Robot Movement Optimization with Using Localization Algorithms

Omid Panah, Amir Panh, Amin Panah and Abolfazl Akbari

Abstract: The majority of localization algorithms start at a known position and add internal movement data and external environment data to this position each cycle. If the robot is replaced or the sensor data quality is too low, these algorithms are usually not able to recover to a useful position estimation. Members of these so-called local approaches are the linear least squares estimator and the Kalman filter. Robots equipped with global localization algorithms like Markov localization and particle filter are able to localize themselves even under global uncertainty. This article focuses on local and global localization, static environments and passive approaches. Active approaches have to be discussed along with the decision making. To be able to cope with dynamic environments, map building is necessary. Both topics are not within the scope of this work.

Key words: Kalman Filter · LLSQ · Markov Localization · Particle Filter

INTRODUCTION

Localization, Estimate the location of the robot within the environment based on observations. These observations typically consist of a mixture of odometric information about the robot’s movements and information obtained from the robot’s proximity sensors or cameras. In first problems are presented which may occur while performing a localization task. After introducing a taxonomy for localization algorithms, four different types of them linear least squares filter, Kalman filter, Markov localization and particle filter are discussed. They are ordered by their complexity and by historic development. First the simplest filter, the linear least squares filter, is described. Afterwards the Kalman filter and its derivates are shown. Next, Markov localization is presented. Finally, the particle filter, which is based upon the principles of the Markov localization, is explained. It is noteworthy to mention that the Kalman filter can be used within a Markov localization or particle filter algorithm as an extension to get better results [2]. After all algorithms are discussed, they are compared via a simulation example.

Linear Least Squares Filter: The linear least squares filter (LLSQ) is a simple state estimator. Based upon an observation history, outliers are muted. The dimensions of the state are not correlated-hence, the x and y values of a position are treated independently resulting in the fact that a faulty value in one dimension has no influence on the trustworthiness of its counterpart in the other dimension. The advantage of the LLSQ for localization is that the interval between the positions does not have to be constant. This filter can only be used for local localization.

The position of a robot is represented by the state vector s (1.1). It consists of the two values for x and y and the direction \( \varphi \).

\[
S = [x \quad y \quad \varphi]
\]  

(1.1)

For non-deterministic moving objects like robots, second order polynomial equations are used to calculate the elements of the state vector at the time t (1.2, 1.3, 1.4). For the ball and other non-self steered objects, a polynomial equation of first order can be used (e.g., \( x = a + bt \)).

\[
x = a + bt + ct^2
\]  

(1.2)

\[
y = d + et + ft^2
\]  

(1.3)

\[
\varphi = g + ht
\]  

(1.4)

Equation (1.5) can be used to calculate the three elements a, b, c. Looking at the matrix notation (1.6) for this equation, the parts of the equation can be identified as the time matrix A, the value history px for x and the result vector rx containing a, b and c.
\[ A r_\phi = p_\phi \]  
(1.5)

\[
\begin{bmatrix}
1 \\
1 \\
\vdots \\
1 \\
\end{bmatrix}
\begin{bmatrix}
t_0 \\
t_1 \\
\vdots \\
t_{n-1}
\end{bmatrix}
= \begin{bmatrix} a \\
b \\
\vdots \\
c
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
\vdots \\
x_{n-1}
\end{bmatrix}
\]
(1.6)

\[ r_\phi = (A^T | A)^T A^T p_\phi \]  
(1.7)

The transformed equation to calculate \( r_\phi \) can be adapted to calculate the result vectors for \( y \) and \( \varphi \) by replacing the history-vector. The same time matrix \( A \) is used to calculate all three result vectors. The length \( n \) of the history defines the adaption rate of the estimation to new circumstances. The resulting fills the data matrix with the incoming sensor updates in a FIFO manner [1]. As soon as enough sensor/position updates have been collected, the position estimation can be calculated. The old values are shifted by one position, resulting in the removal of the oldest ones. If the array is filled and thereafter the number of iterations \( j \) is equal or larger than the size of the array \( n \), the state can be estimated. The result vector \( r_\phi \) is determined using the matrix transformation described in (1.7). The components of \( r_\phi \) are then needed for the state estimation of \( r_\phi \).

**Kalman Filter:** The Kalman filter (KF) or discrete Kalman filter is a recursive state estimator, which can deal with incomplete and noisy data. The algorithm is divided into two update phases: (a) measurement update and (b) prediction update [3]. In (a), new sensor values are used to refine the prediction, whereas (b) predicts the new state estimation using the refined prediction of phase (a).

Other than the linear least squares filter, which basically uses the mean of the last \( n \) observations to determine the new state, the Kalman filter generates and adapts a world transition model [4]. Over time, its state estimations are getting more robust against measurement outliers. Traditionally, the KF is used for local localization problems.

**Kalman Model:** The model underlying the KF (Figure 1.1) is divided into a visible and a hidden part. The visible part consists of the system input and the sensor readings whereas the hidden part contains the state vector, the transition models, etc. As can be seen, the estimated state at time \( k \) is based upon the previous state estimation \( x_{k-1} \) and the system input or control vector \( u_k \). The sensor observation vector \( y_k \) is derived out of the current state \( x_k \).

![Fig. 1.1: Model Underlying the Kalman Filter](image)

Equations (1.8) and (1.9) are describing Figure 1.1. The first one defines the recursive state estimator \( x_k \in \mathbb{R}^n \), where \( n \) is the dimension of the state vector and \( k \) is the time index representing the time in multiples of \( \Delta t \). It is a combination of the transition of the last state, the system input and noise. The transition from one state to the next is defined in the matrix \( A \), which has a dimension of \( n \times n \). Next, to determine the relation between the system input \( u_k \) and the state, the matrix \( B \) with a dimension of \( n \times o \) is used. The constant \( o \) gives the dimension of \( u_k \), where \( u \in \mathbb{R}^o \). The system input \( u_k \) represents for example motion commands. Finally, to model uncertainty, a system noise \( w_k \) is added [5]. The noise is white and with normal probability distribution \( p(w) \sim \mathcal{N}(0,Q) \). Equation (1.9) gives the relation between the sensor observation vector \( y_k \ (y_k \in \mathbb{R}^m \) , where \( m \) is the dimension of the sensor vector) and the state vector \( x_k \). This relation is stored in \( C \). This matrix has a dimension of \( m \times n \). Similarly to \( w_k \), the variable \( v_k \) adds system noise to this relation. This noise is independent of \( w \), but also white and with normal probability distribution \( p(v) \sim \mathcal{N}(0,R) \).

The matrices \( Q \) and \( R \) are the process and the measurement covariance matrices. Next to their task to add system noise, they can be used to define a priori known relations between e.g. sensor readings. Their dimensions are \( n \times n \) for \( Q \) and \( m \times m \) for \( R \).

**Kalman Iteration Process:** This Process is divided into the “Time Update” or prediction phase and the “Measurement Update” or correction phase which are processed periodically (see Figure 1.2). In the first phase, the state is predicted upon the current internal state of the filter. The second phase corrects this prediction using the noisy measurements.
For prediction of the estimation $\hat{x}$ of the state vector $x$, only two components of the original equation (compare with Equation (1.8)) are used: The state transition and the system input. The intermediate error covariance matrix $p$ with the dimension $n \times n$ describes the estimated accuracy of the state estimation [6]. In Equation (1.11), the trustworthiness of the prediction is lowered by the process covariance matrix $Q$.

The Kalman gain $K$ describes the proportion between how much the predicted state can be trusted and up to which extent the new measurements have to be integrated. Ideally, the predicted state is accurate enough that the noisy sensor data is not needed for precise state estimation. $p$ is updated with the Kalman gain in Equation resulting in the error covariance matrix $P$. Thus, $K$ and $P$ are recursively influencing each other. $I$ is the identity matrix with dimension $n \times n$. Finally, the new state $x$ is obtained by adding to the predicted state $\hat{x}$ the deviation of the actual and the predicted measurement in proportion to the Kalman gain.

**Non-Linear Kalman Filters:** The EKF replaces the linear transition model $Ax_{k-1} + Bu_{k}$ and the linear measurement model $Cx_{k}$ with the non-linear functions $g$ and $h$.

Thus, the resulting belief of the estimation is no longer represented by a gaussian distribution.

$$x_{k} = g(u_{k}, x_{k-1}) + w_{k}$$

$$y_{k} = h(x_{k}) + v_{k}$$

The non-linearization of the matrices $A$, $B$ and $C$ is done using Taylor approximation and Jacobians [7]. The matrices $A$ and $B$ are represented by the Jacobian $G_{k}$ and the matrix $C$ by $H_{k}$.

**Markov Localization:** The linear least squares filter and the Kalman filter are only able to represent one state estimation. If e.g. based upon the current sensor readings the robot can be at two different positions but not in between, other approaches have to be used. An intuitive approach is to use topological or grid maps of the environment. For each possible position, the possibility of the robot to be there is calculated based upon the previous distribution and the actual sensor readings. Finally, the area with the highest belief is selected as estimated position. In case of global uncertainty—several not connected areas; each area with a high possibility—the robot has to continue moving until the possibility for all but one area have decreased and, therefore, only one possible estimation of the robot position remains. Like the Kalman filter, the Markov localization (ML) is based upon the Markov assumption. Thus, all past information is already inside the model through recursively applying the algorithm and sensor data on one set of locations.

$$Bel(L_{k} = i) = P(L_{k} = i)$$

$$P(L_{k} = i | L_{k-1}, a_{k-1}) = \sum_{a} P(L_{k} = i | L_{k-1}, a_{k-1}) \text{Bel}(L_{k-1} = i)$$

$$Bel(L_{k} = i) = P(s | t) P(L_{k-1} = i)$$

$$Bel(L_{k} = i) = \frac{\text{Bel}(s | t) P(L_{k-1} = i)}{P(s | a_{k-1})}$$

all locations have a uniform belief (Equation (1.17)). In case the starting position is known, the initialization of the localization beliefs can be adopted to this a priori knowledge. In each cycle, the location probabilities are updated. This is not only done using sensor readings but also—as an extension to the intuitive approach—actions executed by the system. Equation (1.18) applies the actions. E.g. if the robot moves one step right, all position estimations can be moved accordingly. After the sensor inputs have been applied (Equation (1.19)), all beliefs have to be normalized (Equation (1.20)). The sum of the beliefs for each location has to be The normalization step can be skipped if no new sensor inputs are available.

For each element that has to be considered for localization, another dimension is added to the state space. Thus, for the typical case of $x$, $y$ and $\varphi$ and grid representation of the locations, we have a three dimensional grid (see Figure 1.3).

Even with optimization techniques, the time needed to process this large array is problematic. As mentioned above, the space can be reduced using topological maps, but this may not be possible for all applications. Furthermore, even topological maps can get huge.
Fig. 1.3: Three-Dimensional Grid for Markov Localization

This localization method can easily be used not only to observe the self position, but also to observe several moving entities like the ball and other robots. Each area which is rated with a high belief represents at least one of these entities. Thus, it is a powerful tool for sensor data integration.

**Particle Filter:** Like mentioned above, the Markov localization can only be optimized to some extent. For larger environments, it is certain that either the precision or the calculation time is getting problematic. To overcome this, the number of cells has to be reduced without deterioration of the desired precision. The Particle filter or Monte Carlo Localization (MCL) replaces the dense grid representation by a much lower number of particles. These particles are state vectors representing the belief that the true location is exactly here.

The basic concept for the algorithm is a Monte Carlo approximation with Sequential Importance Sampling (SIS). Initially, random particles are generated and their weight is set to the reciprocal value of the number of particles. In each cycle, the weight of a particle is calculated by multiplying its old weight with the probability that this particle represents the real state regarding the newest sensor readings. After normalization of the weights, the real state can be estimated using them.

The SIS step multiplying the old weight with the new probability leads to degeneration. After several cycles, particles with a low probability are converging towards zero. Even if the sensor readings suggest that one of these particles should increase its weight, the previously strong particles will override this and stay strongly weighted. Thus, once a degree of certainty of the estimation has been reached, the model cannot change its estimation any more.

Here, the weight of each particle is determined each round independently from the previous one. After estimation of the state, a new set of particles is drawn from the actual set. This new set is used as basis for the next cycle. The particles are drawn in such a way that more important ones are more likely to be picked. Thus, areas with a high probability are gaining more weight by replicating their particles, whereas areas with a low probability are thinning out. Once an area has no particle left, another source for degeneration displays due to the lack of appropriate particles, local maxima will dominate the estimation.

During initialization, all of the N particles are generated by distributing them uniformly over the state space (Equation (1.21)). At the beginning of every cycle, the cycle index \( k \) is increased and temporary particles \( \tilde{x}_i \) are generated out of the state space \( x_i \) under the conditions of the particle from the previous cycle \( \tilde{x}_{i-1}(i) \) and the command control \( u_i \). Using the actual sensor data \( y_{in} \), the weight \( \tilde{w}_i(i) \) for the particles is calculated. The normalized weights are used to calculate the state estimation for \( x_i \) under the condition of all previous sensor readings \( y_{1:k} \). As preparation for the resampling step, all tuples \( \{\tilde{x}_i(i), \tilde{w}_i(i)\} \) have to be sorted in descending order with the highest weight-value first. During resampling, N times a random number \( j \) is drawn in such:

\[
x_n(i) \sim p(x_n) \quad (1.21)
\]

\[
x_i(i) \sim p(x_i | u_i, x_{i-1}(i)) \quad (1.22)
\]

\[
\tilde{w}_i(i) = p(y_i | \tilde{x}_i(i)) \quad (1.23)
\]

\[
\tilde{w}_i(i) = \tilde{w}_i(i) \left[ \frac{\tilde{w}_i(i)}{\sum_{j=1}^{N} \tilde{w}_j(j)} \right]^{-1} \quad (1.24)
\]

\[
E(g(x_i | y_{i:k})) = \sum_{j=1}^{N} g(\tilde{x}_j(j)) \tilde{w}_j(j) \quad (1.25)
\]

\[
\left\{ \frac{1}{N} \right\} = \{\tilde{x}_i(j), \tilde{w}_i(j)\} \quad (1.26)
\]

a way that tuples with a higher weight are preferred. For each \( j \), the corresponding \( \tilde{x}_i(i) \) is drawn and assigned to the final state \( x_i(i) \). All final state vectors are assigned with a uniform weight. To avoid local maxima, finally, \( T \) particles are replaced with newly generated uniformly distributed ones. \( T \) is much smaller than \( N \), typically 100 times and more.

**Comparison Experiment:** To compare all four localization algorithms handled in a simulation has been carried out during this work. The aim of this simulation is to track a single object, which is observed by a non-moving single vision system, which is mounted above the area. The object to be tracked is a robot, which travels at a constant speed in total two meters.

The outputs of the vision system are the absolute position coordinates \( x \) and \( y \), but no direction \( \varphi \). The standard deviation of the perceived position is six
centimeters from the real position. The robot moves at constant speed and during the simulation run, 100 data points are collected. Thus, the robot travels 2 cm per time frame. The localization algorithms have to estimate the real robot position using only the few unreliable sensor readings. Additional data like odometry, multiple objects and motion control have been left out intentionally.

**Configuration:** The following list gives only information about the parameters and optional elements for each algorithm. Linear Least Squares Algorithm Has a queue length of 5 steps. It uses a polynomial equation of second order to approximate the position.

**Kalman Filter:** The matrices are initialized as follows:

\[
A = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix},
B = \begin{bmatrix}
0 \\
0 \\
0 \\
\end{bmatrix},
C = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
\end{bmatrix},
P_0 = \begin{bmatrix}
3000 & 0 & 0 \\
0 & 3000 & 0 \\
0 & 0 & 3000 \\
\end{bmatrix},
Q = \begin{bmatrix}
0.14 & 0 \\
0 & 0.14 \\
\end{bmatrix},
R = \begin{bmatrix}
3600 & 0 & 0 \\
0 & 3600 & 0 \\
0 & 0 & 3600 \\
\end{bmatrix},
\]

**Markov Localization:** The grid is configured to have a cell size of 5×5 cm². As action update, the probability for the vicinity of the estimated position is increased. This is done applying a multi-variant distribution with \(\sigma\) much larger than the sensor error.

**Particle Filter:** 1000 particles are used. Further, in each cycle, 10 particles are reinjected. No action update is used.

**Results and Interpretation:** Figure 1.4 and Table 1.1 show error statistics for the path estimations depicted in Figure 1.5. The path given by the sensor (Figure 1.5(b)) is perceived by the vision system using the real path of the robot (Figure 1.5 (a)). Sensor data is also used as a benchmark for the localization algorithms-only if such a localization is on average better than the perceived sensor data, the effort is worth while.

![Fig. 1.4: Distance Between the Real Position and the Estimated Positions](image1)

![Fig. 1.5: Comparison of Localization Algorithms](image2)

Although the estimated path of the linear least squares algorithm moves sometimes back and forth (Figure 1.5 (c)), the overall performance is reasonably well
Table 1.1: Statistics for the comparison Experiment

<table>
<thead>
<tr>
<th></th>
<th>Max. Error</th>
<th>Min. Error</th>
<th>Avg. Error</th>
<th>better than Sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor data</td>
<td>176.6</td>
<td>3.6</td>
<td>68.6</td>
<td>0%</td>
</tr>
<tr>
<td>LLSQ</td>
<td>100.1</td>
<td>9.5</td>
<td>51.1</td>
<td>67%</td>
</tr>
<tr>
<td>KF</td>
<td>126.6</td>
<td>13.2</td>
<td>57.7</td>
<td>54%</td>
</tr>
<tr>
<td>ML</td>
<td>189.9</td>
<td>28.9</td>
<td>110.8</td>
<td>22%</td>
</tr>
<tr>
<td>MCL</td>
<td>153.6</td>
<td>5.5</td>
<td>51.8</td>
<td>76%</td>
</tr>
</tbody>
</table>

Table 1.2: Comparison of the Localization Algorithms

<table>
<thead>
<tr>
<th>Measurements</th>
<th>LLSQ</th>
<th>KF</th>
<th>EKF</th>
<th>ML</th>
<th>MCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Noise</td>
<td>landmarks</td>
<td>landmarks</td>
<td>landmarks</td>
<td>raw</td>
<td>raw</td>
</tr>
<tr>
<td>Optional sensor inputs</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Global Localization</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Multi-Hypothesis</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Efficiency (time)</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Efficiency (memory)</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Resolution</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Robustness</td>
<td>-</td>
<td>++</td>
<td>++</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Flexibility</td>
<td>++</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

good. It is second best compared to the benchmark and in the category of the average distance to the real position it is the best [8]. Also, the maximum error is the lowest in the test.

The Kalman filter follows the real path even with these faulty sensor readings (Figure 1.5(d)). The delayed adaption to course changes is systematic. To overcome it, additional information like motion commands is needed to apply the EKF. According to the table, the KF is second worst in the overall performance, only outperforming the Markov localization. But if the shape of the path is more important than the error distance to the real path, the KF is the best choice.

The worst result in this test is the one of the Markov localization. While Figure 1.5(e) suggests the same result interpretations like for KF, the error distances in Figure 1 are showing that the estimated path is on average twice as bad as the other estimations.

Finally, the particle filter produces similar results as the LLSQ. If the state vector is extended by the direction $\varphi$, the MCL and the other two algorithms would outperform LLSQ. In LLSQ, each dimension is treated independently, while the other algorithms are putting them into relation [9]. Thus, if there is a main movement direction, sensor readings, which suddenly change the direction to the opposite are more likely to be discarded. Because no code optimizations were done, timings of the different algorithms are not shown here. However, it can generally be said that the LLSQ is the fastest, shortly followed by the KF. The MCL is slower by a power of ten but this number varies for different numbers of particles. The slowest algorithm is the ML.

**Summary:** The simulation experiment presented in Section 6 showed that there cannot be such a thing as a general optimal localization algorithm. The choice for the best algorithm for a given application depends on several parameters, like available performance, type of objects to be tracked, robustness against outliers, or flexibility towards course changes. Table 1.2 gives an overview of the strengths and weaknesses of the different algorithms. It is based.

The first five rows are showing clearly that the ability to deal with multiple hypotheses is strongly related with the ability to handle raw data with any type of noise. Further, multiple hypotheses are needed for global localization. Optional sensor data is necessary for the case that the vision system needs 5 to 10 cycles to generate new landmarks. If the algorithm demands all sensor data in every round, this either leads to a decrease of cycles to fit the worst case, or the landmarks from the vision system have to be approximated if no new data is available.

The next three rows are about efficiency. LLSQ, KF and EKF are more efficient than ML and MCL—they only store a small state space. ML with the largest state space also takes longest and the precision is always lower bounded to the cell size. In the last two rows, robustness and flexibility are shown. Here, the EKF is at its best. KF and LLSQ are serving either the robustness or the flexibility, while ML has problems serving any of the two. In this overview, MCL is always good. Thus, for most applications, it is the algorithm of choice.

**REFERENCE**


