{n} p[i])t^m_{\text{error}}$ (here, "o" and "m" indicate the times referents to the original and modified algorithm, respectively).

We call by modified algorithm the algorithm with the *Evaluate Integer Variables* step (Sect. 3.5). It is worth noticing that the time saved by adding this step varies from algorithm to algorithm to be converted, the more integer variables, more time will be saved. Furthermore, since we had modified the original algorithm, it is impossible to compare the estimative time with it.

In our EKF-SLAM conversion we adopted $n = 107$, $n_i = 17$, $p_0 = 25$ and $\sum_{i=1}^{n} p[i] = 481$, which gave us $t^o_{\text{fine}} = 234758t^o_{\text{error}}$ and $t^m_{\text{fine}} = 189283t^m_{\text{error}}$ meaning that the usage of step 3.5 saved 19.37 % of the execution time of our fine optimization step.

5.2 Bit-range obtained

As mentioned in Sect. 4, the chosen $E_{\max}$ might not be the best value at first try. However, in order to evaluate our conversion algorithm $E_{\max} = 0.01$ % was chosen. Based on this error, the bit range of the variables was reduced interaction after interaction generating, as a result the graphics shown in Fig. 6. These graphics show the average size of the ellipses semi-axis of the float implementation, of the fixed implementation and of the fixed implementation when we reduced j bits from each value of p, which has different values in each position.

Table 4 The dimension, m_0, p_0 given by the conversion with $E_{\max} = 0.01$ % and m_f, p_f for $E^{\text{new}}_{\max} = 1.0$ % of the EKF symbols obtained, where s and r represent the feature and robot state size, v and f the robot and feature position, i the feature number and n the total number of features

Sym.	m_0	p_0	m_f	p_f
μ	12	22	12	22
μ_v	2	25	2	25
μ_f	12	25	12	25
Σ_{vv}	16	25	16	25
Σ_{vf}	16	25	16	25
Σ_{ff}	18	25	18	25
Σ	13	24	13	24
α	–	–	–	–
γ	–	–	–	–
u	2	25	2	25
F	5	25	5	25
G	5	25	5	25
Q	5	25	5	19
H_v	6	23	6	19
H_{fi}	6	25	6	19
H	6	25	6	19
R	15	25	15	19
W	10	24	10	19
v	11	25	11	19
z	12	25	12	19
z_{pred}	12	25	12	19
S	15	25	15	19
Z_1	–	–	–	–
Z_2	–	–	–	–

In Fig. 6, we can see that the shape of the curve changes when $j = 5$. When $j = 10$, we already can see several differences, and when $j = 12$ the fixed implementation diverges. To choose a better $E_{\max}$, say $E^{\text{new}}_{\max}$, we can compare the graphics from Fig. 6. Since $j = 2$ gave us a small difference when comparing with the float graphic, and its error was 1.17 %, we choose as new error $E^{\text{new}}_{\max} = 1.00$ %.

Table 4 shows the bit ranges of the main variables obtained for our EKF-SLAM implementation that were presented in Table 1. In Table 4 the m_0 and p_0 are the resulting bits for $E_{\max} = 0.01$ % and the m_f and p_f are the values for $E^{\text{new}}_{\max} = 1.0$ %. In Table 4, α and γ do not have m and p values associated because they are functions, and neither the zeros matrices Z_1 and Z_2 have, because they are not variables.

6 Conclusion

The paper has presented a method to convert the EKF algorithm from floating-point to fixed-point representation. The method demonstrates a way to measure the EKF computation error, which is crucial for guiding the conversion process. As a final result, a table is presented demonstrating the integer and fractional bit range needed for each EKF variable.

The EKF algorithm is widely used in mobile robotics to solve the problem of navigation and localization. The

fixed-point version presented in this paper might have a significant impact on embedded mobile robotic applications, since the hardware complexity needed to operate over fixed-point data is simpler and faster than any floating-point processing unit. As a consequence, the computation systems can be very optimized in relation to energy consumption, performance, and cost.

Finally, the case study presented in this paper is an EKF for SLAM problem in a two-dimensional planar coordinate system. However, the proposed method can be applied to higher dimensions, since the conversion algorithm and the error measure procedure are independent of the dimensions of the environment representation.

Acknowledgments The authors would like to thank FAPESP (Ref. 2010/05508-4) for the financial support given to develop this research project.

References

1. Smith R, Self M, Cheeseman P (1990) Estimating uncertain spatial relationships in robotics. Auton Robot Vehicles 8:167–193
2. Thrun S, Burgard W, Fox D (2005) Probabilistic robotics. MIT Press, Cambridge
3. Fox D, Hightower J, Liao L, Schultz D, Borriello G (2003) Bayesian filters for location estimation. IEEE Pervasive Comput 2(3):24–33
4. Bonato V, Marques E, Constantinides GA (2009) A floating-point extended Kalman filter implementation for autonomous mobile robots. J VLSI Sig Proc 56:41–50
5. Xilinx (2006) AccelDSP synthesis tool floating-point to fixed-point conversion of MATLAB algorithms targeting FPGAs. Xilinx Inc.
6. Belanović P, Rupp M (2005) Automated floating-point to fixed-point conversion with the fixify environment. In: 16th International workshop on rapid system prototyping
7. Moyers M, Stevens D, Chouliaras V, Mulvaney D (2009) Implementation of a fixed-point FastSLAM 2.0 algorithm on a configurable and extensible VLIW processor. In: IEEE international conference on eletronics circuits and systems (ICECS), Tunisia
8. Mingas G et al (2011) An FPGA implementation of the SMG-SLAM algorithm. Microprocess Microsyst. doi:10.1016/j.micpro.2011.12.002
9. Roy S, Banerjee P (2004) An algorithm for converting floating-point computations to fixed-point in MATLAB based FPGA design. In: Annual ACM IEEE design automation conference proceedings of the 41st annual design automation, pp 484–487
10. Newman PM (2005) C4B—mobile robotics, version 2.0. Robotics Research Group, The Department of Engineering Science, University of Oxford. http://www.robots.ox.ac.uk/~pnewman/Teaching/C4CourseResources/C4BResources.html. Accessed Aug 2011
11. MathWorks (2010) Fixed-point toolbox TM user's guide. http://www.mathworks.com/help/pdf_doc/fixedpoint/FPTUG.pdf. Accessed Aug 2011
12. Stuart A, Ord JK (1994) Kendall's advanced theory of statistics, vol I, 6th edn. Hodder Arnold, New York
13. Wolf PR, Ghilani CD (1997) Adjustment computations—statistics and least squares in surveying and GIS, Chap 5, 3rd edn. Wiley, New York