Non-Uniform Grid Performance Using The Model Fitness Algorithm

ARGUS - Activity Recognition and Object Tracking Based on Multiple Models

PTDC/EEA-CRO/098550/2008

Technical Report N° 7297

Sérgio Copeto

June, 2011
Contents

1 MATLAB Implementation for Model Fitness Algorithm 2
   1.1 Experimental Results ................................. 5

2 Non Uniform Grid Generators 6
   2.1 Moving Nodes ....................................... 6
   2.2 Nodes over trajectories method .................... 8
   2.3 Cluster adaptation .................................. 9
   2.4 Density-based node positioning ...................... 10
   2.5 Error compensation method ......................... 11

3 Model Performance Using the Modelfit Algorithm 15
Chapter 1

MATLAB Implementation for Model Fitness Algorithm

In this chapter we show the computations required for the model fitness algorithm, useful to select between different estimated models, the one that best describes a set of trajectories.

The main idea is to compute the probability that a given trajectory had been generated using a given model parameters \( \theta \), i.e.:

\[
Pr \{ x_{1:L_s} \mid \theta \}
\]

where \( L_s \) is the trajectory length, and \( \theta = (T, B, S) \), which includes the set of vectors \( T = t_k \), the set of transition matrices \( B = \{B_{ij}\} \), \( i, j \in \{1,...,K\} \), and the covariance matrices \( S = \{\Sigma_1,...,\Sigma_K\} \) associated with each vector field \( K \). The probability described in eq. 1.1 can be computed using the forward part of the forward-backward algorithm, described below:

1. Forward initialization:

\[
\alpha_1(i) = Pr \{ x_1, k_1 = i \mid \theta \} = Pr \{ x_1 \mid k_1 = i, \theta \} Pr \{ k_1 = i \mid \theta \}
\] (1.2)

2. Induction step:

\[
\alpha_t(i) = Pr \{ x_t \mid x_{t-1}, k_t = i, \theta \} Pr \{ x_{1:t-1}, k_t = i \mid \theta \}
\]

\[
= Pr \{ x_t \mid x_{t-1}, k_t = i, \theta \} \sum_{j=1}^{K} Pr \{ k_t = i \mid k_{t-1} = j, x_{t-1}, \theta \}
\]

\[
= N(x_t | x_{t-1} + T_i(x_{t-1}), \Sigma_i) \sum_{j=1}^{K} b_{ij}(x_{t-1}) \alpha_{t-1}(j)
\] (1.3)

3. Fitness value computation:
\[ \alpha = \sum_{j=1}^{K} \alpha_t(j) \quad (1.4) \]

The multivariate Gaussian distribution is:

\[ \mathcal{N}(x_t^n|x_{t-1}^n, \Sigma_k) = \frac{1}{\sqrt{(2\pi)^D|\Sigma_k^{-1}|}} e^{-\frac{1}{2}||x_t^n-x_{t-1}^n-T_k(x_{t-1}^n)||^2_{\Sigma_k^{-1}}} \quad (1.5) \]

Since the covariance matrices can have small values, a given trajectory with a large number of points can lead to a numerical overflow, due to the fact that the values obtained in the induction step grows exponentially in each iteration. Although the covariance matrices can’t be discarded (because each field \( K \) can have different covariance values), it’s still possible to avoid numerical overflow making the following variable substitution. Let \( y_t^n \) be:

\[ y_t^n = S_k[x_t^n-x_{t-1}^n-T_k(x_{t-1}^n)] = S_k e_t^n \quad (1.6) \]

where \( S_k \) represents a matrix such that \( S_k^T S_k = \Sigma_k^{-1} \). This matrices can be obtained though Cholesky factorization of \( \Sigma_k^{-1} \).

Making this substitution, the new covariance for the distribution becomes:

\[
E_k[y_t^n y_t^{nT}] = E_k[S_k e_t^n e_t^{nT} S_k^T] \\
= S_k E[e_t^n e_t^{nT}] S_k^T \\
= S_k \Sigma S_k^T \\
= S_k (S_k^T S_k)^{-1} S_k^T \\
= S_k S_k^{-1} S_k^{-T} S_k^T \\
= I
\]

(1.7)

It’s important to note that since the probabilities are computed based on the error from the trajectory model, once the variable substitution is taken into account, the new probability distribution have zero mean and identity covariance matrix (as described in eq. 1.7)

\[ y_t^n \approx \mathcal{N}(0, I) \quad (1.8) \]

Another thing to take into consideration is that:

\[ |\Sigma_k| = |(S_k^T S_k)^{-1}| = \frac{1}{|S_k^T S_k|} = \frac{1}{|S_k^T||S_k|} = \frac{1}{|S_k|^2} \quad (1.9) \]

As such, the probability distribution for variable \( y_t^n \) can be computed as:

\[
\frac{1}{\sqrt{(2\pi)^D|\Sigma_k|}} e^{-\frac{1}{2}(y_t^n y_t^{nT})} = |S_k| \frac{1}{\sqrt{(2\pi)^D}} e^{-\frac{1}{2}(S_k^T y_t^n y_t^n)}
\]

(1.10)
The final step to avoid the numerical overflow is to compute $|S_k|$ for all vector fields and normalize the probability computed in eq. 1.10 according to $\max(|S_K|)$:

$$\frac{|S_k|}{\max(|S_K|)} \frac{1}{\sqrt{(2\pi)^D}} e^{-\frac{1}{2}(y_i^T y_i^*)}$$  \hspace{1cm} (1.11)

This way we assure that values computed in the induction step of the forward algorithm only propagates values between 0 and 1.

These steps are important because, in order to obtain the real probability value, it’s only needed to multiply the final value of $\alpha_k$ by:

$$\left(\max(|S_K|) \frac{1}{\sqrt{(2\pi)^D}}\right)^{L_S}$$  \hspace{1cm} (1.12)

where $L_S$ denotes the number of trajectory points. Since these values are constant, there’s no need to considerate them in the fitness computations.

The forward algorithm implemented for the model fitness algorithm is:

1. Forward initialization:

$$\alpha_i^1(k) = 1/K$$  \hspace{1cm} (1.13)

2. $|S_k|$ and $\max(|S_K|)$ computations:

for all $k$:

$$S_k = \text{Cholesky}(\Sigma_k^{-1})$$  \hspace{1cm} (1.14)

3. Find $\max(|S_K|)$

4. Compute $y_i^T(k)$ for all $k$:

$$y_i^T(k) = S_k[x_i^* - x_{i-1}^* - T_k(x_{i-1}^*)] = S_k e_i^*$$  \hspace{1cm} (1.15)

5. Compute probability for all trajectory points and every $k$:

$$\text{Prob}(y_i^T(k)) = \frac{|S_k|}{\max(|S_K|)} \frac{1}{\sqrt{(2\pi)^D}} e^{-\frac{1}{2}(y_i^T y_i^*)}$$  \hspace{1cm} (1.16)

6. Forward induction:

For all points of the trajectory:

$$\alpha_i^T(i) = \text{Prob}(y_i^T(k)) \sum_{j=1}^{K} b_{ji} \alpha_i^T_{i-1}(j)$$  \hspace{1cm} (1.17)

7. Fitness value computation:

$$\alpha^* = \sum_{j=1}^{K} \alpha_i^T(j)$$  \hspace{1cm} (1.18)
1.1 Experimental Results

Once the algorithm is implemented, it is convenient to test if it produces consistent results. For that, we will use a model generated with the implemented algorithms, $T_k$, described below:

For each vector field model $T_k$, we will compute the fitness to 4 trajectories $X^*$ with the same lengths $L$, two of them generated using the test model and other two random trajectories. The trajectories and correspondent fitness results are shown in figure 1.2:

The results show that there’s a significant difference in the fitness results between trajectories generated by the model (the two from the left) and the aleatory trajectories (the two from the right).

Although the fitness results for the test model generated trajectories appear low, it is still possible to establish comparison between different kinds of trajectories, and even between the same trajectories applied to different models.
Chapter 2

Non Uniform Grid Generators

Until now, trajectory models have been estimated using uniform interpolation grids. One important question is to know how good the uniform grid models represent the input trajectories for estimation, and if there are better options. This chapter shows some algorithms for non-uniform interpolation grid estimation, based on the input trajectory information.

2.1 Moving Nodes

The first algorithm for non-uniform grid creation starts with an uniform grid, as shown in 2.1, where all the trajectories are represented in blue and the generated uniform grid points in red circles. The main idea is to move interpolation nodes from regions where no data exists to regions with higher information.

![Uniform grid created based on the trajectories point cloud](image_url)

Figure 2.1: Uniform grid created based on the trajectories point cloud
The algorithm follows this procedure:

For all interpolation nodes $g_n$, $n = \{1, \ldots, N\}$:

1. Find the number of trajectory points near a certain node $g_n$, considering a radius $R$ as the search distance.

2. Compare the amount of information near the node with a defined threshold $TH$ and if it’s above this value, then the node may not be moved, so the iteration stops here.

3. If an interpolation node has to be moved, compute its new location $g'_n$, based on a given percentage $G$ of the distance to the closest trajectory point, i.e.:

   $$g'_n = g_n + G \cdot \arg \min_{x_i} \| g_n - x_i \|$$  \hspace{1cm} (2.1)

The result of this step, for all interpolation nodes, is illustrated in figure 2.2.

![Figure 2.2: Uniform grid created based on the trajectories point cloud](image)

Testing the algorithm with $R = 0.08$ and $G = 1$, the resulting grid is shown in figure 2.3.
2.2 Nodes over trajectories method

The second developed method tries to distribute the interpolation nodes randomly over the trajectories point cloud. The main problem comes when different locations have a different point density, resulting in a higher probability of not having sufficient interpolation nodes in low point density zones.

In this method, the trajectory points are treated as a point cloud. The steps are the following:

1. Convert all trajectory information into a point cloud:

   \[ P_i, \quad (i = 1, \ldots, I) \]  

   where \( P_i \) denotes a certain point \( i \) from the whole point cloud, being \( I \) the total number of trajectory points.

2. Interpolation nodes are selected according to this rule:

   \[ g_n = P\left( I \frac{n}{N} \right) \]  

   Where \( N \) is the total number of interpolation nodes.

One can keep the points unorganized, or organize them according to the distance to the cloud center. Testing the algorithm with ordered and unordered distances to the point cloud center, the resulting grids are shown in figures 2.4 and 2.5.
It’s possible to conclude that this method may not produce good results as there are some zones in the point cloud with little representation in terms of interpolation nodes.

### 2.3 Cluster adaptation

The cluster adaptation method tries to better distribute the interpolation nodes over the point cloud, not only the nodes in low point density zones, but also re-arranging the interpolation nodes in high point density zones. The process
starts with an uniform grid (see figure 2.1), and the interpolation nodes are updated this way:

1. Convert all trajectory information into a point cloud:
   \[ P_i, \quad (i = 1, \ldots, I) \]  \hspace{1cm} (2.4)

2. For each node \( g_n \), identify the points from the point cloud inside a radius \( R \) of the interpolation node.
   \[ C_n = \{ P_i : \| g_n - P_i \| \leq R, i = 1, \ldots, I \} \]  \hspace{1cm} (2.5)
   Where \( C_n \) is a vector containing the data inside a radius \( R \) neighborhood from \( g_n \)

3. Move node \( g_n \) based on the percentage \( G \) of the distance to the center of \( C_n \), i.e.:
   \[ g_n' = g_n + G \cdot \| g_n - \overline{C_n} \| \]  \hspace{1cm} (2.6)

Following this method, the resulting grid is illustrated in figure 2.6.

![Figure 2.6: Grid generated using cluster adaptation](image)

### 2.4 Density-based node positioning

It’s important to keep the interpolation nodes distribution as homogeneous as possible to avoid having underrepresented areas in the model estimation. However, one important question is how good a vector field will be if interpolation nodes are placed according to the amount of trajectory information, having more nodes where there is more trajectory information and a minor percentage of nodes in areas with less information.
In this method, the point cloud is divided into $A$ slots in order to place $A^D$ nodes, where $D$ denotes the space dimension. The algorithm is:

1. Compute the percentage of points inside each slot, $C_a$, so one can estimate the number of interpolation nodes to include in each area.

2. Uniformly distribute the same percentage of the total number of interpolation nodes, $C_a$.

3. Perform a cluster adaptation to all of the interpolation nodes

This way, nodes are distributed according to the density of information. As the last step, a cluster adaptation is used in order to better adapt the interpolation nodes to the trajectories.

As an example, the algorithm will be used to distribute 100 nodes over a set of 2D trajectories (dividing the trajectory space into 10 slots). The resulted grid is shown in figure 2.7.

![Figure 2.7: Grid generated using density-based node positioning](image)

### 2.5 Error compensation method

This algorithm created tries to make a grid adaptation based on it’s estimated model. Given the set of trajectories $X_s$, the interpolation nodes $g$ used for the model estimation, the resulting $K$ vector fields $T_k$ and the transition probabilities $B$:

1. Compute the error associated with each point of the trajectories. The error associated with the instant $t$ using vector field $k$ for trajectory $s$ is
given by:

$$e^*(t) = \frac{1}{K} \sum_{k=1}^{K} \| x^*(t) - x^*(t-1) - T_k \Phi(x^*(t)) \|^2 \quad (2.7)$$

The result of this process is an error map, as illustrated in figure 2.8. This information is useful to find the areas that could be better represented, where the overall error value is higher. Hence, the objective is to move some nodes so the global error is the lowest possible. Therefore, interpolation nodes have to be identified according to their position and utility, in order to select which ones have to be moved.

2. Identify the isolated nodes as their position is not relevant for the model estimation. Interpolation nodes are considered isolated if there are no trajectory points in a radius $R$ neighborhood. As an example, for $R = 0.05$, isolated nodes are identified in figure 2.9.
3. Compare the error value of each trajectory point $e^i(t)$ with a threshold $TH$, identifying the underrepresented regions.

$$E = \{ x^i(t) : e^i(t) \geq TH \}$$  \hspace{1cm} (2.8)

4. Search for the interpolation nodes laying in the identified areas. Once again, radius search is used, and a certain interpolation node is considered as a candidate to be moved if:

$$\exists x^i(t) \in E \| g_n - x^i(t) \| \leq R$$  \hspace{1cm} (2.9)

One possible problem is related to the high error area boundaries. A certain interpolation node can be a candidate, but if there are also trajectory points under the error threshold inside it’s radius $R$, a choice have to be done whether the node will be moved or not. The decision is made by comparing the error value for each near point. If a node has more trajectory points with high error value in relation to lower error, then that node is moved. Figure 2.10 represents all the nodes selected by the algorithm to use for a better representation of high error areas.
5. Perform a cluster adaptation to the grid, using just the nodes with high error values and the nodes identified as movable. Note that it’s also possible to perform a cluster adaptation to the low error areas. The grid obtained at the end of the whole process is illustrated in figure 2.11.

As this algorithm is used after a model estimation, it’s possible to make several iterations, in order to improve the estimated models.
Chapter 3

Model Performance Using the Modelfit Algorithm

This chapter establishes comparison between the performance of an uniform grid and all the previously presented algorithms for non-uniform interpolation grids. The tests follow this procedure:

1. Estimate the vector field model for all the interpolation grids presented in the previous chapter and also for an uniform interpolation grid.

2. Generate five trajectories of 500 points each for all the models.

3. Measure the performance of each model for the respective generated trajectories using the Model Fitness algorithm, described in the first chapter.

The fitness values for all the tests are shown in table 3.1.

<table>
<thead>
<tr>
<th>Grid Type</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>3.2523e-014</td>
<td>1.3635e-014</td>
<td>7.9309e-009</td>
<td>1.2104e-014</td>
<td>3.0176e-013</td>
</tr>
<tr>
<td>Moving Nodes</td>
<td>1.1258e-026</td>
<td>2.7034e-010</td>
<td>3.0495e-019</td>
<td>9.9732e-018</td>
<td>1.1618e-014</td>
</tr>
<tr>
<td>Cluster</td>
<td>2.2228e-012</td>
<td>1.0414e-010</td>
<td>4.0761e-012</td>
<td>1.0559e-006</td>
<td>1.2438e-007</td>
</tr>
<tr>
<td>Point Density</td>
<td>5.8456e-025</td>
<td>4.0307e-009</td>
<td>1.7402e-020</td>
<td>1.0871e-021</td>
<td>1.0364e-023</td>
</tr>
<tr>
<td>Error Compensation</td>
<td>3.6295e-012</td>
<td>6.1126e-007</td>
<td>5.7261e-012</td>
<td>2.043e-010</td>
<td>2.8939e-014</td>
</tr>
</tbody>
</table>

Comparing all the grid types, it’s possible to see that grids following the point density criteria (nodes over trajectories and point density grids) have worse performance than expected. These methods, although better represent areas where the information density is greater, continue to misrepresent areas with low density of information, which causes a high prediction error.

The moving nodes method, which moves nodes to the closest trajectory point, the performance is lower than using a regular grid. this may be caused
by the interpolation method used for the trajectory estimation.

The cluster adaptation and the error compensation methods are the only ones with better performance in relation to the uniform grid model. For the error compensation method, only one iteration was used, but as the algorithm identifies high error areas, it’s estimated models will perform better.