Improving the computational efficiency of ROVIO

S.A. Bahnam[*], C. de Wagter, and G.C.H.E de Croon
Delft University of Technology, Kluyverweg 1, Delft

ABSTRACT

ROVIO is one of the state-of-the-art mono visual inertial odometry algorithms. It uses an Iterative Extended Kalman Filter (IEKF) to align features and update the vehicle state simultaneously by including the feature locations in the state vector of the IEKF. This algorithm is single core intensive, which allows using the other cores for other algorithms, such as object detection and path optimization. However, the computational cost of the algorithm grows rapidly with the total number of features. Each feature adds three new states (a 2D bearing vector and inverse depth), leading to bigger matrix multiplications which are computationally expensive. The main computational load of ROVIO is the iterative part of the IEKF. In this work, we reduce the average computational cost of ROVIO by 40% on an NVIDIA Jetson TX2, without affecting the accuracy of the algorithm. This computational gain is mainly achieved by utilizing the sparse matrices in ROVIO.

1 INTRODUCTION

Visual Inertial Odemetry (VIO) and Simultaneous Localization And Mapping (SLAM) are popular methods to navigate in GPS-denied environments. However, Micro Air Vehicles (MAVs) with extreme Size, Weight, and Power (SWaP) restrictions do not have enough computational power to do onboard loop closure computation. Especially, if the MAV has multiple computational tasks, such as object detection and path planning, it is important that computational effort for VIO is minimal. Moreover, minimal computation time and latency are important for high-speed flight, as in autonomous drone racing.

The most common VIOs are either monocular or stereo. Stereo VIO has the advantage of being able to triangulate features to immediately get a depth estimation for new features. Even though it requires an extra step (stereo matching), compared to mono VIO, it does not have to be computationally more expensive [1]. However, it requires an accurate stereo calibration and adds the weight of an extra camera. Furthermore, for drones with a smaller stereo baseline, the resolution of the depth estimation is also smaller.

Monocular VIO is preferable for MAVs with extreme SWaP constraints, as it requires only a single camera. The state-of-the-art filter-based mono VIOs are ROVIO [2] and MSCKF VIO [3]. ROVIO uses a patch-based direct method to align features and estimate the state in an Iterative Extended Kalman Filter (IEKF). Whereas, MSCKF tracks features and updates the state each time a feature is lost. A disadvantage of MSCKF is that the computational load per frame varies as it only updates if a feature is lost or when a maximum number of camera states are in the buffer. Next, there are optimization-based VIOs, like VINS-mono [4] and OKVIS [5]. However, these are generally computationally more expensive, because they optimize over a window of states.

Of the above-mentioned algorithms, ROVIO is the only algorithm that is single core intensive, which allows using the other cores for other computational tasks. Next, ROVIO is the only direct method, whereas the others are feature-based methods. The advantage of direct methods is that they are able to estimate the motion even in low-texture environments [6]. Furthermore, ROVIO is able to track features on an edge (e.g. line features) due to the initial feature location prediction it receives from the IMU-driven state propagation [2].

ROVIO has been used in various drone applications, ranging from cave exploration [7] to autonomous drone racing [8]. In a drone delivery application [9] ROVIO was considered, but SVO [10] was chosen because it is computationally cheaper and therefore has a smaller computational delay.

In this work we reduce the computational cost of ROVIO without affecting the accuracy. To be more precise we reduce the average computational time by 40% on an NVIDIA Jetson TX2. The main modifications to ROVIO are:

1. We substantially reduce the size of the Jacobian used in the IEKF of the ROVIO algorithm
2. We reduce the computational cost of the prediction step of the IEKF by exploiting the sparsity of the matrices.

The remainder of the article is organized as follows. Firstly, in Section 2 we give a short overview of ROVIO. In Section 3 it is shown what we have modified in ROVIO to reduce the computational cost. Next, the results on the EuRoC and UZH-FPV Drone Racing dataset are shown in Section 4. This is followed by the conclusion in Section 5.

2 RELATED WORK

ROVIO mainly differs from other VIOs by using an IEKF that uses photometric errors of patches as an innovation term in the filter update step. This means that the feature alignment is done simultaneously with the state update. The features

[*]Email address: S.A.Bahnam@tudelft.nl

are included in the state vector where each feature has a 2D bearing vector and an inverse distance parameter. Furthermore, the state vector includes 21 other states: robocentric position, velocity and attitude of IMU (9), accelerometer and gyroscope biases (6), and linear and rotational part of the the IMU-camera extrinsics (6). Therefore, the state vector has a size of $n = 21 + 3 \cdot m$, where m is the maximum number of features (25). The computation for each new image frame can be described in 3 steps. Firstly, the state is predicted using the IMU data between time t and $t-1$. Next, for each feature, the state vector is iteratively updated until the feature is matched (or discarded as an outlier). Lastly, new features are added when the number of tracked features drops below a certain threshold (of $0.8 \cdot m$).

Adding new features has a low average computational cost, but can result in computational peaks. The reason for this is that only new features are added when the number of features is dropped below 80% maximum number of features. Therefore, in most frames no features need to be detected, which result in a low average computation time. However for frames which adds new features, it requires additional computation for feature detection next to the IEKF computation, which result in a computational peak in ROVIO. When looking only to the new feature selection part of ROVIO, the computation of the Shi-Tomasi score [11] of all candidates is the computational most expensive part.

In most VIO evaluations on benchmarks, the VIO performance is not affected by computational peaks, because a camera buffer is used in the VIO. However, the control performance of applications, such as autonomous drone racing, is affected by delay. Therefore, one would like to decrease the computational delay. For this reason, one could decide to reduce the camera buffer in such applications. However, this comes at the cost that the accuracy of the VIO may decrease or in worst case that the filter diverges if the peak processing time is too high. In [8] it is reported that ROVIO was processed at 35 Hz, but the total delay was 130 ms. It is also mentioned that the main contribution to the total delay was interfacing with the camera and running the VIO. In this work we reduce the computational cost of ROVIO to increase the execution frequency and reduce delays..

3 METHOD

3.1 *Prediction step of IEKF*

In the prediction step of the IEKF, the states are propagated and the covariance is estimated using the IMU data. ROVIO uses the average IMU data between two frames in order to compute the prediction step only once every frame and reduce the computational cost without a notable performance loss. The covariance matrix computation is computationally most expensive as it involves $n \times n$ matrix multiplications and can be calculated with Equation 1.

$$P_k^- = F_{k-1} \cdot P_{k-1}^+ \cdot F_{k-1}^T + G_{k-1} \cdot W_{k-1} \cdot G_{k-1}^T \tag{1}$$

, in which P is the covariance matrix, $k-1$ and k are before and after the state prediction, respectively. F is the system transition matrix, G is the noise input matrix and W is the continuous time noise covariance. Each matrix here has a size of $n \times n$, where n is equal to $21 + 3 \cdot m$, where m is the maximum number of features. Therefore, the multiplication of the matrices is computationally expensive. However, matrix F, G, and W are all sparse matrices. W is a constant diagonal matrix, where all entries are the estimated noise variances from the input. F and G can be found in Equation 2 and 3, respectively. For more details how F and G are constructed, see [2].

$$F = \begin{bmatrix} I_{3\times3} & & & \\ & B_{15\times12} & 0_{15\times6} & 0_{21\times f} \\ 0_{n-3\times3} & 0_{6\times12} & I_{6\times6} & \\ & B_{f\times12} & 0_{f\times6} & BD_{f\times f} \end{bmatrix} \tag{2}$$

$$G = \begin{bmatrix} & 0_{3\times3} & & \\ D_{12\times12} & B_{3\times3} & 0_{15\times6} & 0_{21\times f} \\ & 0_{6\times3} & & \\ & B_{3\times3} & & \\ 0_{n-12\times12} & 0_{6\times3} & D_{6\times6} & \\ & B_{f\times3} & 0_{f\times6} & BD_{f\times f} \end{bmatrix} \tag{3}$$

, in which B is a block matrix, BD a block diagonal (3×3 block diagonal for F and G), D is a diagonal matrix and f is the number of all feature states ($3m$).

3.2 *Iterative update of IEKF*

The difference between an EKF and IEKF is that the update step is performed multiple (j) iterations until the update is small/converged, the measurement is discarded (detected as outlier), or the maximum number of iterations (20 for the original settings) is reached. ROVIO iteratively updates the state vector for each feature candidate (i) separately.

Each iteration requires big matrix multiplications to calculate the Jacobian, Kalman gain, update vector, and the covariance matrix of the state update. The big matrices are sparse, however the used MatrixXd from the Eigen library [12] in ROVIO does not perform sparse matrix multiplication efficiently.

A feature candidate is generated using the the predicted feature location and its covariance. This is done at least once and a maximum of 3 times (for the original settings) per detected/tracked feature on the previous frame. Each time the state vector is updated as in Equation 4.

$$x = x + P_{i_{n\times n}}^- \cdot -c_{i_{2\times n}}^T \cdot Py_i \cdot dy_i \tag{4}$$

, in which $P_{i_{n\times n}}$ is the state covariance matrix, c_i is the pixel location of the feature inserted in a $2 \times n$ zero matrix, which is done to get the correct size for matrix multiplication. dy_i is a 2×1 vector, which depends on the eigenvalues of the Py_i. And Py_i is a 2×2 matrix and can be calculated with Equation 5.

$$Py_i = -c_{i_{2\times n}} \cdot P^-_{i_{n\times n}} \cdot -c^T_{i_{2\times n}} \tag{5}$$

Since c_i only contains information about the pixel location we can save computation time. Depending on which feature is processed, we use a certain 2×2 block of those matrices. Therefore, we can modify Equation 4 and 5 to Equation 6 and 7, respectively.

$$x = x + P^-_{i_{n\times 2}} \cdot -c^T_{i_{2\times 2}} \cdot Py_i \cdot dy_i \tag{6}$$

$$Py_i = -c_{i_{2\times 2}} \cdot P^-_{i_{2\times 2}} \cdot -c^T_{i_{2\times 2}} \tag{7}$$

Next, a multilevel patch is extracted (a patch of 6×6 on two image levels) for each feature candidate. The Jacobian of the feature is calculated based on the multilevel patch gradient, adaptive light condition parameters and the (distorted) feature pixel location. The gradient of the previous image is used, therefore the multilevel patch gradient is constant for all iterations per feature in a single frame. However, the original ROVIO algorithm unnecessarily recomputes the gradient at every iteration of each feature per frame (i, j, k). We store the gradient of the patch of a feature on the first iteration of each frame. Therefore, we only need to calculate the gradient once per feature per frame (i, k).

In ROVIO the 2x2 Jacobian is inserted in a $2 \times n$ zero matrix, with n being the size of the state vector. This is done to allow matrix multiplications with the $n \times n$ covariance matrix. However, this is very inefficient as many zero multiplications are involved. Therefore, we extract the useful information in the 2×2 block Jacobian for calculations. This allows us to use smaller blocks for the covariance matrix as well. The original code of ROVIO calculates the 2×2 matrix $Py_{i,j}$ using Equation 8. We propose to calculate $Py_{i,j}$ as in Equation 9.

$$Py_{i,j} = H_{i,j_{2\times n}} \cdot P^-_{i_{n\times n}} \cdot H^T_{i,j_{2\times n}} + R \tag{8}$$

$$Py_{i,j} = H_{i,j_{2\times 2}} \cdot P^-_{i_{2\times 2}} \cdot H^T_{i,j_{2\times 2}} + R \tag{9}$$

, in which $P^-_{i_{2\times 2}} = P^-_i.block(21 + 3i, 21 + 3i, 2, 2)$ is a 2×2 block of the covariance matrix of the prediction step, where i is the index of the processed feature in the state vector. $H_{i,j_{2\times 2}}$ is the Jacobian without zeros. R is the measurement noise matrix with size 2×2. Note, that the computational time of our proposed method is independent of the size of the state vector and the original method is $O(n^3 + n^2)$. Next, the Kalman gain (of size $n \times 2$) is calculated with the original code as in Equation 10. We propose to modify it to Equation 11.

.

$$K_{i,j} = P^-_{i_{n\times n}} \cdot H^T_{i,j_{2\times n}} \cdot \left(Py_{i,j_{2\times 2}}\right)^{-1} \tag{10}$$

$$K_{i,j} = P^-_{i_{n\times 2}} \cdot \left(H^T_{i,j_{2\times 2}} \cdot \left(Py_{i,j_{2\times 2}}\right)^{-1}\right) \tag{11}$$

Since K_i, j is of size $n \times 2$ we have to use all rows of our covariance matrix, but we only need two columns. Furthermore, we first do the 2×2 matrix multiplication as this will save computational work. This reduce the computational cost from $O(n^3 + n)$ to $O(n)$.

The original code uses Equation 12 to compute the update vector and we propose Equation 13 instead.

$$\Delta x_{i,j} = (x^-_i \boxminus x^+_{i,j}) - K_{i,j} \cdot \left(z_{i,j} + H_{i,j_{2\times n}} \cdot (x^-_{i_{n\times 1}} \boxminus x^+_{i,j_{n\times 1}})\right) \tag{12}$$

$$\Delta x_{i,j} = (x^-_i \boxminus x^+_{i,j}) - K_{i,j} \cdot \left(z_{i,j} + H_{i,j_{2\times 2}} \cdot (x^-_{i_{2\times 1}} \boxminus x^+_{i,j_{2\times 1}})\right) \tag{13}$$

, in which the $\boxminus$ is the boxminus operator. The computational cost is reduced from $O(n^2 + n)$ to $O(n)$.

Furthermore, the Jacobian, H_j, and the measurement, z_j, (difference of the multilevel patches) are computed twice per iteration in the original ROVIO code. This is modified, such that it is only calculated once per iteration.

4 Results

We test the modified and original ROVIO on the EuRoC [13] dataset and on the UZH-FPV Drone Racing Dataset [14]. We run the algorithms on an NVIDIA Jetson TX2, which has a dual-core Denver 2 64-Bit CPU and a quad-core ARM Cortex-A57. We use the tool of [15] to get the RMS of the APE. All trajectories are aligned with the ground truth, optimizing position and yaw only, which is proposed for mono VIOs in [15].

Firstly, we show the difference in computation time, due to the modifications. Next, we show that the modified ROVIO does get a similar trajectory estimation when processing all frames on the EuRoC dataset. Finally, we reduce the camera buffer and compare the accuracy of the original and modified algorithm on the UZH-FPV Drone Racing Dataset.

4.1 EuRoC

We compare the original ROVIO and modified ROVIO on the EuRoC dataset. We exclude sequence Machine Hall 2, because ROVIO (with original parameters) diverges on this sequence. Furthermore, we exclude the first 21 seconds of the sequence Machine Hall 1, in order to ensure ROVIO initializes correctly (and similarly) for the original and modified

Table 1: RMS of the APE of the original and modified ROVIO on the EuRoC dataset after aligning position and yaw with the ground truth. In bold we show when the modified ROVIO matches the original ROVIO either on the laptop or TX2.

		MH01[1]	MH03	MH04	MH05	V101	V102	V103	V201	V202	V203
Laptop	Original	0.346	0.449	0.819	1.307	0.161	0.133	0.225	0.239	0.346	0.249
	Modified	**0.346**	**0.449**	**0.819**	**1.307**	**0.161**	0.196	**0.176**	**0.242**	**0.394**	**0.251**
TX2	Original	0.346	0.398	0.819	1.307	0.156	0.175	0.176	0.242	0.394	0.251
	Modified	**0.346**	**0.449**	**0.819**	1.311	**0.161**	**0.175**	**0.225**	**0.242**	**0.394**	0.237

version. Furthermore, we check for both algorithms that they initialize at the exact same (IMU) timestamp and process the same number of frames.

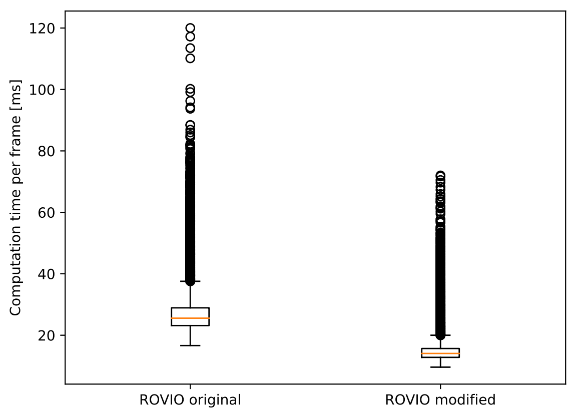

Figure 1: Computation time per frame on the EuRoC dataset (all sequences except MH02) on an NVIDIA Jetson TX2

In Figure 1 the computation time on an NVIDIA Jetson TX2 can be found for the original and modified ROVIO. It can be seen that the average and the maximum computational time per frame is reduced. The original algorithm has an average computation time per frame of 25.6 ms and the modified algorithm this is reduced to 14.1 ms. It can be seen that the modified algorithm still have computational peaks. Those mainly correspond to frames where new features are added. The main computational cost of the feature detection is the computation of the Shi-Tomasi score of the (usual) many detected candidates.

In Table 1 it can be seen that the accuracy sometimes differs when we run the original and modified algorithm. Also it differs when we run the same algorithm on different hardware. The reason for this are rounding errors, which sometimes can lead to a feature at the image border being seen as ”out of frame”. This may result in different features being detected and tracked, which results in a change in accuracy. However, it can be seen that the modified algorithm always has the same accuracy as the original algorithm either on the laptop or Nvidia Jetson TX2. Except for V102 on the Laptop and V203 on the TX2.

[1]The first 21 seconds are excluded of this sequence

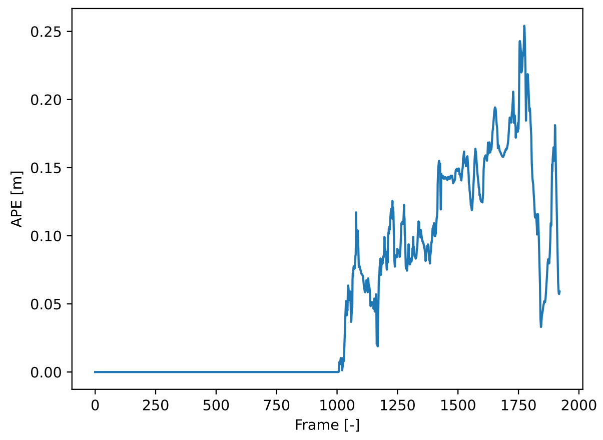

Figure 2: APE of the modified ROVIO w.r.t. the original ROVIO on sequence V203 of the EuRoC dataset on an NVIDIA Jetson TX2

The difference on V102 on the laptop is quite big. However, when we change the feature detection threshold from 0.8 to 0.999 we get and average RMSE of 0.135 m for sequence V102 for modified and original algorithms on both hardware. By changing this threshold new features are added every time a feature is inactive (not tracked), instead of when the number of tracked features drops below 80% of the maximum number of features. Therefore, the change in feature set is reduced.

The modified ROVIO on the TX2 gets a slightly lower RMSE on sequence V203, but the difference is small. In Figure 2 the difference between the original and modified algorithm on the TX2 are shown. The reason for the difference is that at the 1006th frame one feature is lost in the modified algorithm, however that same feature is tracked with the original algorithm. The reason for this is a rounding error when computing Equation 7, if we use the original Equation 5, the modified algorithm matches the original ROVIO on sequence V203. However, it results in that other sequences will experience a similar rounding error (in another part of the modified algorithm).

Table 2: Total frames processed per sequence using a camera buffer of 1 on the UZH-FPV Drone Racing Dataset with an NVIDIA Jetson TX2.

	03	05	06	07	09	10
Original	2124/2552	4073/4162	1589/1970	2850/3158	2051/2068	2082/2127
Modified	**2437/2552**	**4157/4162**	**1935/1970**	**3142/3177**	**2066/2068**	**2124/2127**

Table 3: RMS of the APE of the original and modified ROVIO on the UZH-FPV dataset after aligning position and yaw with the ground truth.

	03	05	06	07	09	10
Original	1.47	0.72	0.50	1.08	**0.38**	0.62
Modified	**0.97**	**0.59**	**0.48**	1.08	0.52	**0.58**

In order to verify that the modified matrix multiplication is done correctly, we use the Frobenius norm of the matrices for a fuzzy comparison (Eigen::isApprox) as in Equation 14.

$$||V - W||_F = p \cdot min(||V||_F, ||W||_F) \qquad (14)$$

, in which V is the matrix calculated using the proposed equations, W is the result using the original matrix multiplication from ROVIO. We set p to 10^{-12}, because this is the default value for a fuzzy comparison with double precision matrices. We do this test for all proposed modifications separately. All modified equations pass the test, except the update vector calculation from Equation 13. In all sequences it returns 0.1% false or less. This is because the magnitude of the update vector is sometimes small (when the prediction is close to the measurement), which results in a very strict comparison.

4.2 *UZH-FPV Drone Racing Datset*

We run the Original and modified version of ROVIO as well on the UZH-FPV dataset [14]. We are using sequence 03, 05, 06, 07, 09 and 10 of the forward facing camera indoor, because those sequences contain the ground truth of the trajectory. We reduce the camera state buffer to 1, which means that a frame (the oldest) is dropped when the algorithm is not fast enough to process two consecutive frames. We do the evaluation only on the NVIDIA Jetson TX2, because the laptop is fast enough to process all frames. This type of evaluation would be realistic for cases where the state estimation delay is of importance. Which could be for example autonomous drone racing, because a delay in the state estimation affects the control performance. We do not change any other parameters of the algorithm.

In Figure 3 the computation time on the UZH-FPV dataset can be seen. The average computation time for the original algorithm is 31.5 ms and for the modified 17.4 ms. Even though, the image resolution is smaller on the UZH-FPV dataset (640 × 480) than on EuRoC (752 × 480), the computation time is higher on the UZH-FPV dataset. We think this is because the motion on the UZH-FPV is bigger, which

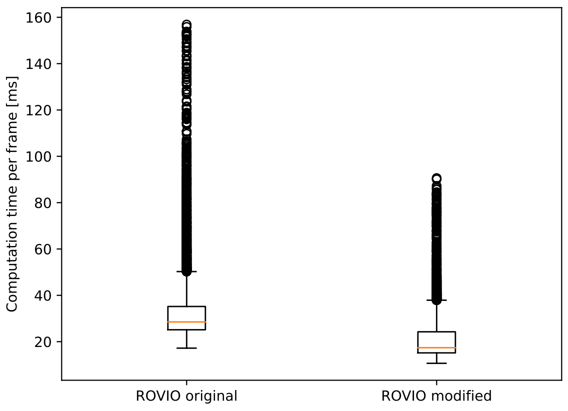

Figure 3: Computation time per frame on (all sequences of) the FPV dataset on an NVIDIA Jetson TX2

requires to use more iterations in the IEKF to align the features.

In Table 2 it shows how many frames out of the total frames are processed for the modified and original ROVIO. Note, that the full (rosbag) sequence is used here. It can be seen that fewer frames are thrown away for the modified ROVIO than the original ROVIO. The reason for this is the computational gain from the sparse matrix operations. It can also be seen that almost all frames are processed for the modified algorithm. The missed frames are expected to come from frames where new features are added, because this adds additional computation in a frame.

The RMSE of the original and modified ROVIO can be found in Table 3. Again we align the the estimated trajectory with the ground truth by optimizing position and yaw only. Note that for the UZH-FPV dataset that the ground truth is only available for part of the sequences. The modified algorithm is more accurate for most sequences, which is expected since it was able to process more frames. The difference in accuracy is most visible in sequence 03. In this sequence there is a big difference in the number of processed frames. However, processing more frames does not guarantee a lower RMSE. On sequence 07 the RMSE of the modified ROVIO is the same as the original algorithm, but the modified algorithm processed almost 300 more frames. On sequence 09, the original algorithm has even a smaller RMSE, however sequence 09 the original ROVIO did not drop many frames either.

5 CONCLUSION

We have made ROVIO computationally more efficient, mainly by utilizing the sparse matrices. The accuracy is the same as the original ROVIO, but requires 40% less computation time on an NVIDIA Jetson TX2. The computational gain depends on the number of features used, because the computational cost of the modified ROVIO is less dependent on the size of the state vector compared to the original ROVIO.

Furthermore, it was shown that the modified ROVIO has a higher accuracy when using a camera buffer of one. However, the difference was small for most sequences. We expect that the difference is bigger when running on a computationally more limited device or when increasing the maximum number of features.

The computational peaks are reduced as well, but they are still there. To mitigate the computational peaks, we recommend do modify the feature selection method. To be more specific: reducing the number of feature candidates of which the Shi-Tomasi score is computed. This could for example be done by using a feature mask around tracked features or using an adaptive FAST threshold. However, this will affect the accuracy of ROVIO and therefore we have not included this in this work.

REFERENCES

[1] Ke Sun, Kartik Mohta, Bernd Pfrommer, Michael Watterson, Sikang Liu, Yash Mulgaonkar, Camillo Taylor, and Vijay Kumar. Robust stereo visual inertial odometry for fast autonomous flight. *IEEE Robotics and Automation Letters*, PP, 11 2017.

[2] Michael Bloesch, Michael Burri, Sammy Omari, Marco Hutter, and Roland Siegwart. Iterated extended kalman filter based visual-inertial odometry using direct photometric feedback. *The International Journal of Robotics Research*, 36:1053–1072, 09 2017.

[3] Anastasios I. Mourikis and Stergios I. Roumeliotis. A multi-state constraint kalman filter for vision-aided inertial navigation. In *2007 IEEE International Conference on Robotics and Automation, ICRA'07*, Proceedings - IEEE International Conference on Robotics and Automation, pages 3565–3572, November 2007.

[4] T. Qin, P. Li, and S. Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34(4):1004–1020, 2018.

[5] Stefan Leutenegger, Simon Lynen, Michael Bosse, Roland Siegwart, and Paul Furgale. Keyframe-based visual-inertial odometry using nonlinear optimization. *The International Journal of Robotics Research*, 34, 02 2014.

[6] Guoquan Huang. Visual-inertial navigation: A concise review. *2019 International Conference on Robotics and Automation (ICRA)*, pages 9572–9582, 2019.

[7] Mihir Dharmadhikari, Huan Nguyen, Frank Mascarich, Nikhil Khedekar, and Kostas Alexis. Autonomous cave exploration using aerial robots. In *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 942–949, 2021.

[8] Philipp Foehn, Dario Brescianini, Elia Kaufmann, Titus Cieslewski, Mathias Gehrig, Manasi Muglikar, and Davide Scaramuzza. Alphapilot: autonomous drone racing. *Autonomous Robots*, 46, 01 2022.

[9] Gino Brunner, Bence Szebedy, Simon Tanner, and Roger Wattenhofer. The urban last mile problem: Autonomous drone delivery to your balcony. In *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1005–1012, 2019.

[10] Christian Forster, Zichao Zhang, Michael Gassner, Manuel Werlberger, and Davide Scaramuzza. Svo: Semidirect visual odometry for monocular and multicamera systems. *IEEE Transactions on Robotics*, 33(2):249–265, 2017.

[11] Jianbo Shi and Tomasi. Good features to track. In *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 593–600, 1994.

[12] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[13] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 35, 01 2016.

[14] Jeffrey Delmerico, Titus Cieslewski, Henri Rebecq, Matthias Faessler, and Davide Scaramuzza. Are we ready for autonomous drone racing? the uzh-fpv drone racing dataset. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 6713–6719, 2019.

[15] Zichao Zhang and Davide Scaramuzza. A tutorial on quantitative trajectory evaluation for visual(-inertial) odometry. In *IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IROS)*, 2018.

APPENDIX A: CODE

A branch of the updated code can be found at: http://www.github.com/sbahnam/rovio2