AXLE: Computationally-efficient trajectory smoothing using factor graph chains

Edwin Olson

Abstract—Factor graph chains— the special case of a factor graph in which there are no potentials connecting non-adjacent nodes— arise naturally in many robotics problems. Importantly, they are often part of an inner loop in trajectory optimization and estimation problems, and so applications can be very sensitive to the performance of a solver.

Of course, it is well-known that factor graph chains have an $O(N)$ solution, but an actual solution is often left as “an exercise to the reader”... with the inevitable consequence that few (if any) efficient solutions are readily available.

In this paper, we carefully derive the solution while keeping track of the specific block structure that arises, we work through a number of practical implementation challenges, and we highlight additional optimizations that are not at first apparent. An easy-to-use and self-contained solver is provided in C, which outperforms the Apriltag general-purpose sparse matrix factorization library by a factor of 7.3x even without specialized block operations.

The name AXLE reflects the names of the key matrices involved (the approach here solves the linear problem $AX = E$ by factoring $A$ as $LL^T$), while also reflecting its key application in kino-dynamic trajectory estimation of vehicles with axles.

I. INTRODUCTION

Let’s begin with a motivating application— suppose we have a self-driving car tracking a number of other agents around it. Each observation of other agents is contaminated by noise, and we’d like to fit a maximum likelihood trajectory to those points in order to predicting the object’s future position.

Better predictions can be obtained by imposing a motion model while performing the trajectory fitting [1]. This can greatly improve the quality of the results, improving the safety of the car. But we don’t know what motion model to use a priori, so we might perform model selection over a number of candidate models including holonomic, unicycle, or bicycle [2]. We might also only observe position (e.g. from a LIDAR) but also want to infer velocity or acceleration as well, given dynamic constraints on the vehicle model.

No problem— we can pose the trajectory estimation problem as a least-squares problem: we collect observations of the agent positions over a several seconds of motion, assume a type of motion model, then solve for the state of the vehicle at each time step subject to the constraints of the motion model. This gives us a posterior trajectory given the observations, conditioned on the kina-dynamic model used.

We also obtain a quality-of-fit measure ($\chi^2$) that we can use to compare the results obtained for different kina-dynamic models, and thus help us with model selection.

In a real-world system, dozens of agents may need to be tracked simultaneously; for each, we may need to consider multiple kina-dynamic models, or models reflecting context-dependent behavior (e.g., is another car’s trajectory best described by “stay in lane” or by a “lane change” behavior?). Hundreds of trajectory fitting operations might be required, only to have to repeat this process 50 ms later when sensors provide a new set of observations.

One alternative approach to this problem is to maintain a bank of (recursive) Extended Kalman Filters [3], one for each possible kina-dynamic model. However, EKFs pose several practical challenges. Suppose, for example, that orientation of other vehicles is either poorly observed or not directly observable at all. Because unicycle and bicycle kinematic models are controlled through the vehicle orientation, a poor initial orientation estimate can lead to poor predictions. Worse, because errors caused by linearization are irreversible in an Kalman filtering paradigm, even the addition of future unambiguous observations cannot repair past errors. An EKF might be attractive due to its constant-time update (the model size is constant), but because a full posterior trajectory is desired (not just the posterior final pose), a linear-time smoothing pass would be required anyway.

These problems make a non-linear smoothing approach necessary. In a smoothing paradigm, the entire state vector
is revised during each iteration. Each iteration allows for a
new linearization point to be selected, allowing new informa-
tion to cause old information to be “reinterpretted”. This
flexibility comes at the cost of computational complexity,
and with so many potential smoothing operations to perform,
performance can quickly become a bottleneck.

In this paper, we will use a factor graph approach to
solving these trajectory smoothing problems [4], [5]. We’ll
exploit the special structure of these trajectory fitting prob-
lems, namely that they are chains.

Ultimately, the tools used here are the same high-
performance SLAM systems. However, SLAM systems must
handle more complex factor graphs than chains— after all, a
SLAM system would be fairly pointless without the ability
to handle loop closures. In order to perform well, SLAM
systems must address a variety of additional challenges:
maintaining sparsity through variable reordering [6], pre-
conditioners [7], handling information matrices with an
unpredictable block structure, handling fill-in in the matrix
factorization of an unpredictable block structure [8], and so
forth. These systems perform well and automatically exploit
the structure of a chain to solve them in (generally) \(O(N)\)
time. However, they carry the baggage of their general-
purpose nature, which limits their performance versus a
solver optimized for a special case.

For safety-critical systems, validation is also critically
important. A MISRA-C [9] implementation of a general-
purpose factor graph solver, along with analysis of runtimes
and memory usage would be difficult to say the least. The
special case outlined here, in contrast, could be implemented
in a strict conformance with MISRA-C provided that the
length of the state vector was fixed in advance. That makes
this particular special case especially important to safety-
critical applications.

The contributions of this paper include:

- We provide an accessible derivation of a Cholesky
  factorization-based solver for factor graph chains. This
derivation extensively exploits the special block structure
  arising in chains in order to operate faster than a
general-purpose solver. This derivation might also be
  useful to those building intuition about sparse matrix-
  based SLAM systems, as the sparsity structure is made
  explicit.
- We provide a stand-alone implementation of our solver
  in C, and benchmark it against a state-of-the-art solver,
  AprilSAM [10].

II. PRELIMINARIES

The state-of-the-art in Simultaneous Localization and
Mapping (SLAM) is based on least-squares optimization
over the entire state vector using sparse matrix factorization.
Nodes in the factor graph represent variables whose values
should be computed, while factors represent information
about the value of the nodes (i.e., an observation).

In this section, we’ll review the basic mathematics in-
volved so as to illustrate how the a factor graph chain leads
to special structure that can be exploited. We’ll do this in the
context of a specific example: fitting a trajectory to a time
sequence of point observations (i.e., as though the vehicle
was observed via a LIDAR), adding factors representing a
unicycle motion model, and solving for posterior position,
velocity, and orientation at each time step. Naturally, the
method proposed here can be used for simpler or more
complex kino-dynamic models as well.

Our state vector is \(x\), which is a \(4N\times1\) vector that “stacks”
all of the variables for all \(N\) vehicle positions (see Fig. 1).
We’ll write the state for just the \(i^{th}\) time step as \(x_i\). Each \(x_i\)
is a four element vector, containing (in this order) positions \(x\)
and \(y\), velocity \(v\), and heading \(\theta\). When we want to specify
a single scalar component of a variable, for example the
orientation \(\theta\) for the \(i^{th}\) pose, we’ll use super-scripts: \(x_i^\theta\).

We will have two kinds of factor potentials: unary poten-
tials constraining the position \((x_i^x\) and \(x_i^y)\) of the vehicle
according to the LIDAR observation, and binary potentials
connecting consecutive nodes (e.g. \(x_i\) and \(x_{i+1}\)) that reflect
a unicycle motion model. If there are \(N\) states (i.e., \(x_0\) through
\(x_{N-1}\)), there will be \(N\) unary constraints and \(N-1\) binary
constraints.

To form the optimization problem, we must provide for
each factor \(j\):

- The residual \((r_j)\), an \(M \times 1\) column vector. This is a
  measure of how much error there is in the current factor.
- The Jacobians of the residual function with respect to
each \(x_i\). The dimension of each Jacobian is \(M \times 4\). Note
  that in our problem, these will be zero for all but one or
two \(i\).
- An \(M \times M\) symmetric weight matrix \(W_j\), which en-
codes the relative importance of the elements in the
residual, and the importance of factor \(j\) in comparison
to the other factor potentials.

To make this more concrete, let’s consider two basic obser-
vation models. Note that the dimension \(M\) varies according
to the type of factor.

A. Position observation

Every \(x_i\) will have a unary factor representing the position
of the vehicle as measured by a LIDAR. Suppose that a
LIDAR detects the position of the other vehicle at time \(i\)
as being at location \((x = 5, y = 7)\). Supposing that this
is the \(j^{th}\) observation made, we define the two-dimensional
residual function \(r_j\) as:

\[
    r_j(x) = \begin{bmatrix} 5 - x_i^x \\ 7 - x_i^y \end{bmatrix} 
\]  

We next need to linearly approximate \(r_j(x)\) around our
current best estimate of the state \(x^-\) (which we will assume
is all zeros). Note that because \(r_j(x)\) is linear, this linearization
is exact:

\[
    r_j(x) = r_j(x^-) + J_{x_i}r(x^-) \Delta x_i 
\]  

where the first term is simply the value of the observation
equation evaluated at our current state estimate (assumed
to be zero for this example). In the final line, we simplify notation by letting \( r_j = r_j(x^-) \). The residual and Jacobian are then:

\[
    r_j = \begin{bmatrix} 5 \\ 7 \end{bmatrix}
\]

\[
    J_{x_i}^r = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix}
\]

The \( \chi^2 \) error for this factor– which is the quantity we will minimize using least squares– is the weighted square error:

\[
    \chi^2_j = r_j^T W_j r_j = (r_j + J_{x_i}^r \Delta x_i)^T W_j (r_j + J_{x_i}^r \Delta x_i)
\]

We’ll set \( W_j = I \), though in practice, some “tuning” of the weights would be used to reflect the noise model of the LIDAR.

It will be useful below for us to rewrite the \( \chi^2 \) expression in terms of the whole state vector (\( \Delta x \) and not just \( \Delta x_i \)). To do this, we simply need to extend \( J_{x_i}^r \) to be \( M \times 4N \), adding zeros in the positions the derivative is zero, and adding zeros to \( W_j \) so that it is \( 4N \times 4N \):

\[
    J_{x}^r = \begin{bmatrix} 0_{2 \times 4} & \ldots & J_{x_i}^r & \ldots & 0_{2 \times 4} \end{bmatrix}
\]

(7)

allowing us to write:

\[
    \chi^2_j = (r_j + J_{x_i}^r \Delta x_i)^T W_j (r_j + J_{x_i}^r \Delta x_i)
\]

(8)

which can be expanded:

\[
    \chi^2_j = r_j^T W_j r_j + 2 \Delta x^T J_{x_i}^r T W_j r_j + \Delta x^T J_{x_i}^r T W_j J_{x_i}^r \Delta x
\]

and differentiated with respect to \( \Delta x \), setting the result to zero:

\[
    \frac{\partial \chi^2_j}{\partial \Delta x} = 2 J_{x_i}^r T W_j r_j + 2 J_{x_i}^r T W_j J_{x_i}^r \Delta x = 0
\]

(9)

\[
    J_{x_i}^r T W_j J_{x_i}^r \Delta x = -J_{x_i}^r T W_j r_j
\]

While it may not be obvious, we have a \( 4N \times 4N \) matrix (\( J_{x_i}^r T W_j J_{x_i}^r \)), multiplied by the \( 4N \times 1 \) vector that we are solving for (\( \Delta x \)), equaling a \( 4N \times 1 \) vector on the right-hand side (\( -J_{x_i}^r T W_j r_j \)). As we add additional factor potentials to our problem, each factor potential will contribute to the left-hand matrix or the right-hand vector additively.

If you are unfamiliar with how the structure of the factor graph leads to sparsity in the linear system, this is a key moment. Consider the \( 4N \times 4N \) matrix \( J_{x_i}^r T W_j J_{x_i}^r \). The product will only be non-zero where the Jacobians (Eqn. 7) are non-zero. In this case, because the factor potential is unary, the Jacobian is only non-zero for the state variables belonging to \( x_i \), i.e.:

\[
    J_{x_i}^r T W_j J_{x_i}^r = \begin{bmatrix} 0_{4 \times 4} & \ldots & 0_{4 \times 4} & \ldots & 0_{4 \times 4} \\ 0_{4 \times 4} & \ldots & J_{x_i}^r T W_j J_{x_i}^r & \ldots & 0_{4 \times 4} \\ 0_{4 \times 4} & \ldots & 0_{4 \times 4} & \ldots & 0_{4 \times 4} \end{bmatrix}
\]

\[
    J_{x_i}^r T W_j J_{x_i}^r = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

(10)

(11)

We can now write the linear approximation of \( r_j(x) \):

\[
    r_j(x) \approx r_j + J_{x_i}^r \Delta x_i + J_{x_{i+1}}^r \Delta x_{i+1}
\]

(12)

And of course, the \( \chi^2 \) loss for the observation is of the same form as before, but with a longer expression for \( r_j(x) \) which has two Jacobian terms instead of just one. We can similarly write the full-state Jacobian, but now it has two non-zero sections:

\[
    J_{x}^r = \begin{bmatrix} 0_{4 \times 4} & \ldots & J_{x}^r & \ldots & 0_{4 \times 4} \end{bmatrix}
\]

(13)

(14)

As with the unary potential, it is critical to understand the impact of this form on the sparsity of the matrix \( J_{x}^r T W_j J_{x}^r \), which will have a \( 2 \times 2 \) block of non-zero sub-matrices, and block-indices \((i, i), (i, i+1), (i+1, i), \) and \((i+1, i+1)\). Note that the sub-blocks at \((i, i+1)\) and \((i+1, i)\) will be transposes of each other:

\[
    J_{x}^r T W_j J_{x}^r = \begin{bmatrix} 0_{4 \times 4} & 0_{4 \times 4} & 0_{4 \times 4} & 0_{4 \times 4} \\ 0_{4 \times 4} & J_{x_i}^r T W_j J_{x_i}^r & J_{x_i}^r T W_j J_{x_i+1} & 0_{4 \times 4} \\ 0_{4 \times 4} & J_{x_i+1}^r T W_j J_{x_i}^r & J_{x_i+1}^r T W_j J_{x_i+1} & 0_{4 \times 4} \\ 0_{4 \times 4} & 0_{4 \times 4} & 0_{4 \times 4} & 0_{4 \times 4} \end{bmatrix}
\]

(15)
C. Building the linear system

We now leave the language of factor graphs and enter the language of linear algebra. By looping over all of the factor potentials in our chain, we sum up the contributions to the left-hand matrix and right-hand side column vector. With unary constraints only contributing non-zero blocks along the diagonal, and the binary constraints only contributing a 2 × 2 block of non-zeros along the diagonal, the sum of all the contributions will have the following form:

\[
\begin{bmatrix}
A_0 & A_1 \\
A_1^T & A_2 & A_3 \\
A_3^T & A_4 & A_5 \\
A_5^T & A_6 \\
\end{bmatrix}
\begin{bmatrix}
\Delta X_0 \\
\Delta X_1 \\
\Delta X_2 \\
\Delta X_3 \\
\end{bmatrix}
= 
\begin{bmatrix}
E_0 \\
E_1 \\
E_2 \\
E_3 \\
\end{bmatrix}
\tag{16}
\]

In our example, each block \(A_i\) has dimension 4 × 4, and both \(\Delta X_i\) and \(E_i\) have dimension 4 × 1.

We solve the system for \(\Delta X\) by factoring \(A\)–which is symmetric and positive definite\(^1\) using Cholesky decomposition, i.e., by computing the lower triangular matrix \(L\) such that \(A = LL^T\). Critically, because \(A\) has a special block structure, so does \(L\):

\[
L = \begin{bmatrix}
L_0 & L_1 & L_2 & L_3 & L_4 & L_5 & L_6 \\
\end{bmatrix}
\tag{17}
\]

We can solve for the individual sub-matrices \(L_i\) by algebraically manipulating the product \(LL^T\) in terms of the individual \(L_i\)s and setting those products equal to the corresponding elements of \(A\), giving:

\[
L_i = \begin{cases}
\text{chol}(A_0), & \text{if } i = 0 \\
A_i^T(L_{i-1}^T)^{-1}, & \text{if } i \text{ odd} \\
\text{chol}(A_i - L_{i-1}L_{i-1}^T), & \text{otherwise}
\end{cases}
\tag{18}
\]

Note that we assume that \(\text{chol}\) computes the lower left triangular factor \(L\) of its argument \(A\) such that \(LL^T = A\). Note that \(\text{chol}\) is only being computed for individual sub-matrices of dimension 4 × 4.

We will solve the resulting problem \(AX = LL^TX = E\) in the usual fashion, by first letting \(U = L^TX\) and solving \(LU = E\) in increasing order of \(i\):

\[
U_i = \begin{cases}
L_{2i}^{-1}E_0, & \text{if } i = 0 \\
L_{2i}^{-1}(E_i - L_{2i-1}U_{i-1}), & \text{otherwise}
\end{cases}
\tag{19}
\]

And finally, solving \(L^TX = U\) for \(X\), this time solving for \(X\) in decreasing order of \(i\):

\[
X_i = \begin{cases}
L_{2i+1}^{-1}U_i, & \text{if } i = N - 1 \\
L_{2i}^{-1}(U_i - L_{2i+1}X_{i+1}), & \text{otherwise}
\end{cases}
\tag{20}
\]

These update equations represent the primary contribution of this paper. An implementation will likely create an array of the \(A_i\), \(E_i\), \(L_i\), \(U_i\), and \(X_i\) matrices. The length of each list, and the dimension of each matrix, are generally known in advance. An implementation then simply computes the \(L_s\), \(U_s\), and \(X_s\). From this, it is obvious that:

- The computational costs are not only \(O(N)\) but actually fixed.
- The memory requirements are also \(O(N)\), and could be pre-allocated.
- The mix of dense matrix operations needed to implement a solution are fairly limited and operate on matrices of fixed size, which makes implementation with highly-optimized codes (e.g. using SIMD instructions) possible.

For safety-critical applications, these properties are highly desirable, as they make the runtime deterministic. Even if runtime determinism is not a priority, the memory access patterns are highly regular, which can lead to better cache performance.

The overall flow of the algorithm, and the sequence in which each matrix is computed, is given in Alg. 1.

### III. Discussion

We address a handful of assorted points here.

**QR versus Cholesky.** It is also possible to formulate the solution of a factor graph chain in terms of QR decomposition, instead of Cholesky. The advantage of QR decomposition is that the condition number of the key matrix is the square root of the condition number formed by the normal equations (i.e., what is done here.) For problems that are ill-conditioned, QR factorization can be more stable. On the other hand, building up the \(A\) matrix from factors is quite convenient from an implementation perspective, and trajectory fitting problems are usually well-conditioned.

**Other kinodynamic factor potentials.** Different kinodynamic motion models can be incorporated through appropriate modification of the binary factors (and possibly modifications to the state variables). For example, a second order model could be obtained by replacing \(\theta\) with a linear acceleration, and by turning the binary constraints into a double integrator.

**Other unary factor potentials.** In some systems, it may be possible to directly observe vehicle orientation (e.g., by recognizing the shape of a car) or to directly measure closing

---

\(^1\)The matrix \(A\) will be positive definite when the problem is “fully constrained”. In situations where some variables may be only weakly observable, Tikhonov regularization can be helpful.
rate (e.g., by doppler measurements). This approach is not limited to one unary potential per state—many different potentials could be incorporated simultaneously. In a different application, we also added unary potentials to penalize negative velocities. MaxMixtures [11] could be also used to increase robustness to particularly poor data.

Variable reordering. In conventional SLAM systems, a critical step in maintaining the sparsity of the Cholesky factors is in finding a permutation matrix of the state vector that minimizes “fill-in”. Here, however, we are simply using the time-based variable ordering which achieves the $O(N)$ performance for the case of a chain. This significantly reduces the complexity of the solver code.

IV. EXPERIMENTAL RESULTS

A. Trajectory fitting

A basic trajectory fitting problem is shown in Fig. 2. This problem follows the formulation and factor potentials given in this paper. CPU time, as measured on a i9-8950HK processor was 140 us (two iterations of 70 us each.)

B. Trajectory interpolation

We also show how the same code can be used to perform trajectory interpolation. The interpolations are almost identical to the trajectory fitting operations, except that the unary potentials also observe heading (and thus are $3 \times 1$ measurements), and that only the first and last node have the unary potentials. It is thus up to the optimization framework to “fill in” the rest.

In Fig. 3, we show an interpolation for a lane-change like maneuver, and in Fig. 4, we show an interpolation where a k-point turn is required. Both systems use the unicycle kinematic constraints described here without any additional modifications.

C. Computational Performance

The performance of our method is shown in Fig. 5. Our method is between 3.7x and 7.3x faster than a general-purpose matrix factorization library. This is attributable to both the streamlined computation on small block matrices and the elimination of dynamic memory allocation.
V. Conclusion

We have presented an algorithm for performing inference on factor graph chains, with a worked-out application to trajectory fitting. This work was motivated by the need to rapidly fit trajectories to sensor data, both filtering noise and inferring latent state of the vehicle (like velocity and heading) which were not directly observable given position observations of the target.

While it is well-known that factor-graph chains have an $O(N)$ solution [12], we propose a particularly simple way of indexing the matrices that implicitly exploits the sparsity of the underlying problem. An advantage of this approach is that an implementation can be much more simple than a general-purpose sparse-matrix solver, with implementations compliant with MISRA-C even being possible.

The performance of this approach is significantly faster than that using a general-purpose SLAM system. An example implementation of this algorithm in C is available at https://github.com/edwinolson/axle.

References


