MATH-GA.2012 High Performance Computing
Expectation-Maximization Algorithm with Kalman Filter for a Linear Guassian Model

Yuan Chen Tao Chen Danilo Trinidad Pérez-Rivera

1 Introduction

In many scientific endeavors, observations are quantified by the taking of a particular measurement. The precision of a particular observation is often reported in relation to the level of uncertainty existent in the measurement of the observation, commonly referred to as the signal-to-noise ratio. A classical form of maximizing this signal-to-noise ratio has been the repeated measurements approach, whereby averaging multiple measurements, noise is minimized. However, this rudimentary approach depends on many factors, chiefly, the stationary nature of the signal allowing for the repeated measurements. For particularly well-behaved variables, non-stationary observations can be characterized by a linear dynamical system, in which the repeated observations are characterized by the underlying evolution of hidden states that determine these observations. Kalman Filtering arises in this context as the optimal solution that integrates any underlying knowledge about the evolution of these hidden processes, and the obtained measurements, given Gaussian noise. [1]

In the decades since the development of this approach, it has seen widespread application particularly in physical engineering problems when the underlying physical process provides clear relationships between the variables of interest. For example, most modern cellphones are equipped with an accelerometer, odometers and sensors that allow connection with satellite-based global positioning systems. Suppose we would be interesting in tracking the cellphones position, depending on any of these exclusively would be prone to error, such as drift introduced by accelerometer integration, or increased measurement error of GPS systems during passage through regions of reduced coverage. In this context, the Kalman Filtering framework optimally combines these measurements with appropriate consideration of their corresponding noises, producing the best possible estimate for the position. As can be seen in Figure 1, an ever-increasing number of scientific papers have been published utilizing Kalman Filters.

A less frequent application of Kalman Filters is possible, even when the underlying relationships are unknown. To do so, these relationships must be inferred utilizing statistical approaches, such as the Expectation Maximization algorithm. Expectation maximization is an iterative method that can be used to find local maximum likelihood estimates of these parameters. After being initialized at some reasonable approximation of what the true parameters may be, each iteration of the Expectation Maximization algorithm produces an updated parameter set that would is more likely to produce the observations that are obtained, based on the hidden states that were inferred.[3] Needless to say, this procedure is highly computationally intensive, for which reason

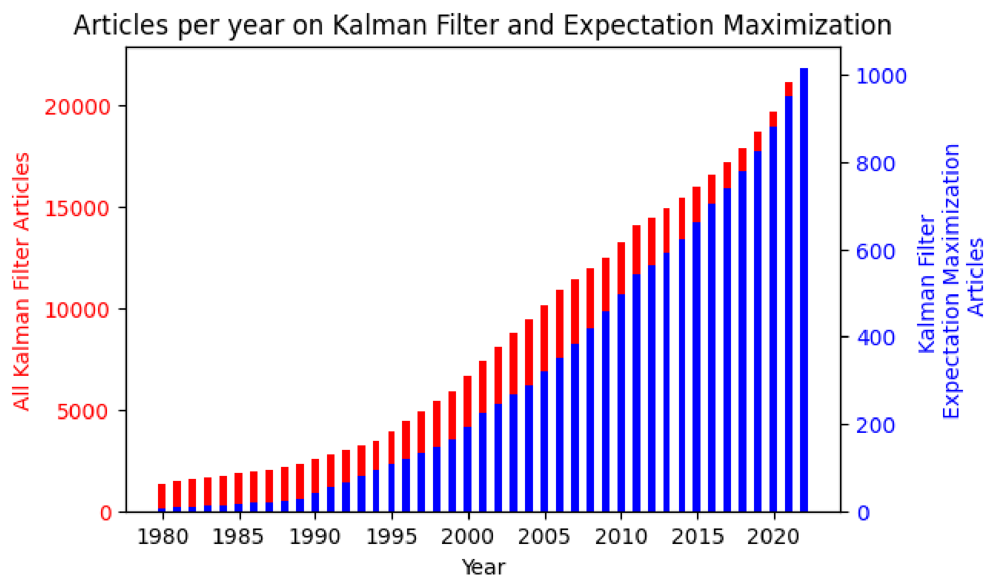


Figure 1: *Publications per year for all Kalman Filter (red) and Kalman Filters and Expectation Maximization (blue). Source: Google Scholar [2]*

historically it has observed limited application since its first description. Nonetheless, as can also be confirmed in Figure 1, the rate of growth of applications of this type has recently been outpacing even the general rate of growth of the Kalman Filter literature. This has been driven by the ever-increasing size of datasets produced by experimentalists interested in describing latent structures in their data, particularly for time series data such as neural recordings.

Given this heightened interest, we undertook the parallelization of this algorithm as our project. Referring to the ample literature describing the algorithm, we were able to identify some key computations we have learned to parallelize during the course including Matrix-Matrix Multiplication, Summation of point-wise Covariance Matrices and Matrix Inversion.

2 Algorithm and Parallelization

Figure 2 presents the pseudocodes of the whole algorithm. There are two functions, learning function and inference function. Learning is to obtain the parameters that generate the linear Guassian model given a trajectory of the observed states, while inference is to calculate the hidden states based on the observed states and the parameters. Inside the learning it invokes inference many times until the change in the log likelihood is small enough. The calculations of the algorithm are entirely matrix operations such as matrix multiplication and matrix inverse.

After coding the pesudocode into C++, we used Openmp for the parallelization. Omp for is added to all the matrix operations including matrix adding/subtraction, matrix multiplication, and matrix inverse. Apart from this, however, we cannot parallelize the three for loops in the learning and inference functions, because the next iteration depends on the values obtained in

```
LDSLearn($\mathbf{Y}$,$k$,$\epsilon$)
  initialize $\mathbf{A}$, $\mathbf{C}$, $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{x}_1^0$, $\mathbf{V}_1^0$
  set $\boldsymbol{\alpha} \leftarrow \sum_t \mathbf{y}_t \mathbf{y}_t'$

while change in log likelihood $> \epsilon$
  LDSInference($\mathbf{Y}$,$\mathbf{A}$,$\mathbf{C}$,$\mathbf{Q}$,$\mathbf{R}$,$\mathbf{x}_1^0$,$\mathbf{V}_1^0$)  % E
  initialize $\boldsymbol{\delta} \leftarrow \mathbf{0}$, $\boldsymbol{\gamma} \leftarrow \mathbf{0}$, $\boldsymbol{\beta} \leftarrow \mathbf{0}$
  for $t = 1$ to $T$
    $\boldsymbol{\delta} \leftarrow \boldsymbol{\delta} + \mathbf{y}_t \hat{\mathbf{x}}_t'$
    $\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma} + \hat{\mathbf{x}}_t \hat{\mathbf{x}}_t' + \hat{\mathbf{V}}_t$
    $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \hat{\mathbf{x}}_t \hat{\mathbf{x}}_{t-1}' + \hat{\mathbf{V}}_{t,t-1}$ if $t > 1$
  end
  $\boldsymbol{\gamma}_1 \leftarrow \boldsymbol{\gamma} - \hat{\mathbf{x}}_T \hat{\mathbf{x}}_T' - \hat{\mathbf{V}}_T$
  $\boldsymbol{\gamma}_2 \leftarrow \boldsymbol{\gamma} - \hat{\mathbf{x}}_1 \hat{\mathbf{x}}_1' - \hat{\mathbf{V}}_1$
  % M step
  $\mathbf{C} \leftarrow \boldsymbol{\delta} \boldsymbol{\gamma}^{-1}$
  $\mathbf{R} \leftarrow (\boldsymbol{\alpha} - \mathbf{C}\boldsymbol{\delta}')/T$
  $\mathbf{A} \leftarrow \boldsymbol{\beta} \boldsymbol{\gamma}_1^{-1}$
  $\mathbf{Q} \leftarrow (\boldsymbol{\gamma}_2 - \mathbf{A}\boldsymbol{\beta}')/(T-1)$
  $\mathbf{x}_1^0 \leftarrow \hat{\mathbf{x}}_1$
  $\mathbf{V}_1^0 \leftarrow \hat{\mathbf{V}}_1$
end
return $\mathbf{A}$, $\mathbf{C}$, $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{x}_1^0$, $\mathbf{V}_1^0$
```

```
LDSInference($\mathbf{Y}$,$\mathbf{A}$,$\mathbf{C}$,$\mathbf{Q}$,$\mathbf{R}$,$\mathbf{x}_1^0$,$\mathbf{V}_1^0$)  % Kalman smoother
  for $t = 1$ to $T$  % Kalman filter (forward pass)
    $\mathbf{x}_t^{t-1} \leftarrow \mathbf{A}\mathbf{x}_{t-1}^{t-1}$ if $t > 1$
    $\mathbf{V}_t^{t-1} \leftarrow \mathbf{A}\mathbf{V}_{t-1}^{t-1}\mathbf{A}' + \mathbf{Q}$ if $t > 1$
    $\mathbf{K}_t \leftarrow \mathbf{V}_t^{t-1}\mathbf{C}'(\mathbf{C}\mathbf{V}_t^{t-1}\mathbf{C}' + \mathbf{R})^{-1}$
    $\mathbf{x}_t^t \leftarrow \mathbf{x}_t^{t-1} + \mathbf{K}_t(\mathbf{y}_t - \mathbf{C}\mathbf{x}_t^{t-1})$
    $\mathbf{V}_t^t \leftarrow \mathbf{V}_t^{t-1} - \mathbf{K}_t\mathbf{C}\mathbf{V}_t^{t-1}$
  end
  initialize $\hat{\mathbf{V}}_{T,T-1} = (\mathbf{I} - \mathbf{K}_T\mathbf{C})\mathbf{A}\mathbf{V}_{T-1}^{T-1}$
  for $t = T$ to 2  % Rauch recursions (backward pass)
    $\mathbf{J}_{t-1} \leftarrow \mathbf{V}_{t-1}^{t-1}\mathbf{A}'(\mathbf{V}_t^{t-1})^{-1}$
    $\hat{\mathbf{x}}_{t-1} \leftarrow \mathbf{x}_{t-1}^{t-1} + \mathbf{J}_{t-1}(\hat{\mathbf{x}}_t - \mathbf{A}\mathbf{x}_{t-1}^{t-1})$
    $\hat{\mathbf{V}}_{t-1} \leftarrow \mathbf{V}_{t-1}^{t-1} + \mathbf{J}_{t-1}(\hat{\mathbf{V}}_t - \mathbf{V}_t^{t-1})\mathbf{J}_{t-1}'$
    $\hat{\mathbf{V}}_{t,t-1} \leftarrow \mathbf{V}_t^t\mathbf{J}_{t-1}' + \mathbf{J}_t(\hat{\mathbf{V}}_{t+1,t} - \mathbf{A}\mathbf{V}_t^t)\mathbf{J}_{t-1}'$ if $t < T$
  end
  return $\hat{\mathbf{x}}_t$, $\hat{\mathbf{V}}_t$, $\hat{\mathbf{V}}_{t,t-1}$ for all $t$
```

Figure 2: *Pseudocodes*

the previous iteration.

Initially, we didn't get any speed up after our parallelization. By testing the operations separately and individually, we found that the issue is on the matrix inverse. Gaussian elimination, which is used in matrix inverse, has three nested for loops and at its outermost loop it has a sequential dependence. Therefore we cannot parallelize the outermost for loop but instead to parallelize the middle loop, in this way we have to invoke OMP for N times, where N is the dimension of the matrix. Since the number of operations required inside the middle for loop is proportional to N^2, and it is rather time-consuming to invoke OMP for, it can cause severe overhead when N is small, e.g. when $N < 250$. One thing to notice is that matrix inverse has the highest time complexity among all the matrix operations we have, so it is no wonder that it can ruin the overall speed up if the overhead of matrix inverse is severe. To fix this, we didn't parallelize matrix inverse when the dimension of the data is small, only add OMP for when the dimension is large enough.

3 Experiments and results

All the experiments are under the system with Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz as CPU, totally 2 nodes, with 10 CPUs each node, and under the x86_64 architecture.

To verify the algorithm correctness, we tested the prediction before and after parallizations for hidden states x comparing with original hidden states. By the nature of Kalman Filter EM algorithm, it should be able to predict the x through the linear dynamical system. We tested the output with a small batch of data , where $dim_x = 5$ and a large dimension of data where $dim_x = 250$, as

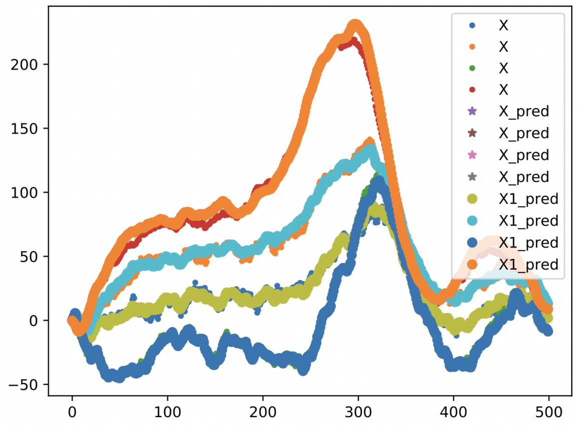


Figure 3: *dim_x = 5 predicted x vs original x*

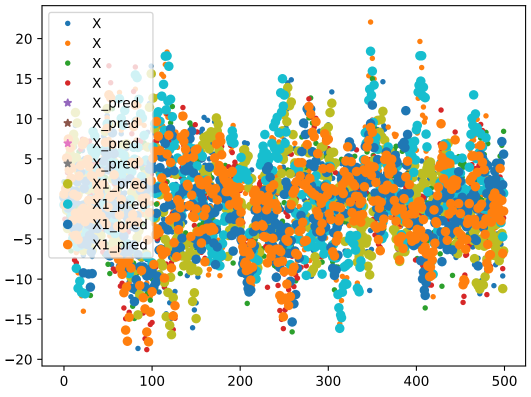
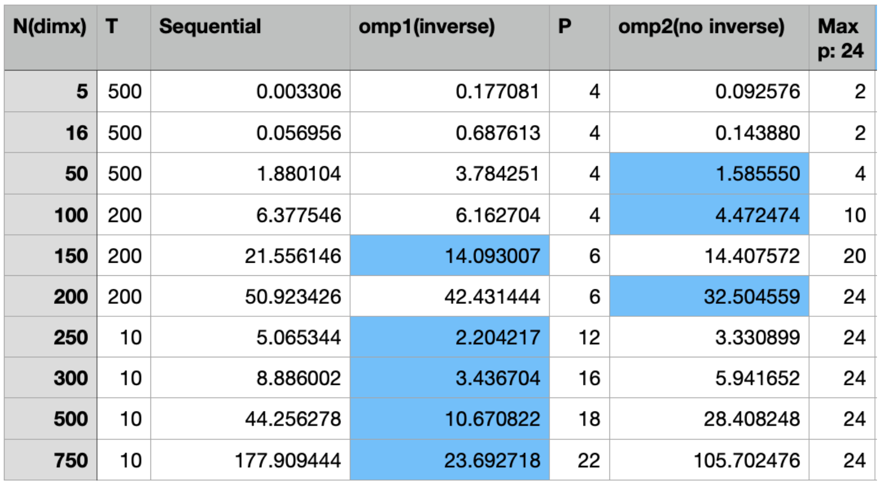


Figure 4: *dim_x = 250 predicted x vs original x*



N(dimx)	T	Sequential	omp1(inverse)	P	omp2(no inverse)	Max p: 24
5	500	0.003306	0.177081	4	0.092576	2
16	500	0.056956	0.687613	4	0.143880	2
50	500	1.880104	3.784251	4	1.585550	4
100	200	6.377546	6.162704	4	4.472474	10
150	200	21.556146	14.093007	6	14.407572	20
200	200	50.923426	42.431444	6	32.504559	24
250	10	5.065344	2.204217	12	3.330899	24
300	10	8.886002	3.436704	16	5.941652	24
500	10	44.256278	10.670822	18	28.408248	24
750	10	177.909444	23.692718	22	105.702476	24

Figure 5: *Timings in different cases*

seen in Figure 3 and Figure 4. Only 4 trajectories are plotted upon random selection. The x_{pred} in the figure is derived from serial code, and $x1_{pred}$ is given by parallel code. From the plots, the trajectories of x_{pred} and $x1_{pred}$ overlap with each other, and it doesn't differ much from that of x which is the orignal hidden state. Therefore, the correctness of both serial and parallel algorithm can be verified.

To get the performance of parallization, we tested under different number of threads, different datasizes, as well as the versions with and without parallel inverse. Figure 5 shows the overall result, which includes timing for all cases. Note that to get the optimized performance on the system, even for parallel version, we used `-O3 -march=native -fno-tree-vectorize` clause for compiling all codes.

dim_x is the same as dim_y in the experiment setting. Since in EM algorithm, the prediction of current timestamp t depends on all previous timestamp results, there is not much parallization for T, T is fixed as 10 for larger dimension x. P is the number of threads. When dim_x exceeds 250, omp1, the version with parallel inverse takes dominance in shortening the timings, so we set max P to be 24 for no parallel inversion version without influencing the overall conclusion. Table 5 reports the optimized timing and number of threads for the algorithm under each dim_x. We observed a bottleneck when p is greater than the number reported in the table, because it drains CPU time by giving the kernel more work to do as number of threads increases. The speed up and scaling of the algorithm is discussed in the following.

a **strong scale**

By Amdahl's law, speed up is calculated by $S(p) = \frac{t(1)}{t(p)} = \frac{1}{c_s + \frac{1-c_s}{p}}$. Figure 6 shows the speed up with optimized timing (optimized number of threads) observed with respect to different data sizes (dim_x) ranging from 50 to 750. As the data size increases, we can get a more obvious speed up. Also, we can observe that when N<200, the non-parallization of inverse is more efficient, so it would come to a bottleneck. But then N>250, inverse is dominant, so the speed up also increases faster.

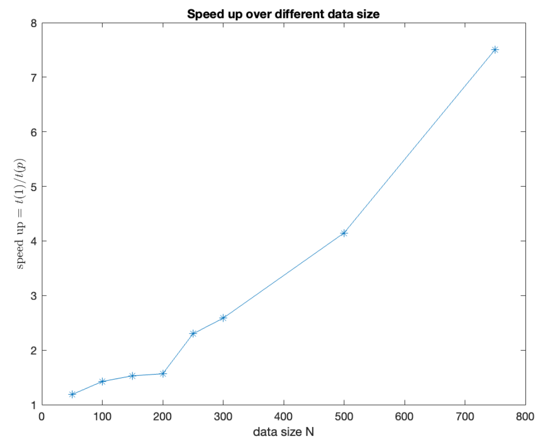


Figure 6: *Speed up vs data size*

The strong scalability is tested on two data sizes, $dim_x = 500$ and $dim_x = 750$. By Amdahl's law, efficiency is calculated by $E(p) = \frac{S(p)}{p} = \frac{t(1)}{pt(p)}$. Figure 8 shows the result of strong scalability v.s. different number of threads, ranging from 2 to 24. The y-axis is the efficiency, which is better when closer to 1. From the plot, for a larger data size, it has a higher efficiency, so the strong scalability of the system behaves as expected. However, in the idealworld, a problem would scale linearly where the speedup (in terms of work units completed per unit time) is equal to the number of processing elements used, as indicated by red line in Figure 7. Amdahl's Law assumes an infinitely fast network, but in our project, the communication overhead increases with number of processes, also there is a proportion of non-parallization part of EM algorithm. Hence the ideal linear strong scaling can't be achieved in real world.

b **weak scale**

Weak scalability is to test the constant workload per processor. Efficiency formulation is

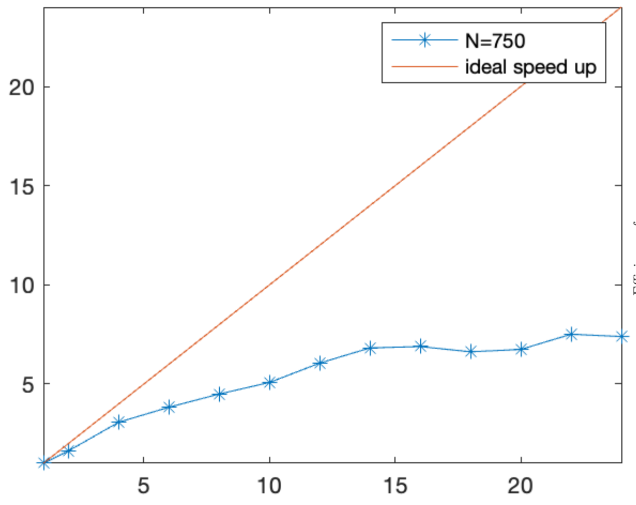


Figure 7: *Speed up vs number of threads, compared to ideal speed up*

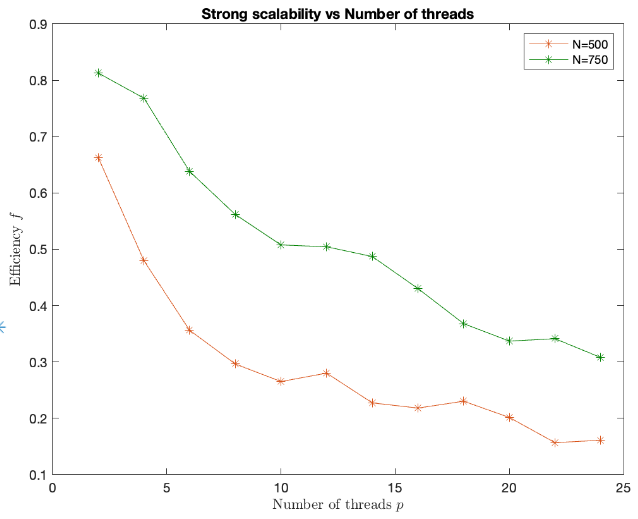


Figure 8: *Strong scalability*

$E(p) = \frac{t(1)}{t(p)}$, but the problem size increases proportionally with number of processors. As for the weak scalability, Our test Starts from $N = dim_x = 200$, $p = 8$ which means $N = 25$, $T = 10$ for each processor. Since our parallelization is preformed within one loop, use N rather than $N \times N$ as the data size. Ideally, weak scalability should perform as 1 for all number of processors, but our test in Figure 9 shows it drops dramatically when processor increases. The main reason here is that as problem size gets larger, the latency of data transfer between the nodes increases for the limited bandwidth of communication network. Also due to the nature of EM algorithm, there is not much code optimization for parallel runs, resulting into a load balancing problem.

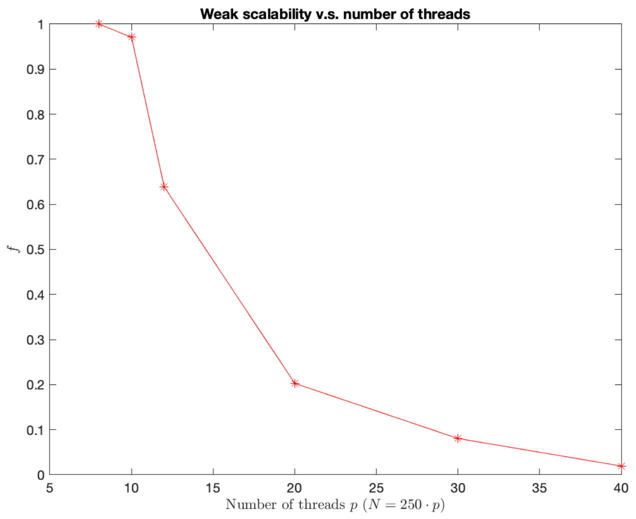


Figure 9: *Weak scale*

4 Conclusion

We implemented the Expectation-maximization algorithm with Kalman filter in C++ and tested the correctness of our code by plotting and comparing the true trajectory and the predicted trajectory of one sample data. We then implemented an Openmp version by parallelizing matrix operations in the algorithm, and tested the speed up, the strong scalability, and the weak scalability. Although speed up and weak scalability deviates the ideal results a lot, it makes sense because there is a large portion of sequential dependence in the algorithm which we can not parallelize.

References

[1] Sam Roweis and Zoubin Ghahramani. A unifying review of linear gaussian models. *Neural Computation*, 11(2):305–345, 1999.

[2] Inc. Google. Google scholar. 2022.

[3] Zoubin Ghahramani and Geoffrey E Hinton. Parameter estimation for linear dynamical systems. 1996.