Improving the Computational Efficiency of ROVIO

Bahnam, S. A.; De Wagter, C.; De Croon, G. C.H.E.

DOI
10.1142/S2301385024410012

Publication date
2024

Document Version
Accepted author manuscript

Published in
Unmanned Systems

Unmanned Systems, Vol. 2, No. 11 (2022) 1–9

Improving the Computational Efficiency of ROVIO

S.A. Bahnam[1]*, C. De Wagter[1], G.C.H.E de Croon[1]

[1] *Faculty of Aerospace Engineering, Delft University of Technology, 2629 HS Delft, The Netherlands*

ROVIO is one of the state-of-the-art monocular visual inertial odometry algorithms. It uses an Iterative Extended Kalman Filter (IEKF) to align visual 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 the other cores to be used for other algorithms, such as object detection and path optimization. However, the computational cost of the algorithm grows rapidly with the maximum number of features to track. Each feature adds three new states (a 2D bearing vector and inverse depth), leading to bigger matrix multiplications that are computationally expensive. The main computational load of ROVIO is the iterative update step 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 exploiting the sparse matrices in ROVIO. Furthermore, we reduce the computational peaks by pre-selecting new features based on their already calculated FAST score. The combination of both modifications allows us to run ROVIO on the computationally restricted Raspberry Pi Zero 2 W.

Keywords: Visual Inertial Odometry; Computational efficiency; ROVIO.

1. Introduction

Visual Inertial Odometry (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 easily carry enough computational power to do onboard loop closure computations. Especially, if the MAV has multiple computational tasks, such as object detection and path planning, the computational effort for VIO must be minimized. Moreover, minimal computation time and latency are also important for high-speed flight such as autonomous drone racing.

The most common VIOs use either monocular or binocular, i.e. using stereo vision. 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 accurate stereo calibration and adds the weight of an extra camera. Furthermore, for drones with a smaller stereo baseline, the maximal observable depth 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). 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. Moreover, ROVIO is the only direct method, whereas the others are feature-based methods. The advantage of direct methods is that they can estimate the motion in low-texture environments [6]. Furthermore, ROVIO can track features on an edge (e.g. line features) due to the initial feature location prediction that is obtained 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]. Brunner et al. considered using ROVIO in a drone delivery application [9], but ended up choosing SVO [10] because it is computationally cheaper and therefore has

*E-mail: S.A.Bahnam@tudelft.nl

a smaller computational delay. This illustrates the importance of computational load for real-time applications.

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. Furthermore, we reduce the computational peaks, which slightly affects the accuracy. However, this allows us to run ROVIO on a Raspberry Pi (RPI) Zero 2 W. The RPI Zero 2 W is a computationally restricted platform with a small size of 65×30 mm and a weight of 11 g. The main modifications to ROVIO are:

(i) We substantially reduce the size of the Jacobian used in the IEKF of the ROVIO algorithm,
(ii) We reduce the computational cost of the prediction step of the IEKF by exploiting the sparse matrices,
(iii) We use the FAST score instead of the more expensive Shi-Tomasi score [11] for feature selection and we preselect feature candidates with the highest FAST score when many feature candidates are detected.

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 datasets 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 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 the linear and rotational part of 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, the state vector is iteratively updated until the feature is matched (or discarded as an outlier) for all features. Lastly, new features are added when the number of tracked features drops below a certain threshold (of $0.8 \cdot m$).

In most frames, no features need to be detected, which results in a low average computation time. However, for frames that add new features, it requires additional computation for the feature detection next to the IEKF computations. This results in a computational peak in ROVIO.

In most VIO evaluations on benchmarks, the VIO performance is not affected by computational peaks, because a camera buffer is used in the pipeline. However, the control performance of real-time 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. In the worst case, the filter could even diverge 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. We suspect that the difference between the delay and FPS might also be due to the computational peaks, because ROVIO has a varying computational cost per frame. In this work, we reduce the computational cost of ROVIO to increase the execution frequency and reduce delays.

Others use the GPU to (partially) accelerate the VIO and increase the execution frequency. In [12] they speed up VINS-mono [4] by implementing a parallelization scheme. The optical flow tracking requires 1.9 times less computation time and the marginalization is $1.5 - 1.7$ times faster on an NVIDIA Jetson TX2. In [13] they have a GPU-accelerated feature detector and tracker that runs 1000+ FPS on the NVIDIA Jetson TX2. For the backend, they run ICE-BA [14], which is a lightweight bundle adjustment, on the CPU. The VIO runs around 200 FPS on the NVIDIA Jetson TX2. There are also learned VIO neural networks, like VIOLearner [15]. However, currently, they run either on big GPUs or have a slow inference speed. For example, VIOLearner requires 27 ms on an NVIDIA Titan X GPU.

3. Method

3.1. *ROVIO's prediction step*

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

$$P_k^- = F_{k-1} \cdot P_{k-1}^+ \cdot F_{k-1}^T + G_{k-1} \cdot W_{k-1} \cdot G_{k-1}^T, \qquad (1)$$

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 $21 + 3 \cdot \ m$, and m is the maximum number of features. Therefore, the multiplication of the matrices is computationally expensive. However, matrices 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.

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

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

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 ($3m$). For more details on how F and G are constructed, see [2]. Note that $B_{f\times12}*$ contains 6 zero columns and $B_{15\times12}*$ contains zero entries, but are written such for simplification of the notation.

3.2. *ROVIO's iterative update step*

The difference between an EKF and IEKF is that the update step is performed multiple (j) iterations until the update is converged, the measurement is discarded (detected as an 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 covariance matrix of the state update. The big matrices are sparse, however, the used MatrixXd from the Eigen library [16] in ROVIO does not perform sparse matrix multiplication efficiently.

A feature candidate is initialized using the predicted feature location and its covariance. This is done at least once and a maximum of 9 times (for the original settings) per detected/tracked feature on the previous frame. Each time the state vector is initialized as in Equation 4.

$$x = x + P^-_{i_{n\times n}} \cdot -c^T_{i_{2\times n}} \cdot S_i^{-1} \cdot dy_i, \quad (4)$$

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 S_i. And S_i is a 2×2 matrix and can be calculated with Equation 5:

$$S_i = -c_{i_{2\times n}} \cdot P^-_{i_{n\times n}} \cdot -c^T_{i_{2\times n}}. \quad (5)$$

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

$$x = x + P^-_{i_{n\times2}} \cdot -c^T_{i_{2\times2}} \cdot S_i^{-1} \cdot dy_i, \quad (6)$$

$$S_i = -c_{i_{2\times2}} \cdot P^-_{i_{2\times2}} \cdot -c^T_{i_{2\times2}}. \quad (7)$$

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.

In ROVIO the 2×2 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 $S_{i,j}$ using Equation 8. We propose to calculate $S_{i,j}$ as in Equation 9.

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

$$S_{i,j} = H_{i,j_{2\times2}} \cdot P^-_{i_{2\times2}} \cdot H^T_{i,j_{2\times2}} + R, \quad (9)$$

in which $P^-_{i_{2\times2}} = P_i^-.block(21+3i, 21+3i, 2, 2)$ is a 2×2 block of the covariance matrix of the prediction step, where i is the index of the update feature. $H_{i,j_{2\times2}}$ 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, whereas the original method is $\mathcal{O}(n^3+n^2)$. Next, the Kalman gain (of size $n\times2$) is calculated with the original code as in Equation 10. We propose to modify it to Equation 11.

$$K_{i,j} = P^-_{i_{n\times n}} \cdot H^T_{i,j_{2\times n}} \cdot \left(S_{i,j_{2\times2}}\right)^{-1}, \quad (10)$$

$$K_{i,j} = P^-_{i_{n\times2}} \cdot \left(H^T_{i,j_{2\times2}} \cdot \left(S_{i,j_{2\times2}}\right)^{-1}\right). \quad (11)$$

Since K_i, j is of size $n\times2$ 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 reduces the computational cost from $\mathcal{O}(n^3+n)$ to $\mathcal{O}(n)$.

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

$$\Delta x_{i,j} = (x_i^- \boxminus x_{i,j}^+) - K_{i,j} \cdot \left(z_{i,j} + H_{i,j_{2\times n}} \cdot (x^-_{i_{n\times1}} \boxminus x^+_{i,j_{n\times1}})\right), \quad (12)$$

$$\Delta x_{i,j} = (x_i^- \boxminus x_{i,j}^+) - K_{i,j} \cdot \left(z_{i,j} + H_{i,j_{2\times2}} \cdot (x^-_{i_{2\times1}} \boxminus x^+_{i,j_{2\times1}})\right), \quad (13)$$

in which the $\boxminus$ is the boxminus operator. The boxminus operator takes the difference on a Lie group and maps it to its Lie algebra, a detailed explanation can be found in [17]. The computational cost is reduced from $\mathcal{O}(n^2+n)$ to $\mathcal{O}(n)$.

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.

3.3. *Fast feature selection*

ROVIO detects features when the number of tracked features drops below 80% of the maximum number of features. It uses a FAST(9-16) feature detection on an image pyramid (1/2 and 1/4 image size) to find new feature candidates. A low FAST threshold (of 5) is used, which can detect feature candidates when there is little texture in the image. New features are selected based on the Shi-Tomasi score and the location of the feature candidates, features candidates near tracked features are penalized.

However, the ROVIO's feature selection can cause computational peaks, which makes the filter diverge on computationally limited devices. The reason for the computational peaks is that the feature selection sometimes is computationally too expensive and needs to run after the filter prediction and iterative update step for frames that track too few features. The computation time required highly depends on the number of detected feature candidates, which is very high for images with high texture. A patch is extracted for each feature candidate to calculate the Shi-Tomasi score, which is computationally expensive.

To reduce the computational peaks we make three modifications. Firstly, we use the FAST score instead of the Shi-Tomasi score. The FAST score is already calculated in the used OpenCV implementation of the FAST feature detector. Therefore, we do not require to extract a patch and calculate the Hessian matrix. FAST features generally have a lower quality than SIFT [18], SURF [19], ORB [20] or Shi-Tomasi [11] features, but FAST features require less computation. Therefore, the computational peaks will be smaller when using FAST features, which is beneficial when ROVIO runs on a computationally limited device.

Secondly, we only detect features on the 1/4 image size to reduce the feature detection time and the number of feature candidates. Lastly, if the FAST feature detector detects more than 250 feature candidates, we select 150 features with the highest FAST score as feature candidates. This allows us to keep the FAST threshold low, and therefore detect feature candidates in low-textured images. Furthermore, it makes the computational cost of the feature selection process more consistent.

4. Results

We test the original ROVIO, sparse ROVIO and fast ROVIO, which in sparse ROVIO uses the sparse matrices of Section 3.1 and Section 3.2 and fast ROVIO uses the sparse matrices and fast feature selection of Section 3.3. We use the EuRoC dataset [21] and the UZH-FPV drone racing dataset [22] for evaluation of the modifications. We run the algorithms on an NVIDIA Jetson TX2, which has a 2 GHz dual-core Denver 2 64-Bit CPU and a quad-core ARM Cortex-A57. Furthermore, we show that fast ROVIO can run on computationally limited devices, such as the RPI Zero 2 W. The RPI Zero 2 W has a 1 GHz quad-core 64-bit Arm Cortex-A53 CPU and 512 MB RAM with a size of 65×30 mm and a weight of 11 g.

Furthermore, we compare our results on the RPI Zero 2 W on the EuRoC dataset with the original ROVIO on other computational limited devices from [23]. The computationally most limited device that was able to run ROVIO was the ODROID XU4, which has a 32-bit dual quad-core, ARM A7 at 1.5 GHz and ARM A15 at 2.0 GHz. It has 2 GB RAM and a size of 83×58 mm with a mass of 59 g.

We use the tool of [24] to get the RMS of the APE. All trajectories are aligned with the ground truth, optimizing the position and yaw only, which is proposed for mono VIOs in [24]. Except, for the comparison with [23] we use a 7DOF alignment in order to have a fair comparison.

4.1. *EuRoC*

We compare the original ROVIO, sparse ROVIO and fast ROVIO on the EuRoC dataset. We are using the original settings from ROVIO with the only modification that we increase the prediction noise of the velocity estimation from $4{\cdot}10^{-6}$ to $4{\cdot}10^{-5}$, similarly to [23]. Furthermore, to compare the accuracy of the different ROVIOs, we ensure that each ROVIO version initializes at the same (IMU) timestamp and processes the same number of frames on the laptop and NVIDIA Jetson TX2.

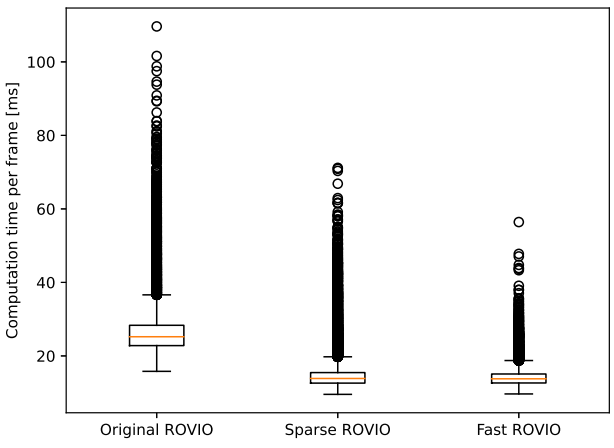

Fig. 1. Computation time per frame on the EuRoC dataset on an NVIDIA Jetson TX2.

Table 1. RMS of the APE in meters after 4DOF (position-yaw) alignment with the ground truth trajectory when processing all frames on the EuRoC dataset.

Machine	Algorithm	MH01	MH02	MH03	MH04	MH05	V101	V102	V103	V201	V202	V203
Laptop	Original	**0.189**	**0.298**	0.437	0.601	**0.855**	**0.113**	0.165	0.096	0.148	0.215	**0.126**
	Sparse	**0.189**	0.412	0.399	0.601	**0.855**	**0.113**	0.165	0.096	0.148	0.215	0.129
	Fast	0.310	0.370	**0.332**	**0.373**	4.508	0.127	**0.143**	**0.092**	**0.098**	**0.139**	0.159
NVIDIA Jetson TX2	Original	**0.189**	0.507	0.437	0.601	**0.855**	**0.113**	0.165	0.096	0.148	0.215	**0.129**
	Sparse	**0.189**	0.631	0.538	0.601	**0.855**	**0.113**	0.165	0.096	0.148	0.215	0.133
	Fast	0.310	**0.390**	**0.332**	**0.373**	5.222	0.127	**0.143**	**0.092**	**0.098**	**0.139**	0.159

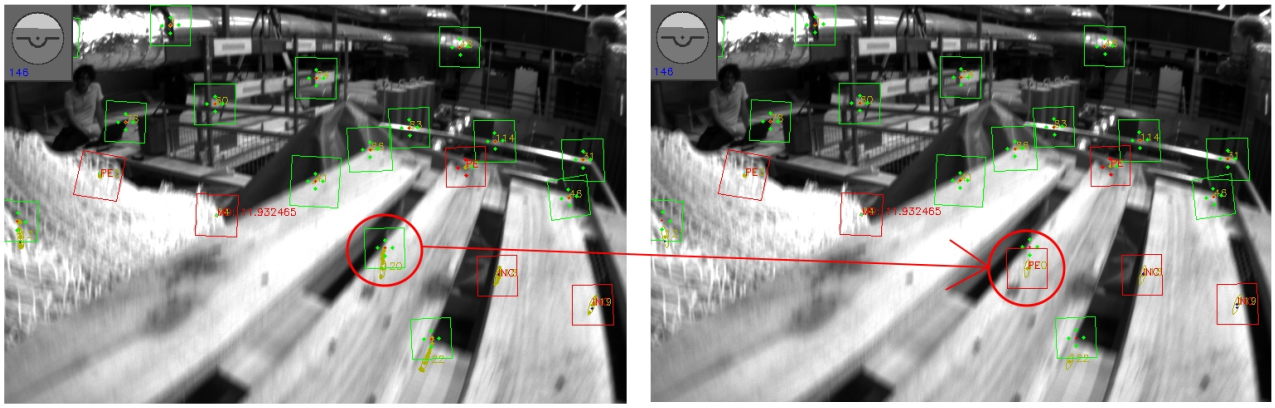

Fig. 2. Original ROVIO (left) tracks a feature on frame 146 while sparse ROVIO (right) loses track of it.

In Fig. 1 the computation time on an NVIDIA Jetson TX2 can be found for the original, sparse and fast ROVIO. It can be seen that the average and the maximum computational time per frame are reduced. The original algorithm has an average computation time per frame of 25.2 ms. Using the sparse matrix multiplications reduces the average computation per frame to 13.9 ms. The fast feature selection barely reduces the average computation time (13.8 ms), the reason for this is that features are not detected at every frame. However, it does reduce the computational peaks from 71 ms to 56 ms. This is because the computational peaks often correspond to the frames that detect new features. And the proposed feature selector is computationally cheaper, especially when many feature candidates are detected.

In Table 1 the Root Mean Square (RMS) of the Absolute Position Errors (APE) are shown for the original ROVIO, sparse ROVIO and fast ROVIO on a Dell XPS 13 laptop and the NVIDIA Jetson TX2. It was expected that the original ROVIO and sparse ROVIO would have the exact same RMSE (RMS of the APE), but this is not the case for MH02, MH03, and V203. Furthermore, we see that MH02 and V203 are different for the original ROVIO on the laptop and the NVIDIA Jetson TX2. The cause of the difference is that a rounding (machine) error can push a feature that is on the border of the requirements to be seen as tracked or lost. This may result in different features being detected and tracked, which results in a change in accuracy.

For example in Machine Hall 03 on the NVIDIA Jetson TX2, the difference between the original and sparse ROVIO is due to a feature that is tracked for the original ROVIO at frame 146 and the same feature is lost in the sparse version, see Fig. 2. Consequently, the rest of the sequence sparse ROVIO will track different features than the original ROVIO. Therefore, they have a different RMSE. Something similar happens in MH02 and V203. The difference on MH02 and MH03 is bigger than V203 because the lost feature happens earlier in the sequence.

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

$$||V - W||_F \leq p \cdot min(||V||_F, ||W||_F) \tag{14}$$

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 in the Eigen Library [16]. 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 false 0.1% of the time or less. We see that in all cases the norm of the update

vector is smaller than 10^{-4}. The reason for this is that the fuzzy comparison becomes more strict and gets sensitive to rounding errors when the update vector is small. When changing p to 10^{-10} it passes the test for Equation 13 on all sequences. This shows that the differences are very small.

When comparing fast ROVIO with original ROVIO and sparse ROVO we see a very similar RMSE in Table 1, except for Machine Hall 05. Fast ROVIO gets a way higher error in the trajectory estimation, which is due to a single feature that is tracked incorrectly at the beginning of the sequence. Between frame 319 and frame 414 when the drone is not moving, it falsely observes feature 403 as coming closer to the camera. Even though all the other features are tracked correctly (to be not moving), the VIO heavily drifts in this case. This is because it cannot estimate depth from a non-moving sequence of images. Therefore, ROVIO estimates that the camera is moving towards feature 403 and the distance of all features to be really far away except for feature 403, see Fig. 3 and 4. From a mathematical perspective, this is a correct prediction because features at an infinite distance do not have an optical flow for translation. In Fig. 5 it can be seen that the depth of most features is recovered after taking off, because of the optical flow information. The remaining path of sequence MH05 with fast ROVIO is estimated correctly, which can be seen in Fig. 6 where we align the trajectory with a 6DOF Umeyama [25] alignment (purely for visualization purposes).

The reason why this only happens for fast ROVIO is that the feature selection of fast ROVIO is mainly based on the FAST score. The FAST score is based on the difference between the center pixel and the 16 pixels on a circle (of radius 3). Whereas, the Shi-Tomasi score is based on the gradient of the image patch. Feature 403 has a low image gradient, therefore a low Shi-Tomasi score. However, it has a brightness contrast in the center of the patch and therefore it has a relatively high FAST score.

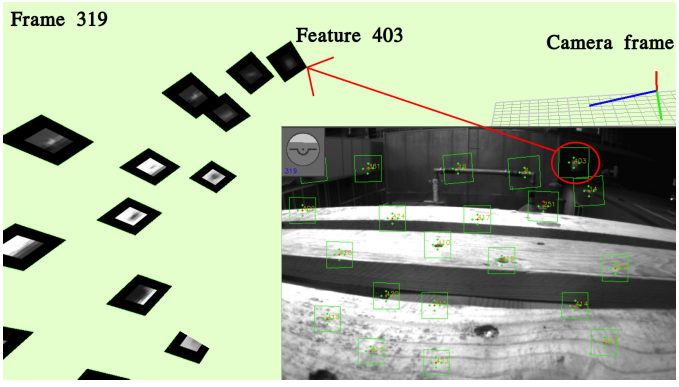

Fig. 3. Features in camera view (right-bottom) and camera frame and features with depth in the world frame of fast ROVIO on Machine Hall 05 frame 319 (laptop).

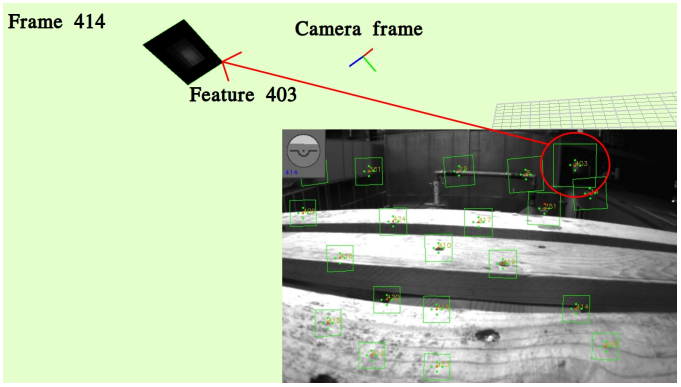

Fig. 4. Feature 403 is falsely observed as moving closer to the camera, which results in the VIO drifting and incorrectly estimating the depth of the features on Machine Hall 05 frame 414.

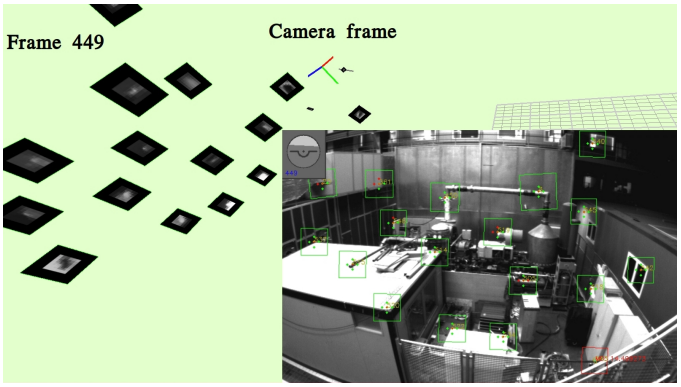

Fig. 5. ROVIO recovers depth estimation of features when the drone/camera starts moving on Machine Hall 05 frame 449.

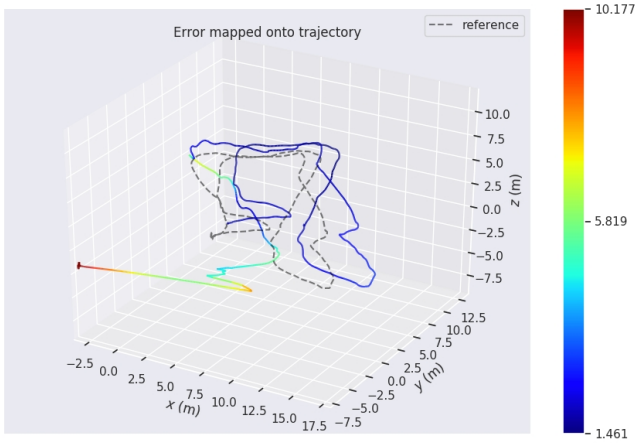

Fig. 6. The estimated trajectory of fast ROVIO on MH05 on the laptop after 6DOF Umeyama alignment shows that the trajectory is initially drifting, but after the camera starts moving it can correctly estimate the trajectory. Using the tool from [26].

Table 2. Average RMS of the APE in meters / successful runs out 5 runs on the RPI running real-time with a 4DOF post-process trajectory alignment.

Algorithm	MH01	MH02	MH03	MH04	MH05	V101	V102	V103	V201	V202	V203
Original-25	X/0	X/0	**0.31**/3	0.93/1	0.65/3	0.16/3	0.17/**5**	0.14/**5**	0.18/4	0.43/1	0.47/4
Sparse-25	X/0	X/0	0.58/3	1.16/2	1.19/3	0.15/**5**	0.18/4	0.20/**5**	X/0	0.39/2	0.22/**5**
Fast-25	X/0	**0.38/3**	0.32/**5**	**0.49/4**	**0.64/4**	**0.14**/4	**0.11/5**	**0.13/5**	**0.10/5**	**0.14/5**	**0.15/5**
Original-15	X/0	X/0	0.42/**5**	0.80/1	**0.86**/4	**0.14/5**	**0.14/5**	X	0.18/**5**	0.28/**5**	**0.17**/2
Sparse-15	0.37/1	X/0	0.46/**5**	1.1/4	1.41/**5**	0.19/**5**	0.20/**5**	0.29/1	X/0	0.82/3	0.29/**5**
Fast-15	**0.33/5**	**0.40/5**	**0.37/5**	**0.50/5**	1.46/**5**	**0.14**/3	0.17/**5**	**0.17/5**	**0.17/5**	**0.15/5**	0.20/**5**

Table 3. Average processed frames of the successful runs on the RPI Zero 2 W running real-time.

Algorithm	MH01	MH02	MH03	MH04	MH05	V101	V102	V103	V201	V202	V203
Original-25	X	X	751	589	717	1095	459	664	795	635	696
Sparse-25	X	X	1568	1086	1355	1970	1038	1422	X	1388	1387
Fast-25	X	**2084**	**1896**	**1446**	**1529**	**2138**	**1217**	**1521**	**1589**	**1608**	**1481**
Original-15	X	X	1735	1380	1434	2333	1228	X	1601	1526	1478
Sparse-15	3041	X	2265	1698	1872	2677	1523	1989	X	1954	1739
Fast-15	**3461**	**2813**	**2650**	**1939**	**2106**	**2847**	**1663**	**2101**	**2162**	**2181**	**1813**
Total frames	*3682*	*3040*	*2700*	*2033*	*2273*	*2912*	*1710*	*2149*	*2280*	*2348*	*1922*

Table 4. RMS of the APE in meters with a 7DOF post-process trajectory alignment of ROVIO on computational limited devices.

Device	MH01	MH02	MH03	MH04	MH05	V101	V102	V103	V201	V202	V203
Intel NUC	**0.21**	0.25	**0.25**	0.49	**0.52**	**0.10**	**0.10**	**0.14**	**0.12**	**0.14**	**0.14**
UP Board	X	X	X	X	X	X	X	X	X	X	X
ODROID-10	0.36	**0.23**	0.58	0.81	0.78	0.15	0.24	0.20	0.15	0.17	0.23
RPI-15-fast	0.30/5	**0.23**/5	0.35/5	**0.44**/5	1.18/5	0.14/3	0.17/5	0.16/5	0.13/5	0.15/5	0.19/5

4.2. *Raspberry Pi Zero 2 W*

We run a 32-bit Raspbian Buster on the RPI Zero 2 W. Next to the limited computational resources, the RPI Zero 2 W has a limited RAM of 512 MB. When running ROVIO and the EuRoC ROSbag with the original settings (25 features) it almost uses the total available RAM. Furthermore, running the original ROVIO on the RPI Zero 2 W results in the images being stored in the camera buffer, since it is too slow to process all frames in real-time. The increasing number of images stored in the camera buffer results in that they are saved in the SWAP memory. Therefore, we modify the camera buffer to a maximum of 1 frame, which means that the oldest frame is dropped when the algorithm is too slow to process two consecutive frames at the camera frame rate. Next to limiting memory usage, it also reduces the delay between the VIO and the real system (since older frames are discarded). To compile ROVIO we add 4 GB of SWAP memory. However, during runtime, we disable SWAP memory to prevent any memory from being stored as SWAP. Without changing the camera buffer and enforcing to use of RAM, the original ROVIO always diverges.

The computation time of ROVIO on the RPI Zero 2 W is bigger than the camera frame rate, which leads to frames being discarded. For each run the computation time (slightly) varies, therefore different frames are discarded when running the same sequence multiple times. For this reason, we run all sequences 5 times for each version of ROVIO. Furthermore, we test ROVIO with 25 (original value) features and 15 features.

In Table 2 we show the average RMS of the APE of all successful runs and the number of successful runs per

sequence. The average number of frames processed per sequence can be found in Table 3. We excluded runs where the filter has diverged, because if the filter diverges it starts to detect new features at each frame and stops tracking features (since the IMU prediction step estimates the features to be out of the image). We see that the original ROVIO can estimate the trajectory 30 out of 55 runs without diverging while processing only $25 - 35\%$ of the frames. Sparse ROVIO processes almost twice as many frames, which is expected since we saw a computational efficiency gain of 40% on the NVIDIA Jetson TX2. However, it had 29 successful runs out of 55, which is 1 less than the original ROVIO. Fast ROVIO had a similar average computational cost as sparse ROVIO on the NVIDIA Jetson TX2, but on the RPI Zero 2 W fast ROVIO process $10 - 30\%$ more frames than sparse ROVIO. This is because it reduces the computational peaks, and frames are mainly discarded at the computational peaks of the algorithm. The additional processed frames allow the algorithm to successfully run ROVIO 45 out of 55 times. If we compare fast ROVIO on the RPI Zero 2 W with the laptop and TX2 on Machine Hall 5, we see that fast ROVIO on the RPI Zero 2 W does not drift. This is because the RPI Zero 2 W did not detect the bad feature from Fig. 3 and 4, because of the discarded frames.

When reducing the number of features from 25 to 15 we see that more frames are processed. For the original ROVIO we see that it processes more than double the frames, however, it only slightly improves from 30 to 32 successful runs. Similarly, we also see that sparse ROVIO slightly improves from 29 to 34 successful runs. Interestingly, we see that sparse-15 ROVIO processes more frames than fast-25 ROVIO and yet it fails more often. This indicates that computational peaks, when new features are added, are important to minimize to have a stable ROVIO filter. The reason for this might be that when new features are added, the depth of the features are unknown, which makes it hard to track when too many frames are discarded. For sparse-15 ROVIO we see that it processes almost all frames and the successful runs increase from 45 to 53. The average RMSE on Machine Hall 5 for fast ROVIO is increased from 0.64 to 1.46. This is because in 1 out of 5 runs, it detected the bad feature and started to drift at the beginning of the sequence, similarly as on the laptop and TX2 (when processing all frames). It can be seen that 2 out of 5 times fast-15 ROVIO diverges on V101. We observed that in the second processed frame (the first frame that tracks detected features) the features are tracked incorrectly. This causes the attitude to be wrongly estimated. Therefore, there is an offset between the estimated gravity vector and the measured gravity vector in the accelerometers. This causes the filter to estimate a movement (while the drone is not moving yet). New features cannot be tracked due to the wrong estimation of the prediction step and already tracked features are falsely estimated to be far away. For 3 out of 5 runs, the first frame was discarded, which prevented the filter from diverging.

In Table 4 we compare fast-15 ROVIO on the RPI Zero 2 W with the original ROVIO on other computationally limited devices. The RMSE of the trajectories for the Intel NUC, UP Board and ODROID XU4 is taken from [23]. In [23] the trajectories are aligned with the ground truth using a 7DOF alignment [25]. In order to compare the results fairly, we also use a 7DOF alignment for fast-15 ROVIO on the RPI Zero 2 W in Table 4. Original ROVIO on the Intel NUC is the most accurate because it is computationally powerful enough to process ROVIO. Furthermore, we see that original ROVIO does not run with 25 features on the UP board and ODROID XU4. For the ODROID XU4 it was possible to run ROVIO with 10 features, but not on the UP Board [23]. Our modified fast ROVIO can successfully run on the EuRoC dataset 53 out of 55 times using 15 features. Even though the RPI Zero 2 W is computationally more restricted, it has a comparable or better accuracy than original ROVIO on the ODROID XU4 with 10 features, except for MH05 where fast-ROVIO diverges (1 out of 5 times) in the beginning of the sequence (see Fig. 6).

4.3. *UZH-FPV drone racing dataset*

We also run the original, sparse and fast ROVIO on the UZH-FPV drone racing dataset [22]. We are using sequences 03, 05, 06, 07, 09 and 10 of the forward-facing camera indoor, because those sequences contain the ground truth of the trajectory. We do the evaluation on the NVIDIA Jetson TX2 and Raspberry Pi Zero 2 W. On the NVIDIA Jetson TX2, we use 25 features (original setting) and for the RPI Zero 2 W, we use 15 features. We reduce the camera state buffer to 1 and run each sequence 5 times and take the average of the successful runs. All other parameters are set to the original settings (prediction noise of the velocity estimation: $4 \cdot 10^{-6}$, whereas in Section 4.1 and 4.2 we have used $4 \cdot 10^{-5}$).

In Table 5 the average RMS of the APE of the successful runs on the UZH-FPV drone racing dataset is shown with the number of successful runs out of 5 tries per sequence. Furthermore, we show the average processed frames in Table 6, again we have excluded runs where the filter has diverged. We can see that on the NVIDIA Jetson TX2 sparse ROVIO is slightly more accurate than the original and fast ROVIO. This is because sparse ROVIO processes more frames than the original ROVIO, since it is computationally more efficient. Sparse ROVIO already processes almost all frames. Therefore, the main difference between fast ROVIO and sparse ROVIO on the TX2 is that sparse ROVIO selects new features to track more carefully. Furthermore, we can see that fast ROVIO failed twice on sequence 09, whereas original and sparse ROVIO have not failed on this sequence.

However, on the RPI Zero 2 W, we see that fast ROVIO performs better than the original and sparse ROVIO in terms of successful runs. Original ROVIO has diverged 14 times out of the 30 runs, sparse ROVIO has diverged 12 times and fast ROVIO (on the RPI Zero 2 W) did not fail a single time. The reason for original ROVIO and

Table 5. Average RMS of the APE in meters / successful runs out 5 runs on the UZH-FPV drone racing dataset with a 4DOF post-process trajectory alignment.

Machine	Algorithm	03	05	06	07	09	10
NVIDIA	Original-25	0.96/**5**	0.67/**5**	0.53/**5**	1.01/**5**	**0.40/5**	**0.49/5**
Jetson	Sparse-25	**0.86/5**	**0.59/5**	**0.47/5**	**0.86/5**	0.41/**5**	0.58/**5**
TX2	Fast-25	0.89/**5**	0.60/**5**	0.60/**5**	0.94/**5**	0.67/3	0.55/**5**
RPI	Original-15	X/0	**0.53/5**	3.03/2	X/0	0.78/4	0.75/**5**
Zero	Sparse-15	X/0	0.66/**5**	2.36/2	1.23/1	0.61/**5**	**0.62/5**
2W	Fast-15	**1.39/5**	0.67/**5**	**1.00/5**	**1.04/5**	**0.54/5**	0.69/**5**

Table 6. Average processed frames of the successful runs on the UZH-FPV drone racing dataset.

Machine	Algorithm	03	05	06	07	09	10
NVIDIA	Original-25	2165	4080	1617	2864	2050	2083
Jetson	Sparse-25	2432	4157	1943	3163	2065	**2123**
TX2	Fast-25	**2510**	**4159**	**1967**	**3173**	**2062**	**2123**
RPI	Original-15	X	1796	692	X	941	871
Zero	Sparse-15	X	2950	1090	1751	1467	1447
2 W	Fast-15	**1995**	**3678**	**1490**	**2518**	**1870**	**1946**
Total frames		*2552*	*4162*	*1970*	*3177*	*2068*	*2127*

sparse ROVIO diverging that often is that many frames are discarded, which makes it very hard to track when flying at a high speed. Fast ROVIO has processed around $30-40\%$ more frames than sparse ROVIO. The difference is bigger on the UZH-FPV dataset than on EuRoC ($10-30\%$) because the drone is flying at higher velocities in the UZH-FPV dataset. Therefore, features have a higher optical flow, which makes them move faster out of the image and harder to track. Thus, new features need to be detected more often on the UZH-FPV dataset.

5. Conclusion

We have made ROVIO computationally more efficient with sparse ROVIO, mainly by exploiting the sparse matrices. The accuracy is similar to 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, we have reduced the computational peaks of ROVIO by using the FAST score instead of the more expensive Shi-Tomasi score, which we called fast ROVIO. This allowed us to run a stable ROVIO with 15 features on an RPI Zero 2 W. This comes with a cost that the selected features to track are slightly lower quality features. We saw that this is mainly an issue when the camera is not moving since the depth of the features cannot be observed. Therefore, ROVIO is not able to detect the falsely tracked feature as an outlier. For this reason, we recommend using sparse ROVIO on devices that are computationally powerful enough to process the frames at a sufficient frame rate. However, on computationally limited devices, fast ROVIO makes a big difference in terms of the stability of the filter because ROVIO often diverges when too many consecutive frames are discarded.

Appendix A Code

Two branches for sparse ROVIO and fast ROVIO are available on: `https://github.com/tudelft/rovio2`

References

[1] K. Sun, K. Mohta, B. Pfrommer, M. Watterson, S. Liu, Y. Mulgaonkar, C. Taylor and V. Kumar, Robust Stereo Visual Inertial Odometry for Fast Autonomous Flight, *IEEE Robotics and Automation Letters* **3** (April 2018) 965–972.

[2] M. Bloesch, M. Burri, S. Omari, M. Hutter and R. Siegwart, Iterated extended Kalman filter based visual-inertial odometry using direct photometric feedback, *The International Journal of Robotics Research* **36** (September 2017) 1053–1072.

[3] A. Mourikis and S. Roumeliotis, A multi-state constraint Kalman filter for vision-aided inertial naviga-

tion, *2007 IEEE International Conference on Robotics and Automation, ICRA'07*, *Proceedings - IEEE International Conference on Robotics and Automation*, (IEEE, Rome, Italy, November 2007), pp. 3565–3572.

[4] T. Qin, P. Li and S. Shen, VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator, *IEEE Transactions on Robotics* **34** (August 2018) 1004–1020.

[5] S. Leutenegger, S. Lynen, M. Bosse, R. Siegwart and P. Furgale, Keyframe-Based Visual-Inertial Odometry Using Nonlinear Optimization, *The International Journal of Robotics Research* **34** (February 2014) 314–334.

[6] G. Huang, Visual-Inertial Navigation: A Concise Review, *2019 International Conference on Robotics and Automation (ICRA)*, (IEEE, Montreal, QC, Canada, May 2019), pp. 9572–9582.

[7] M. Dharmadhikari, H. Nguyen, F. Mascarich, N. Khedekar and K. Alexis, Autonomous Cave Exploration using Aerial Robots, *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, Athens, Greece (July 2021), pp. 942–949.

[8] P. Foehn, D. Brescianini, E. Kaufmann, T. Cieslewski, M. Gehrig, M. Muglikar and D. Scaramuzza, AlphaPilot: autonomous drone racing, *Autonomous Robots* **46** (January 2022) 307–320.

[9] G. Brunner, B. Szebedy, S. Tanner and R. Wattenhofer, The Urban Last Mile Problem: Autonomous Drone Delivery to Your Balcony, *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, Atlanta, GA, USA (August 2019), pp. 1005–1012.

[10] C. Forster, Z. Zhang, M. Gassner, M. Werlberger and D. Scaramuzza, SVO: Semidirect Visual Odometry for Monocular and Multicamera Systems, *IEEE Transactions on Robotics* **33** (April 2017) 249–265.

[11] J. Shi and Tomasi, Good features to track, *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, (IEEE, Seattle, WA, USA, June 1994), pp. 593–600.

[12] Q. Lu, J. Xu, L. Hu and M. Shi, Parallel VINS-Mono algorithm based on GPUs in embedded devices, *International Journal of Advanced Robotic Systems* **19**(1) (2022) p. 17298814221074534.

[13] B. Nagy, P. Foehn and D. Scaramuzza, Faster than FAST: GPU-accelerated frontend for high-speed VIO, *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE (2020), pp. 4361–4368.

[14] H. Liu, M. Chen, G. Zhang, H. Bao and Y. Bao, ICE-BA: Incremental, Consistent and Efficient Bundle Adjustment for Visual-Inertial SLAM, *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, (2018), pp. 1974–1982.

[15] E. J. Shamwell, K. Lindgren, S. Leung and W. D. Nothwang, Unsupervised Deep Visual-Inertial Odometry with Online Error Correction for RGB-D Imagery, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **42**(10) (2020) 2478–2493.

[16] G. Guennebaud, B. Jacob and others, Eigen v3 http://eigen.tuxfamily.org, (2010).

[17] J. Solà, J. Deray and D. Atchuthan, A micro Lie theory for state estimation in robotics (2018).

[18] D. G. Lowe, Distinctive Image Features from Scale-Invariant Keypoints, *International Journal of Computer Vision* **60** (2004) 91–110.

[19] H. Bay, T. Tuytelaars and L. Van Gool, SURF: Speeded Up Robust Features, *Computer Vision – ECCV 2006*, eds. A. Leonardis, H. Bischof and A. Pinz (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006), pp. 404–417.

[20] E. Rublee, V. Rabaud, K. Konolige and G. Bradski, ORB: An efficient alternative to SIFT or SURF, *2011 International Conference on Computer Vision*, (2011), pp. 2564–2571.

[21] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. Achtelik and R. Siegwart, The EuRoC micro aerial vehicle datasets, *The International Journal of Robotics Research* **35** (September 2016) 1157–1163.

[22] J. Delmerico, T. Cieslewski, H. Rebecq, M. Faessler and D. Scaramuzza, Are We Ready for Autonomous Drone Racing? The UZH-FPV Drone Racing Dataset, *2019 International Conference on Robotics and Automation (ICRA)*, (IEEE, Montreal, QC, Canada, August 2019), pp. 6713–6719.

[23] J. Delmerico and D. Scaramuzza, A Benchmark Comparison of Monocular Visual-Inertial Odometry Algorithms for Flying Robots, *2018 IEEE International Conference on Robotics and Automation (ICRA)*, (IEEE, Brisbane, QLD, Australia, May 2018), pp. 2502–2509.

[24] Z. Zhang and D. Scaramuzza, A Tutorial on Quantitative Trajectory Evaluation for Visual(-Inertial) Odometry, *IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IROS)*, (IEEE, Madrid, Spain, October 2018), pp. 7244–7251.

[25] S. Umeyama, Least-squares estimation of transformation parameters between two point patterns, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **13**(4) (1991) 376–380.

[26] M. Grupp, evo: Python package for the evaluation of odometry and SLAM `https://github.com/MichaelGrupp/evo`, (2017).