Real-Time 6-DOF SLAM for a Quadrotor Helicopter

Stephen Chaves  Schuyler Cohen  Patrick O'Keefe  Paul Ozog
University of Michigan

Abstract—This paper presents a real-time implementation of a simultaneous localization and mapping (SLAM) framework for a quadrotor helicopter. The framework features a front-end interface to the helicopter and a back-end solver of the graph representation of the robot and its environment. The back-end implements techniques from incremental Smoothing and Mapping (iSAM) in order to solve the graph in real time and provide instantaneous visualization of the robot trajectory and landmark positions. iSAM incrementally updates the information matrix factorization, providing significant time savings over other SLAM solvers. The speedups resulting from iSAM allow the 6-degree of freedom (DOF) SLAM problem to be solved in real time on a consumer-grade computer. Our software implementation was successfully tested in simulation and using an AR.Drone quadrotor helicopter.

I. INTRODUCTION

SLAM problems present many computational challenges, most of which stem from the large number of observations and states that must be reconciled. Least-squares approaches are particularly affected by these problems; naive implementations are too slow to be used in real time scenarios. Work from [1] was the basis for incremental approaches such as iSAM [2], which allow least-squares based methods to run in real time. For this work, we seek to implement and apply these techniques to a quadrotor helicopter.

For robotics research, quadrotor helicopters are a popular platform because of their maneuverability, stability, and relatively low mechanical complexity. In [3], an extended Kalman filter (EKF) was successfully used for real-time SLAM on a quadrotor platform, which avoids the problems associated with least-squares altogether. Very recent work [4] has allowed for fully autonomous quadrotors with SLAM capabilities. However, the sensor suite on their platform is very expensive.

In this paper, we present a modular implementation of a SLAM system. We then discuss optimizations to our SLAM algorithm – inspired by iSAM – which allow for real-time operation. This paper also covers our generic observation and motion models for 6-DOF systems. Finally, we apply our system to a number of simulations and real-world datasets including a quadrotor helicopter.

II. SYSTEM ARCHITECTURE

The architecture of our SLAM software was developed with modularity and versatility in mind. As a result, our software consists of two subsystems: a front-end that interfaces with the robot and creates nodes and factors from odometry and landmark observations, and a back-end that solves the SLAM problem from a graph generated from these nodes and factors.

In our framework, each robot node is a six dimensional pose and each landmark node is a three dimensional position. These poses and positions make up the state vector solved by the back-end. The factors represent constraints between two unknowns or a constraint for a single unknown. From these factors we can extract Jacobians and residuals associated with the underlying optimization problem. Describing a system with nodes and factors creates an elegant representation of any least-squares problem. This representation was successfully applied to the SLAM problem in [1].

Our software system consists of multiple processes. In order to pass information between them, we employ the Lightweight Communications and Marshalling (LCM) library [5]. One of the main benefits of using LCM is the ability to record and play back the messages sent between processes. This tool allows us to keep a record of the experiments we performed and evaluate different algorithms on the same real-world dataset. The front-end also uses LCM to receive data from the robot and to send it motion commands.

By designing the software as a modular framework, the back-end is a generic solver for any graph-based SLAM problem. Thus, the back-end can be easily transferred between SLAM applications. Only the front-end is specific to the application, and it is responsible for data association. The back-end solver we implemented is inspired by iSAM (discussed in Section III). Moreover, the modularity of our system allows us to test different least-squares solvers without altering the front-end.

III. INCREMENTAL SMOOTHING AND MAPPING

Incremental smoothing and mapping is a relatively recent approach to solving the full SLAM problem [2]. The full
SLAM problem, in contrast to the online SLAM problem, recovers the full posterior of the robot trajectory and landmark positions instead of just the current robot pose and landmark positions [6].

In standard least-squares SLAM, we solve a system of equations such as
\[
\begin{align*}
\Delta x' &= \arg\min_{\Delta x} (J^T \Delta x - r)^T \Sigma^{-1} (J^T \Delta x - r) \\
&= \arg\min_{\Delta x} \| J^T \Delta x - r \|_2^2
\end{align*}
\]
where \( x \) is the state vector that contains all robot poses and landmark positions, \( J \) is the Jacobian of the observation model that predicts measurements given the state vector, \( \Delta x \) is the deviation of \( x \) from the linearization point, and \( r \) is the residual of observations versus the predicted measurements. The minimizing solution results in the standard normal equations given by
\[
(J^T \Sigma^{-1} J) \Delta x = J^T \Sigma^{-1} r
\]
This is solved via the Cholesky decomposition of the information matrix, \( J^T \Sigma^{-1} J \).

iSAM makes a change to this problem formulation by considering the Cholesky decomposition of \( \Sigma^{-1} \) – written as \( \Sigma^{-T/2} \) to denote the upper triangular result of the decomposition – and rewriting the least squares problem as
\[
\begin{align*}
\Delta x' &= \arg\min_{\Delta x} (J^T \Delta x - r)^T \Sigma^{-1/2} \Sigma^{-T/2} (J^T \Delta x - r) \\
&= \arg\min_{\Delta x} \| \Sigma^{-T/2} (J^T \Delta x - r) \|^2 \\
&= \arg\min_{\Delta x} \| (A \Delta x - b) \|^2
\end{align*}
\]
where
\[
\begin{align*}
A &= \Sigma^{-T/2} J \\
b &= \Sigma^{-T/2} r
\end{align*}
\]
This allows us to solve \( A \Delta x = b \) by applying QR factorization directly to \( A \). This is advantageous because it avoids the squaring of the matrix condition number that is associated with the normal equation form. Moreover, there is the nice property that the \( R \) that results from the QR decomposition of \( A \) is equivalent to the upper triangular term in the Cholesky decomposition of \( A^T A \) if \( A \) is real [2].

iSAM recovers the posterior by doing fast incremental updates to the factorization of \( A \). This means that calculations are only performed on elements that are actually affected by new measurements. We can get away with this because we often have a very good estimate of the state vector, so doing a full batch re-linearization and solution at each step wastes calculations because most elements of \( \Delta x \) will be close to zero. However, iSAM periodically performs a full batch solution in order to compensate for accumulated linearization errors. Over time, loop closures greatly decrease the sparsity of the matrix factorization, so variable reordering is performed during each batch solution step.

A. Givens Rotations

When a new measurement row is added to \( A \), we can incrementally update \( R \) instead of performing the entire QR decomposition again. This can be achieved with Givens rotations.

Givens rotations are a way to introduce a zero at a specified point in a matrix by premultiplying by a matrix \( G \) of the form
\[
G(i, k, \theta) =
\begin{bmatrix}
1 & \cdots & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & c & \cdots & -s \\
\vdots & \ddots & \vdots & \ddots & \vdots \\
k & \cdots & s & \cdots & c \\
\vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & \cdots & 1
\end{bmatrix}
\]
where \( c = \cos(\theta) \) and \( s = \sin(\theta) \) for some \( \theta \). We follow the algorithms in Golub and Van Loan’s Matrix Computations to find \( c \) and \( s \) to zero a particular element in a matrix [7].

The \( R \) term that results from the QR decomposition of \( A \) is upper triangular by definition. When a new measurement occurs, new rows need to be added to \( A \). Here, we append them to \( R \), which is then no longer upper-triangular. A series of Givens rotations is applied to the elements that are below the diagonal. This is called triangularization, and once the below-diagonal elements have been removed via the rotations, the resulting upper-triangular matrix is equivalent to the QR decomposition of the full \( A \) matrix [7]. The same series of rotations needs to be applied to our right-hand side term, \( b \).

After the rotations have been performed, the system can be solved with simple back-substitution.

B. Variable Reordering

When a robot closes a loop, a correlation is introduced between the current pose and a previously observed landmark. This landmark is connected to earlier poses in the trajectory as well. These correlations greatly increase the number of non-zero elements in the matrix factorization which in turn increases the computational complexity of updating and solving the system. Figure 1(a) shows the large swaths of non-zero elements that are introduced during loop closures.
Fig. 1. The factorization of the $A$ matrix after 300 time steps of simulated data that includes loop closures. (a) Before variable reordering, the loop closures introduce many non-zero elements. (b) After variable reordering, we have a much more sparse factorization. Over 50% of the non-zero elements were removed.

In order to avoid this problem, we can perform variable reordering. This is essentially a permutation of the columns of $A$ where columns that belong to the same node in the SLAM graph are kept together. This new order influences the variable elimination, which has a profound effect on the sparsity of the factorization. Figure 1(b) shows the result of applying variable reordering to simulated data that has experienced several loop closures.

The best column variable ordering is NP hard, and iSAM uses COLAMD as a heuristic to assist in the process [8]. Our implementation uses a simplified heuristic that counts the number of connected nodes in the graph. It is not the minimum degree ordering, but we have found it to be more than adequate in our tests. Figure 1 is an example of the application of our simplified heuristic.

IV. BACK-END MOTION AND OBSERVATION MODELS

Because of the relative complexity of 6-DOF coordinate transforms, we used a notation that allows us to abstract away the linear algebra operations necessary for the non-linear motion and observation models. To achieve this, we adopt the notation from [9], [10] to describe three particular spatial relationships of 6-DOF coordinate frames: compound, inverse, and composite. These relationships are used throughout our system.

A. 6-DOF Spatial Relationships

Let $x_{ij} = [t_{ij}^T \Theta_{ij}]^T$ be a vector in $\mathbb{R}^6$ describing the relative pose of frame $j$ with respect to frame $i$. $t_{ij}^T$ is the $3 \times 1$ translation vector from frame $i$ to frame $j$ as expressed in frame $i$. $\Theta_{ij}$ is the $3 \times 1$ vector describing the Euler angles of the $x$, $y$, and $z$ axes. We define the function $\text{rotxyz} : \mathbb{R}^3 \rightarrow \mathbb{R}^{3 \times 3}$ that maps a vector of Euler angles to a rotation matrix as follows:

$$R_{ij}^i = \text{rotxyz}(\Theta_{ij})$$

where $R_{ij}^i$ is the matrix that rotates frame $j$ into frame $i$.

These definitions allow us to describe the transformation matrix that uniquely relates homogeneous 3D points in frame $j$ to homogeneous points in frame $i$ as:

$$H_{ij}^i = [R_{ij}^i \ t_{ij}^T \ 0 \ 1]$$

Note that given $H_{ij}^i$, we can easily compute $x_{ij}$ and vice versa using APRIL’s LinAlg Java package.

1) Compounding Operation: Let the $\oplus$ operator describe the spatial relationship between two frames arranged “head-to-tail”:

$$x_{ik} = x_{ij} \oplus x_{jk}$$

$x_{ik}$ can be computed from recovering the $6 \times 1$ vector corresponding to the matrix

$$H_{ik}^i = H_{ij}^i H_{jk}^i$$

The Jacobian of the compounding operation $J_{\oplus}$ is given in [10]. Using this matrix, we can compute the covariance of $x_{ik}$ as the covariance projection

$$\Sigma(x_{ij}) = J_{\oplus} \left[ \frac{\Sigma(x_{ik})}{\Sigma(x_{kj}, x_{ij})} \right] J_{\oplus}^T$$
2) Inverse Operation: Let the $\odot$ operator describe the inverse of a relative pose vector:

$$x_{ji} = \odot x_{ij}$$

This operation is computed from the transformation matrix

$$H_i^j = (H_i^j)^{-1} = \begin{bmatrix} (R_i^j) & - (R_i^j) \cdot t_{ij}^j \end{bmatrix}$$

The first-order covariance projection of this operation is

$$\Sigma(x_{ji}) = J_\odot \Sigma(x_{ij}) J_\odot^T$$

where the closed-form expression for the Jacobian is given in [10].

3) Composition Operation: We define the composite operation on two frames related “tail-to-tail” using the compounding and inverse operations:

$$x_{jk} = \odot x_{ij} \oplus x_{ik}$$

where the Jacobian of this relationship is

$$\odot J_\oplus = J_\odot \begin{bmatrix} J_\odot & 0_{6x6} \\ 0_{6x6} & I_{6x6} \end{bmatrix}$$

and thus the first-order covariance projection of the composite operation is

$$\Sigma(x_{jk}) = \odot J_\odot \begin{bmatrix} \Sigma(x_{ij}) & \Sigma(x_{ij}, x_{ik}) \\ \Sigma(x_{ik}, x_{ij}) & \Sigma(x_{ik}) \end{bmatrix} J_\odot^T$$

B. Node Initialization

Every timestep in which the robot moves or a new feature is observed, a new node is created and initialized. To initialize a node means to assign that node a pose (for robot poses) or a point (for landmarks) in the global frame. This is computed by the measurement that relates the new node to a previous node. To initialize a node without connecting to a previously created node is a referred to as adding a “prior.”

For the following sections, let $g$ represent the global frame. For each node, a first-order covariance projection can be computed as in Section IV-A.

1) Pose Nodes: Let frame $i$ correspond to an initialized pose node in the SLAM graph. We observe an odometry measurement relating frame $i$ and frame $j$. Then we initialize node $j$ to be in the global frame with the “head-to-tail” operation $x_{gj} = x_{gi} \oplus x_{ij}$

2) Landmark Nodes: Given the relative position of a landmark frame $l$ with respect to robot frame $i$, we construct a relative pose vector by setting the orientation of $l$ to be arbitrary. Then, the relative position of the landmark in the global frame is the translation vector corresponding to $x_{gl} = x_{gi} \oplus x_{il}$

C. Factor Computation

To solve the least squares problem described in Section III, we need to compute an observation model Jacobian $J$ and residual vector $r$. Together, these comprise a “factor” between two unknowns in the graph. In the following section, we describe how these are computed using the notation from Section IV-A.

1) Pose-to-Pose Factors: Given two estimates of robot frames $i$ and $j$ (with respect to the global frame), the predicted relative pose observation is $\hat{x}_{ij} = \odot x_{gi} \oplus x_{gj}$. The Jacobian of this relationship is evaluated as in Section IV-A3. The residual vector is simply the difference between the predicted $\hat{x}_{ij}$ and the observed $x_{ij}$.

2) Pose-to-Landmark Factors: If a landmark $l$ is observed from robot frame $i$, then we can predict this measurement by the “tail-to-tail” operation. We simply assign the orientation of $x_{gl}$ to be arbitrary. Then, the estimated observation is the translational component of $x_{il} = \odot x_{gi} \oplus x_{gl}$. The residual for the landmark is the difference between this prediction and what the robot actually measured.

Note that for the computation of this factor’s Jacobian, we drop the differentiations with respect to the orientation of the landmark. This leaves us with a $3 \times 9$ Jacobian and $3 \times 3$ positional covariance matrix.

V. APRILTAG RELATIVE POSE COVARIANCE ESTIMATE

We used the AprilTag system throughout our experiments and demonstrations for relative position and orientation. The accuracy, detection performance, and small memory footprint made it particularly desirable [11].

We applied principles of linearized covariance projection to estimate a covariance matrix for the 6-DOF relative pose vector $x_{tc}$ that describes the relative pose from the tag’s frame $t$ to the camera’s frame $c$.

Denote $f$ as the function that computes the $9 \times 1$ homography vector $h$ from a set of corresponding world points and image points via the Direct Linear Transform (DLT) algorithm. By linearizing $f$ around a mean input (the observed image point coordinates), we can compute the $9 \times 9$ covariance matrix $\Sigma(h)$ to a first order approximation. From there, we project the covariance estimate of $h$ to a pose covariance by linearizing a function $g$, which maps the homography to $x_{tc}$.

For simplicity, we assume that the uncertainty of the world points and the camera calibration parameters are zero. Furthermore, we assume negligible lens distortion. All differentiations are done numerically.
Under these assumptions, the covariance estimate of the relative pose from tag to camera is simply

\[ \Sigma(x_{tc}) = J_g J_f \Sigma(u) J_f^T J_g^T \]

where \( u \) is the \( 2N \times 1 \) vector of \( N \) noisy image point coordinates and \( \Sigma(u) \) is the associated covariance matrix. We assume the feature noise realizations are independent and isotropic. The diagonal elements of this covariance matrix are tuned by the user.

The Jacobians are computed by finite difference:

\[ [J_f]_{ij} = \frac{\partial f_i}{\partial u_j} = \frac{f_i(u_0, u_1, \ldots, u_j + \epsilon, \ldots, u_{2N}) - f_i(u_0, \ldots, u_{2N})}{\epsilon} \]

for some small \( \epsilon \). One should note that when differentiating \( f \), the homography vectors must have the same sign. This is not guaranteed in typical Singular Value Decomposition (SVD) libraries because the sign of the singular vectors may be flipped when the input matrix is slightly perturbed.

Also, we take \( g \) to directly map the homography to \( x_{tc} \). The covariance projections operations referred to in Section IV-A are therefore included in \( g \).

VI. PARROT AR.DRONE

The quadrotor helicopter we used was a Parrot AR.Drone, shown in Figure 2. Despite its reputation as an augmented reality gaming platform (for which it is marketed), the AR.Drone has a substantial robotics community following. The AR.Drone has a full suite of MEMS inertial sensors: a three-axis accelerometer, a two-axis gyroscope for measuring pitch and roll, and a yaw angle precision gyroscope. It also features an ultrasound altimeter and two cameras for recording images in the forward-looking and downward-looking directions [12].

![Fig. 2. The Parrot AR.Drone used for testing.](image)

The downward-looking camera was used to detect AprilTags acting as landmarks in our environment. The AR.Drone itself fuses measurements from its inertial sensors and cameras to provide estimates of velocities and Euler angles of the robot. The velocities reported by the AR.Drone are not body-frame velocities, but rather velocities oriented with the yaw angle of the body-frame and always parallel with the ground. The data available from the AR.Drone that we used in our experiments is reported by a single LCM datatype. An overview of this datatype is given in Table I.

The front-end converts the measurements from the AR.Drone into relative poses (quadrotor odometry). Both the camera images and state measurements are transmitted by the AR.Drone over its own wireless ad-hoc network and passed into the front-end using LCM.

VII. EXPERIMENTS

We applied our SLAM implementation to simulated and real-world data. In both cases, we evaluated batch least-squares SLAM, incremental smoothing and mapping, and incremental smoothing and mapping with variable reordering.

A. Simulation

To evaluate the speed of the three different methods, we used the simulator developed by the University of Michigan APRIL lab. Before each test, we seeded the random number generator to ensure the exact same data was being generated. The simulator represents a 2D world, so we set the \( z \) position, roll, and pitch observations to be zero with high certainty. We chose to do this instead of using 2D nodes and factors to see the system grow at the same rate of a full 6-DOF application.

The three different methods of SLAM that were evaluated had enormous speed differences. Table II shows the total simulation time for two loops of the simulator, which ensured that there were many loop closures.

Figure 3 shows the computation time spent at each update step for the different methods. Notice the spikes at integer multiples of 100 for the incremental methods. This is expected because it coincides with our batch update, optional
TABLE II
TOTAL SIMULATION TIME FOR DIFFERENT SLAM METHODS

<table>
<thead>
<tr>
<th>Method</th>
<th>Total Simulation Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental + Reorder</td>
<td>18.81</td>
</tr>
<tr>
<td>Incremental</td>
<td>57.04</td>
</tr>
<tr>
<td>Batch Solve</td>
<td>912.96</td>
</tr>
</tbody>
</table>

Fig. 3. The time per step for different SLAM solve methods. The incremental method with reordering (shown in red) is by far the fastest.

variable reordering, and re-linearization. These results confirmed our suspicion that we would need incremental updates as well as periodic variable reordering in order to achieve real-time speeds when working with real-world data.

We also ran our back-end on standard datasets to assess the computational performance and accuracy of our implementation. For the Manhattan dataset [13], our final normalized $\chi^2$ error value was 1.0399, which is comparable to that of iSAM’s. On a consumer-grade computer, we finished the computations in 253 seconds. For the Victoria Park dataset [14], our final normalized $\chi^2$ error value was 0.88614 and the total computation time was 291 seconds. Figure 4 shows these results.

B. Quadrotor Helicopter

By arranging the landmarks in a known orientation, and flying the AR.Drone in a set pattern, we were able to qualitatively evaluate the results of our experiments. Our approach was to arrange 4 landmarks in a square and fly the AR.Drone around each edge twice to gather loop closures. Since we did not use any ground-truth mechanisms, flying in a square allowed us to evaluate the quality of our created map. A picture of one of our test runs can be seen in Figure 5(a).

Our implementation was able to run and process data in real time, providing a map that is qualitatively acceptable. As expected, the back-end corrects for drifts in the robot’s odometry, shown in Figures 5(b) and 5(c).

Without knowing the ground-truth trajectory, absolute error in odometry and landmark positions are impossible to calculate. Therefore, we used normalized $\chi^2$ to quantitatively evaluate the map produced by SLAM. After running on a nominal data set, we find a final normalized $\chi^2$ error of 0.9719. Because the expected value of a normalized $\chi^2$ is 1, our solution is probabilistically reasonable.

One test run of the AR.Drone took about 60 seconds and produced about 1000 updates. We were able to do all data gathering (reading and publishing LCM messages), pre-processing (video decoding and AprilTag homography), and SLAM solving using a personal laptop with standard specifications in real time.

VIII. Interactive Demonstration

To interactively demonstrate our SLAM back-end, we used a consumer-grade webcam tracking a pre-assigned AprilTag to simulate odometry. Five other unique tags acting as positional landmarks were added to the scene and tracked from a camera being moved by a participant. A screen-capture of the program is shown in Figure 6.

This setup visually explained weighted least-squares in the context of SLAM. The user can induce residuals for certain poses by physically moving a tag once the map has been built. The back-end optimized the graph and displayed it in real time. Watching this unfold in a 3D scene proved to be a creative visualization of the problem, especially for students who may not be familiar with least-squares SLAM.

IX. Conclusion

We have presented the development of an optimized, generic back-end solver for a 6-DOF SLAM problem which is fast enough to allow for real-time deployment on a standard computer. Real-time operation was accomplished by using techniques from iSAM. We have deployed it in a real-world 6-DOF application using known data association with acceptable quality and $\chi^2$ error.
Fig. 4. The final maps and normalized $\chi^2$ values for each step from our SLAM implementation on two datasets. As evidenced by the peaks in (b) and (d), iSAM occasionally suffers from linearization errors. This is an inevitable consequence of the incremental update techniques. These errors are completely removed at the next batch step.

Fig. 5. The map of the world created by the AR.Drone quadrotor with landmarks arranged in square. In (b) the dead reckoning trajectory has significant drift in the $z$ direction (up/down). This drift is corrected when using our SLAM implementation (c).

REFERENCES


