F3

Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

Visual Localization with HoloLens

Pavel Lučivňák

Supervisor: doc. Ing. Tomáš Pajdla Ph.D.
Field of study: Artificial Intelligence
Subfield: Open Informatics
August 2020

Acknowledgments

TODO: Děkuji ČVUT, že mi je tak dobrou *alma mater*.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 14. August 2020

Abstract

Visual localization is a common computer vision poroblem of estimation of the camera pose that took a particular RGB image. The pose is estimated relative to a certain coordinate system. One particular instance of this problem occurs in HoloLens mixed reality. In a mixed reality settings, we are projecting virtual objects into the real world environment. In order to maintain the objects as the user navigates around a room, we need to keep track of the device pose. HoloLens already does this, however there is a room for improvement. A new indoor visualization datasets, consisting of 2 rooms and 3 query sets, has been created. Two of these query sets are sequential images (from HoloLens). Reference poses are also provided (although not for all queries). We have designed new methods that aim to merge InLoc [1] approach to indoor visual localization with the data from HoloLens. My implementation outperformed the original InLoc paper on the task of sequential localization from RGB images. However, our approach turned out to perform significantly worse than the pose estimation from HoloLens itself. I provide an overview of sources of errors in the new and InLoc methods for potential future improvement.

Keywords: HoloLens, localization, Matterport, Vicon

Supervisor: doc. Ing. Tomáš Pajdla Ph.D.
CIIRC ČVUT,
Jugoslávských partyzánů 1580/3,
Praha 6 - Dejvice,
160 00

Abstrakt

Vizuální lokalizace je často řešená problematika v počítačovém vidění. Typicky chceme určit pózu (polohu a orientaci) fotoaparátu, který pořídil daný RGB snímek. Odhadnutá póza se vztahuje k nějakému námi definovanému souřadnicovému systému. Konkrétně se tento problém řeší ve smíšené realitě v HoloLens. Promítáme zde virtuální objekty to reálného prostoru. Abychom mohli udržet tyto objekty na správném místě, zatímco se uživatel brýlý pohybuje, je potřeba vědět, kde se HoloLens nachází. HoloLens jako takové umí sledovat svou vlastní pózu, ale výsledek není perfektní. Vytvořil jsem novou sadu dat, která obsahuje skeny dvou místností a tři množiny query obrázků. Dvě z nich pochází právě z HoloLens. Obsahem datové sady jsou i referenční pózy fotoaparátu (u některých zatím chybí, ale dají se v případě potřeby vygenerovat). Navrhl jsem nové algoritmy, které kombinují metodu InLoc [1] s daty, co nám dává HoloLens. Má implementace je na sekvenčních obrázcích přesnější, než původní InLoc. Mé metody jsou ale výrazně méně přesné, než lokalizace ze samotných HoloLens. V práci shrnuji mé poznatky, proč vnikají určité chyby související s novými metodami nebo s InLocem. V budoucnu je možné na práci navázat a chyby zredukovat.

Klíčová slova: HoloLens, lokalizace, Matterport, Vicon

Překlad názvu: Vizuální lokalizace pro HoloLens

Contents

Figures

Tables

Chapter 1

TODO

- Check the assignment whether it corresponds to the plan below.
- Make an outline.
- Suggest a method for localization from a image sequences.
- Evaluate and demonstrate it.
- Get queries for B-670, inspect the data, localize, evaluate.
- Localization of sequences will be based on predicting the next view from the pose obtained by localizing an initial segment of the sequence and attaching the next view(s) using the relative pose between the views provided by HoloLens pose tracking.
- Level-1: localize initial segment of length 1, evaluate, wait or this to work, ...
- Evaluate w.r.t. to the Level-0 (baseline) obtained by localizing just one image without any verification by predicting the next views. Introduce another label = not-localized.
- Level-2: localize initial segments of length > 1. How t do it? Use the maximal 1st/2nd NN ratio to select the best image in the indexing phase. Next use the sequence as a generalized camera and replace p3p with GP6P.
- Level-3: Combine images before image indexing. How to do it? We don't know as of now.
- Zadani ma byt umistene jinde, viz email z 14.7.2020. Fyzicka verze ma obsahovat take podpisy.

- The ctuthesis style is kind of ugly. Consider removal of blank page after chapter (is it on purpose?). Remove "F3" on the top page.
- Mention what I have tried but haven't finished - Habitat (for synthetic datasets), HoloLens sequence #2 evaluation).
- Time and space complexity. Consider asymptotic complexity too.
- Use more of "Our", "We" than "My" and "I"

Chapter 2

Introduction

Visual localization is a common computer vision problem where, given an RGB image, we want to estimate the camera pose. Such a camera pose can be specified by 6 parameters - 3 of which describe its position in space and the other 3 represent its orientation in the space. In case of outdoor visual localization, the problem can be simplified by making use of GPS for approximate position localization. Visual localization is a problem that also needs to be addressed indoors, however. It has use cases in e.g. Augmented and Mixel Reality applications. Of course, the GPS signal is unusable in a building. In this thesis, I am going to focus on indoor visual localization with HoloLens. HoloLens is a mixed reality device. HoloLens provides a powerful tracking of the camera as user navigates around a room. An idea of this thesis is to improve it further. Imagine a use case where we place virtual objects into the mixed reality. As user navigates around the room, we need to track the pose of HoloLens in order to maintain the object placements. The indoor environment is problematic for several reasons. One of them being that there are a lot of similar areas - the same type of windows, doors, textureless walls. Furthermore, the environment can change easily as people interact with it.

The objectives of this work are as follows. State of the art in indoor localization must be reviewed, in particular the NetVLAD [3] and InLoc [1] papers. Create a new indoor dataset based on the InLoc dataset [1]. The WUSTL dataset was used in InLoc. The new dataset must also contain query images that were taken in a sequence (as user with HoloLens walks in the room). Make InLoc run on the newly created dataset, by prossing the query images in non-sequential fashion. Implement an improvement that takes the HoloLens data into account. One of the improvements should include taking multiple historical camera data into account (InLoc currently only uses a single camera pose, because it does not deal with sequential data). At last, the performance of the newly implemented algorithms shall be evaluated.

This work is organized the following way. Chapter 3 contains related work on the topic of indoor visual localization. Background that my software and algortihms rely on is also described. The newly acquired dataset is described in Chapter 4. It contains statistics of the dataset, its structure and how it was created. An implementation of the techniques described here area covered in Chapter 5. Note that it is only a proof-of-concept implementation, unsuitable for real-world use out of the box (just as the InLoc implementation). Chapter 6 evaluates the newly developed methods and compares them with some baseline methods. Sources of errors are also noted. Finally, the Chapter 7 is a summary of this work, whether it fullfilled the assignment and possible future work.

TODO:

- repeat some info about InLoc,
- state of the art HoloLens v1 accuracy so far?

Chapter 3

Literature review

TODO:

- Relevant work with regards to HoloLens or indoor localization,
- HoloLens,
- Matterport,
- Vicon,
- procrustes? I think I just need one or two sentences,
- Single view depth estimation,
- Deep depth completion,
- more?

3.1 InLoc

InLoc [1] a powerful method (state-of-the-art in 2018) for indoor visual localization. All the newly developed algorithms in this work use InLoc or its modification at its core. Not necessarily because there is no better method out there, but because it is the topic of this thesis (and the thesis supervisor is a co-author of the InLoc paper,

which is beneficial for getting familiar with the method quickly). To understand the methods developed in this work better, it is useful to learn about InLoc first.

InLoc operates on top of an InLoc dataset. The dataset contains:

- 256 query RGB photos,
- 10 000 cutout RGBD images,
- 277 reference panorama poses (determined using [4]),
- reference query poses,
- query-cutout similarity matrix (known as score).

The dataset has been acquired at the Washington University in St. Louis. Five floors (at two buildings) were used to build the dataset. The InLoc method assumes existence of a 3D map - which is represented by the RGBD cutouts. The cutout RGBD images are a result of perpesctive view extraction from RGBD panorama images. The panorama images were captured accross all five floors using a high-end Faro 3D scanner. The query RGB images represent the images for which we aim to determine camera pose. These were taken using a smartphone camera with no depth information. Queries were taken only at two floors; the other floors serve as a confusion for InLoc.

The query photos were taken at a different time of the day, to take illimunation and interior changes into account. The query reference poses are needed, to attest how well InLoc performs on the pose estimation task. These reference poses were determined using (taken from [1]):

1. Selection of the visually most similar database images.
2. Automatic matching of query images to selected database images.
3. Computing the query camera pose and visually veri- fying the reprojection.
4. Manual matching of difficult queries to selected database images.
5. Quantitative and visual inspection.

Understanding the details of reference pose generation is not necessary for understanding this work. This is because, as we will see in Chapter 4, our **TODO: or my?** new dataset:

- Already contains reference panorama poses from Matterport.
- Uses a pose-tracking system Vicon to estimate the reference query poses.

Let's take a look at a high-level overview of how the InLoc method operates. The following is a simplified and paraphrased (from the InLoc paper [1]) description:

1. For every query image: find top N similar cutout images using the score similarity matrix.
2. For every query-cutout pair: find tentative pixel-to-pixel correspondences using matching of NetVLAD 3.2 features.
3. Re-rank the top N cutout lists according to the highest number of tentatives found (if there is a draw, the original query-cutout score decides the order).
4. Choose top $M \leq N$ cutouts in the lists.
5. For all query-cutout pairs, construct a pose estimate using P3P-RANSAC [1]. This is possible since the pixel-to-pixel correspondences can be converted into 2D-3D correspondences (cutouts are RGBD).
6. Project the estimated poses and evaluate their similarities to the query images.
7. For every query: choose the cutout for which we have a synthesized query image with highest similarity to the query image (using DenseRootSIFT [6] [7]).

The computational requirements are missing from the InLoc paper. However, the authors mention the need for about 14 GB RAM in their experiment, to hold the image descriptors in memory.

3.2 NetVLAD

NetVLAD [3] is a convolutional netural network [2] (CNN) architecture for visual place recognition. The input to this network is an RGB image and the output is a feature representation of that image. Given two images and their corresponding features, we can compute to what extent they represent the same place. In this work, as well as in InLoc [1], we use a VGG-16 [10] + NetVLAD model that is pre-trained on Pitts30k [3] dataset.

[1] P3P is covered in section 3.3; refer to [5] for RANSAC description.
[2] See the original paper [8] or a Deep learning survey [9].

3.3 P3P

The P3P problem [5] is a problem of estimating a calibrated camera pose using at least three 2D-3D correspondences. A calibrated camera is a camera for which we know its calibration matrix (see section 3.4). RANSAC [5] can be used to further improve estimation accuracy, if some of the correspondences are imprecise or incorrect.

3.4 Camera coordinate system

Figure 3.1 shows a camera coordinate system γ, that defines the camera pose.

Calibration matrix. A camera calibration matrix K is a linear transformation that converts points in camera coordinate system γ into points in image coordinate system β. It is defined by five parameters [2]:

$$\mathrm{K} = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ 0 & k_{22} & k_{23} \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.1}$$

They represent focal length, sensor dimensions, origin of the image coordinate system and more.

Our implementation, however, only requires a subset of those parameters. Thus the calibration matrix K can be construced as:

$$\mathrm{K} = \begin{bmatrix} f & 0 & w/2 \\ 0 & f & h/2 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.2}$$

where w and h are width and height of the camera sensor in pixels. Focal length is also in pixel units.

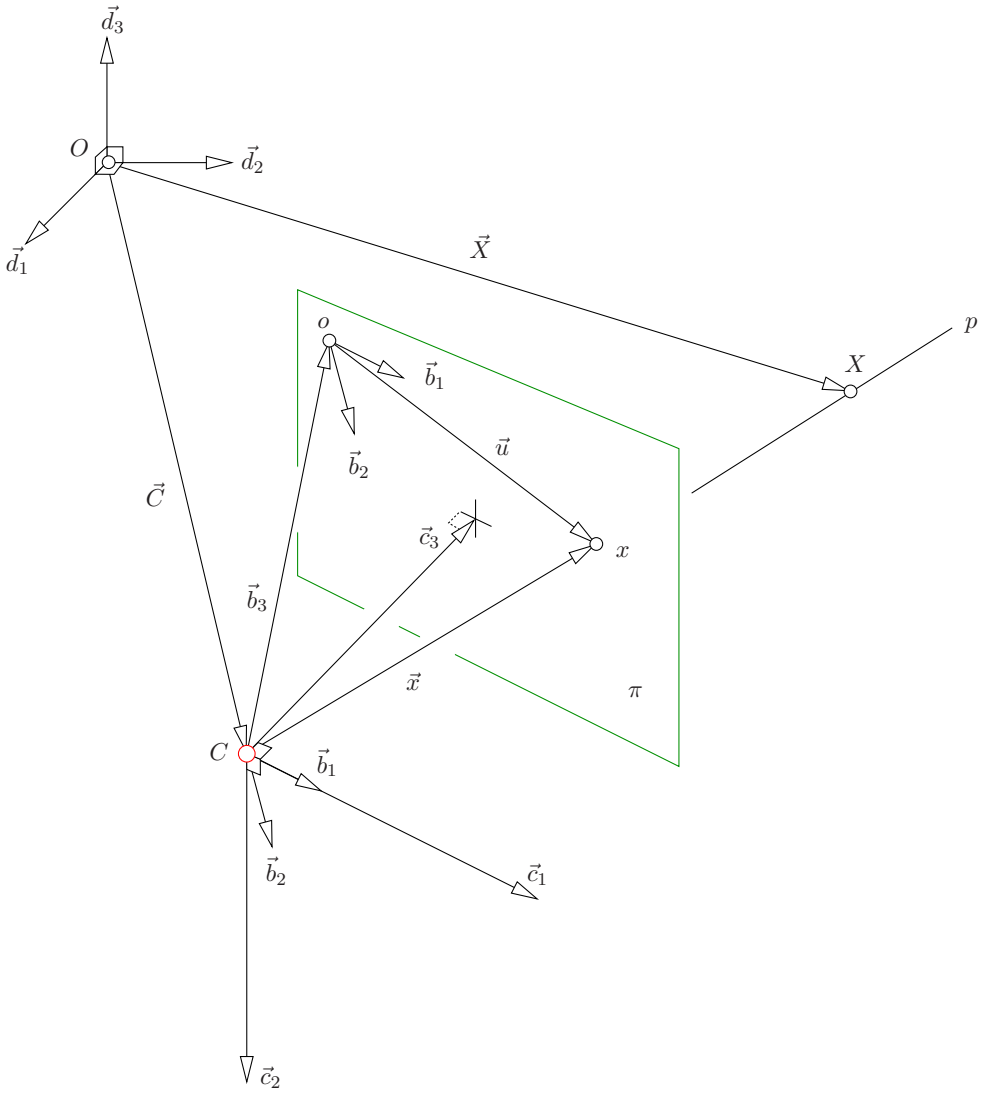


Figure 3.1: Camera coordinate system γ with origin at $\vec{C}$ with respect to some World coordinate system γ; and bases $\vec{c}_1, \vec{c}_2, \vec{c}_3$. Camera points the $\vec{c}_3$ direction, having $\vec{c}_1$ on its right. $\vec{b}_3$ defines the start of image coordinate system (pixel at 0,0). Vectors $\vec{b}_1, \vec{b}_2$ are considered to be orthogonal, as we are dealing with a rectangular sensor in this thesis. Point X with 3D coordinates $\vec{X}$ projects onto the image plane to a point x with image coordinates $\vec{u}$. The two points form a 2D-3D correspondence. Figure is from [2].

3.5 Multi-camera pose estimation

This work often operates on query images that were taken in a sequence: imagine a person walking around a room and taking a picture every once in a while. Considering the sequential nature of the queries can help us improve localization performance. To achieve this, we need to be able to estimate poses of multiple cameras in the sequence.

In general, these cameras are referred to as a rig of cameras. The generalized pose-and-scale problem (GP4Ps/gsP4P), defined in [11] describes such a problem in general and provides an efficient solution. A particular implementation of the solver, called MultiCameraPose, is provided as an open source software. The project is a result of a recent work [12]; for my use-case, I had to modify the implementation slightly and it is available at [13]. The implementation is very fast, computing the pose estimates of a rig with 5 cameras in about 60 milliseconds. A concrete usage of the MultiCameraPose program is described in the Implementation chapter 5.

Chapter 4

Dataset

The original InLoc implementation is using the InLoc dataset [1], which is based on data taken at the Washington University in St. Louis (WUSTL dataset). The InLocCIIRC dataset aims to keep the same structure as the InLoc dataset. The new dataset was created at the Czech Institute of Informatics, Robotics and Cybernetics (CIIRC).

The dataset is a result of scanning two rooms at CIIRC: the B-670 lecture hall and a room B-315. For scanning the environments, a Matterport 3D scanner is used. Let's call the environments *spaces*. This scanner is much faster to operate and cheaper than the Faro 3D scanner used in InLoc (WUSTL dataset). The disadvantage is that the resulting point cloud model tends to be of lower quality. Matterport creates a point cloud and a mesh model of each space. This is made possible by scanning the area at various locations. Let's call each such scan a *sweep*, to match the Matterport API terminology. To construct the models, RGBD panoramas are taken around the rooms. In B-670, I have taken 31 such panoramas. In B-315, I have taken 27 panoramas. Overall, there are 58 RGBD panoramas taken by Matterport 3D scanner. The scanner was mounted on a tripod at height of approximately 1.52cm and I tried to avoid walls and objects in 60cm radius.

When creating an RGBD panorama, the Matterport scanner has to revolve around yaw axis in order to capture the scene in 360°. For each RGBD panorama, we are given the pose of the Matterport scanner at the moment right before the rotation started. These poses are provided by Matterport, so we don't have to bother to estimate them ourselves as in [4].

Another outcome of the sweeps are RGB panoramas. Matterport does not support

automatic gathering of these panoramas, so they have to be downloaded manually for every sweep. Another problem is that these downloaded RGB panoramas are not pointing the same direction as is the initial orientation of the Matterport camera. Therefore, I have created a tool to semi-automatically find the proper orientations. This is done by

1. projecting the point cloud model so that the camera's pose matches the sweep's position and orientation,
2. sampling the RGB panoramas around the yaw axis and picking such a sample that best matches the projection. The matching is done by picking such a sample for which the amount of edges in a difference edge image is minimal.

This approach works well, however it may still fail in an exceptional case. Then, a user is encouraged to try 2nd lowest amount of edges, 3rd least amount and so on. Alternatively, one may try to increase the point size of projected the model. As a last resort, one can manually find the RGB panorama sample by manually rotating it via a provided script.

Once we have the RGB panoramas which are pointing the same direction as the RGBD panoramas, we can move into the next stage. Here we construct cutouts, which are projections of the RGB panoramas at a specific orientation. As in InLoc, I am sampling around the yaw axis per 30° under the pitch direction of $\{-30, 0, 30\}$ degrees. The cutouts also contain information about the depth (not provided by Matterport).

The dataset contains sets of query images (queries). The first set, called s10e, was taken by a smartphone camera — via Samsung Galaxy S10e's wide angle rear facing lens. I have taken 40 query images in a restricted area of room B-315. This room was chosen to be in the dataset, because it contains a pose estimation system called Vicon. The other two sets of queries were obtained using 1st generation HoloLens. The sets are named HoloLens1 and HoloLens2 — the suffix number indicates the sequence number. The major difference between s10e and HoloLens query datasets is that the queries from HoloLens form a sequence of images, as the user walked around the room. The sequential nature of those query datasets shall be leveraged, and data from multiple cameras may be used for a higher precision pose estimation of a current frame.

All of the query images were taken in this area, so that their reference pose is known. No queries were taken in room B-670, as it would be time consuming to estimate the reference poses manually (or creating a program that does this). Hence, its only purpose is to serve as a confuser.

The queries in the s10e set have a pixel resolution 4032×3024. InLoc implementation requires the knowledge of focal length of the camera that was used when taking the query images. I found conflicting information about the S10e's field of view (FoV) online, and the focal length didn't add up. **TODO: talk about a similar problem with HL.** I ended up computing the focal length manually with the help of a tripod and a ruler. The focal length turned out to be 3172 pixels. The IDs of query images are sorted in a non-decreasing difficulty, e.g. queries with IDs 1 to 10 were taken such that the camera's direction vector is roughly parallel with the floor. Queries with higher IDs have the camera rotated on a tripod under any direction.

The HoloLens queries have a pixel resolution of 1344×756 pixels and according to the official documentation, the horizontal FoV is 67°. However, looking at the data generated while capturing the sequences, HoloLens provides a cameraProjectionTransform matrix. According to an article, the effective hFoV can be computed as

$$\text{hFoV} = 2 * \arctan\left(\frac{1}{\text{cameraProjectionTransform.m11}}\right), \tag{4.1}$$

which gives the value of 65.83 degrees.

The sweeps, used to construct the point cloud model, were taken on Thursday/Friday midnight. The s10e query images were taken on a Monday morning 3 days later. Note that there was a weekend within these days, meaning the scene didn't change a lot during that time. The reason the query images were taken later was to test what happens when items such as chair, lighting and people move around or change.

The two HoloLens sequences were captured about three weeks later. This means the environment was more challenging to worth it, because it has changed from the state in which it was scanned by Matterport.

Alignments define the pose of individual sweeps within the space they are in. Because the poses are given to us from Matterport, we do not need to perform the generalized iterative closest point (GICP) step, as in InLoc. Because Matterport gives us an entire model (point cloud and mesh) of each scanned space, we do need to consider alignments at all. They were useful in InLoc, where there were individual point clouds for sweeps and thus the 3D coordinates of the points projecting onto cutouts were wrt the sweep coordinate system.

In InLoc, there are point cloud models for every sweep. On the contrary, in InLocCIIRC we have a model for each space.

The InLoc implementation requires the knowledge of scores between every pair of a query image and a cutout image. An individual score describes similarity between the two images. When the software is run, InLoc chooses, for each query, top N cutouts with the highest scores. The other cutouts will not be considered. It is thus quite important that these scores are relevant. NetVLAD [3] descriptors are computed for both cutouts and query images. The features are the output of the L2 normalization layer. A score between a query image and a cutout is computed using a dot product between the two feature vectors. Note that the similarity scores of cutouts for a query do not represent a probability distribution, and thus don't need to sum up to one. The code for doing so was not provided in InLoc, so I came up with an implementation that reuses existing InLoc MATLAB components. The resulting scores seem to be meaningful, but a reference implementation would have been better.

4.1 Reference poses

For every query we need to know its reference pose, in order to evaluate how accurate the pose estimation algorithms are. The pose of the cameras used to take the query pictures in query sets was also being tracked by a pose estimation system – Vicon. Figure 4.2a shows the s10e camera (thus also its coordinate system) and a coordinate system that is being tracked by Vicon. Let the latter coordinate system be called Marker.

Let's now focus on a more difficult scenario, which is the reference pose determination of the HoloLens queries. There are three reasons why the reference poses cannot be simply taken from the Vicon tracking:

1. camera pose and Marker are widely different,
2. the Vicon coordinate system differs from the World coordinate system,
3. Vicon started tracking before HoloLens was run, as visualized in figure 4.3.

Luckily, the second issue turned out to be easily mitigated. I have been told where the origin of the Vicon coordinate system is. And by experimentation, the rotation matrix that converts Vicon bases to World bases was found. Because the Vicon bases and World bases are aligned to the room (i.e. a basic vector is parallel with the floor or the walls), the rotation matrix can be represented by a simple rotation.

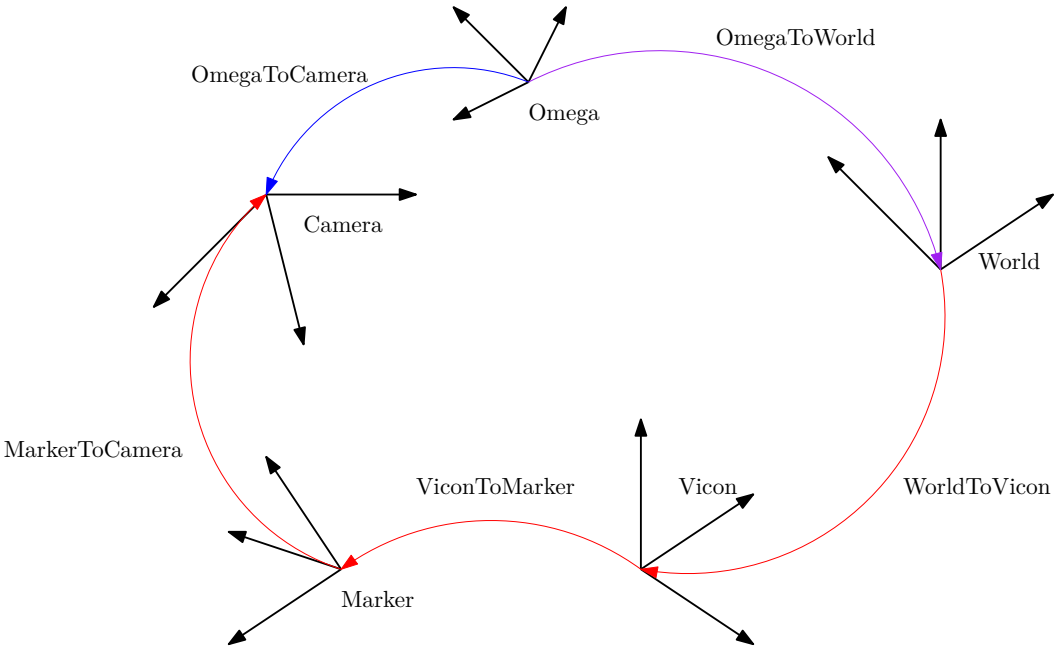
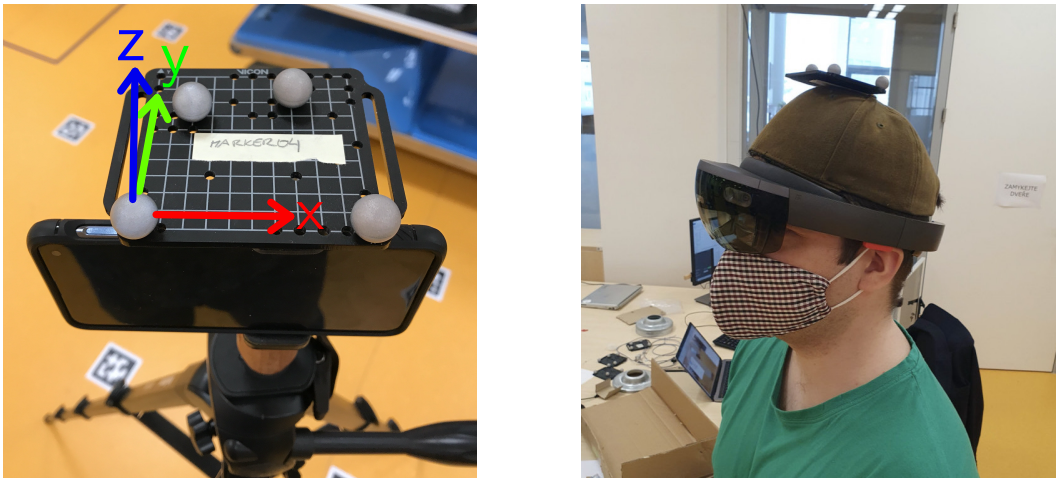


Figure 4.1: Visualization of the coordinate systems we are dealing with. Omega is the initial unknown HoloLens CS **TODO: what is CS?**. Notice that Omega has a scaling independent of the World CS scaling. Linear transformations are shown by the arrows. There are in fact two slightly different Camera coordinate systems – one that is estimated from HoloLens and another one (reference pose) that is estimated using Vicon. OmegaToCamera is known for most of the HoloLens queries [1], because the data comes from HoloLens. ViconToMarker is provided from Vicon tracking. WorldToVicon has been manually determined. MarkerToCamera has been approximated by an algorithm described in the Reference poses section 4.1.

The transformation from Marker to camera is considered to be a constant (for all queries in a query set), because the tracking device is securely attached to the camera. One could manually estimate that transformation and visually evaluate how close the model projection is to the original query image. However, this approach is prone to errors. Instead, a quantitative approach was employed, which I describe next.

For a particular query set, we need to manually set up the reference pose for a small number of queries. I used 6 of them in HoloLens1. Let these queries be called *interesting* queries. For such a query, we manually find 9 2D-3D correspondences. The 2D correspondences are carefully chosen, such that they actually represent the same 3D point – because the 3D points were captured up to three weeks earlier than the query images and the environment has changed. For each query with the correspondences, we compute its initial reference pose using P3P. The pose returned by P3P may not be completely accurate, however.

Given reference poses for 6 queries and corresponding poses from Vicon, we can



(a) : s10e sequence. **(b) :** HoloLens2 sequence.

Figure 4.2: The camera and marker (the object tracked by Vicon). Marker coordinate system is visualized in subfigure 4.2a by the xyz arrows.

almost compute individual Marker to camera transformations. The last piece missing is a synchronization constant, to match the correct Vicon pose taken at Vicon tie with a particular query taken at HoloLens time. I created a script, findOptimalParams-ForInterestingQueries.m, which computes the Marker to camera transformations and evaluates the reference poses quality both quantitatively (reprojection error) and visually (manually investigated by the user). Currently, user must guess a synchronization constant. Finding a reasonable synchronization constant does not take long. Alternatively, one could implement a brute-force search, where various synchronization constants are guessed and the one with lowest quantitative error is chosen. In my case this was not necessary. At the end of the script, a generic transformation is suggested, which is an average of the individual transformations. The quality of the generic transformation is again evaluated on all the 6 queries. This generic transformation and the synchronization constant are used as a baseline and are further optimized, described next.

A brute-force search is employed to find an improved version of the baseline transformation and synchronization constant in nearby space. First, an improved synchronization constant is estimated, by simply evaluating the interesting queries on the same transformation but for different synchronization constants, that are close to the baseline constant. Then, different transformations are being tried. An Marker to camera transformation is described by a 3D translation vector and a 3x3 rotation matrix. Note that this rotation matrix can be represented by three parameters (yaw, roll and pitch). Thus, the code iterates over predefined values of the 6 parameters, such that every combination is tried. For each combination, the reprojection error is computed and stored for later. The parameters are continuous, but I try a sequence of values nearby the baseline value, where the offset is a constant. When it comes to the translation parameters, I have had good experience with trying 17 values, where

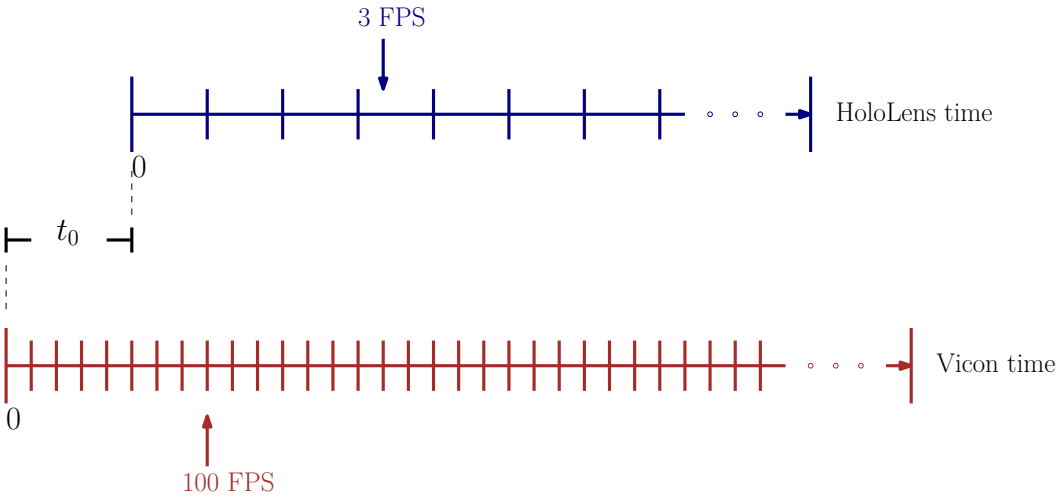


Figure 4.3: Visualization of the HoloLens and Vicon timelines. The synchronization constant must be found. Note that the sampling frequencies are vastly different. However, given a query image from HoloLens taken at some point in time (HoloLens sampling frequency), we find the corresponding reference pose that has the nearest timestamp (after taking the synchronization constant into account; Vicon sampling frequency).

the middle value is the baseline. The offset was 0.023 Matterport meters. Each orientation parameter was evaluated on 11 values with even offsets, where the middle value was the baseline. The offset was 0.5°. Additionally, the brute-force search is very time consuming, taking about 20 hours on a machine capable of processing 45 threads at once. Optionally, one can iterate over 5 synchronization constant values, for even more optimal parameters to be found. Of course, by doing that, the search will take asymptotically 5 times as much time and memory resources.

Table 4.1 shows quantitative evaluation of the quality of reference poses, after the brute-force optimization. Table 4.2 shows the same statistics for parameters prior to the optimization (baseline transformation). The improvement is not significant: 1 cm lower translation error and 0.14° lower orientation error. Figure 4.4 shows an example of the 9 manually defined correspondences and their reprojection errors.

The resulting reference poses are not perfectly matching ground truth poses, which can be seen when projecting the reference poses and comparing the results with the query images. I have created the following procedure in order to estimate the mean translation and orientation error (reference vs ground truth poses). Although we do not know the true ground truth poses, one can use the poses from HoloLens. According to [14], the poses estimated by HoloLens have the following mean accuracy with respect to the ground truth poses:

- 1.6 ± 0.2 cm translation error,

Query ID	Average projection error [px]	Sum of projection errors [px]
1	3.47	31.24
94	9.80	88.19
237	10.06	90.52
281	3.83	34.48
155	5.07	45.63
198	3.23	29.10
Sum	N/A	319.16

(a) : Reprojection error.

	Mean errors	Standard deviation of errors
Translation [m]	0.15	0.08
Orientation [m]	2.09	1.69

(b) : Estimate of reference vs ground truth poses errors. All the queries in the sequence were considered, with two kinds of exceptions. Queries, for which we do not have a reference pose (Vicon got lost) are not considered in the statistics. Queries for which we do not have a corresponding pose from HoloLens (due to the delay) are also not included in the statistics. Ground truth poses are estimated from the poses provided from HoloLens, after conversion to World coordinate system.

Table 4.1: Quantitave evaluation of reference poses quality. HoloLens1 sequence shown. Parameters describing the Marker to camera transformation were **optimized** using brute-force search.

- $2.2 \pm 0.3°$ orientation error.

Notice that namely the the translation error is very low. To estimate the quality of my reference poses wrt ground truth poses, I consider the HoloLens poses as the ground truth poses. However, because the poses from HoloLens are wrt some unknown HoloLens coordinate system, I first need to convert those poses to be wrt World. To achieve this, I use procrustes **TODO: citation**, which finds a linear transformation from one coordinate system to another (translation, rotation, scale), given corresponding 3D points. In my case, the 3D points are simply the camera centers. Procrustes minimizes the sum of squared errors of points in the same coordinate system. After the conversion, we would have ground truth estimates.

Unfortunately, there was another hidden problem that had to be dealt with, prior the reference vs ground truth pose errors could be computed. The problem is that the poses provided from HoloLens do not correspond to the query they are associated with in the data. It turns out the poses are delayed. To make matters worse, both the translation and orientation that are used to construct the camera pose are delayed by a different amount! To resolve this issue, the pose from HoloLens associated to a query is computed to be based on translation and orientation data, that comes from the future queries. I found that the best results were achieved with the following delays:

Figure 4.4: Query 94 of holoLens1 and its reprojections errors. The optimized transformation params were used. The same image on non-optimized parameters is not shown, because the average improvement of reprojection error of a the correspondences is about 2 pixels. Therefore a naked eye can barely tell which image has lower reprojection error. Green points: optimal location of 2D correspondences. Red dots: location of the 3D correspondences (projected onto 2D image plane), under the generic parameters (that aim to work across all queries in the sequence).

A consequence of the data being delayed is that, for some of the queries at end of the sequence, we do not have the poses from HoloLens available. Recall also that some reference poses are blacklisted, because Vicon got lost.

Using these delays and the procrustes method, we can compute the mean reference vs ground truth pose errors, which is:

- 15 cm translation error,
- 2.09° orientation error.

These errors may be either an upper bound (the data being delayed may still cause trouble) on the real mean errors, but they can also be approximately the true mean errors. As you can see, the translation error is significant. This is concerning, because it is not clear whether my method is better or worse than the poses provided by HoloLens themselves. Note that in case of s10e queries, the reference poses seem to have a lower error wrt ground truth. However, because we do not know the ground truth and no HoloLens poses are available here, I cannot quantitatively evaluate it (but I can compute a reprojection error on the queries that were manually assigned 2D-3D correspondences).

Query ID	Average projection error [px]	Sum of projection errors [px]
1	3.38	30.42
94	11.96	107.66
237	9.22	82.98
281	3.62	32.57
155	5.99	53.91
198	3.08	27.68
Sum	N/A	335.22

(a) : Reprojection error.

	Mean errors	Standard deviation of errors
Translation [m]	0.16	0.08
Orientation [m]	2.23	1.62

(b) : Estimate of reference vs ground truth poses errors. All the queries in the sequence were considered, with two kinds of exceptions. Queries, for which we do not have a reference pose (Vicon got lost) are not considered in the statistics. Queries for which we do not have a corresponding pose from HoloLens (due to the delay) are also not included in the statistics. Ground truth poses are estimated from the poses provided from HoloLens, after conversion to World coordinate system.

Table 4.2: Quantitative evaluation of reference poses quality. HoloLens1 sequence shown. Tables show performance on the parameters, describing the Marker to camera transformation, **prior** using brute-force search optimization.

Type	Number of frames
Translation delay	6
Orientation delay	4

Table 4.3: HoloLens provides (**TODO: actually its the Anna's software that does this**) CSV file containing information on the query images it took, when they were taken (timestamp), estimated poses and more. The camera pose estimates are represented by translation and orientation parameters, which are in Omega coordinate system. However, these parameters are wrongly assigned, as they are in fact delayed by a number of frames. The time difference between two consecutive frames is about 333 milliseconds. Optimal delays for HoloLens1 sequence are shown.

The query images can be split into two categories — InMap and OffMap. An InMap query is such a query, for which we have a cutout that has a similar pose. I have defined the pose similarity as:

- the translation difference is less than 1.3 meters,
- the angular difference between reference and retrieved rotation matrices is at most 10 degrees. **TODO: elaborate**.

The set of s10e queries consists of 5 InMap queries and 35 OffMap queries. The set of HoloLens1 queries consists of 111 InMap queries and 239 OffMap queries.

Type	Amount	Without ref. pose	Image size [px]	HFoV [°]
Query - s10e	40	0	4,032×3,024	64.86
Query - HoloLens1	350	24	1344×756	65.83
Query - HoloLens2	618	299	1344×756	65.83
Cutout	2,088	0	1,600×1,200	106.26

Table 4.4: Statistics of the **InLocCIIRC dataset**. Note that some queries are without a reference pose assigned to them. This occurs when Vicon gets lost (returns a non-sense pose for a certain period). Such queries are ignored in performance evaluation. Note that the HoloLens2 sequence contains a lot of queries, for which Vicon failed. This may be related to the fact that I moved slighly faster around the room in that sequence, making it harder for Vicon to keep track of the marker. Cutout poses are provided from Matterport and because of their quantity, not all of them were manually verified. For I never discovered a problem with the cutout poses, I consider their poses to be flawless. HFoV stands for the horizontal field of view.

The HoloLens2 does not have up to date reference poses. According to an outdated result, it contains 48 InMap and 570 OffMap queries.

The entire dataset, including the output of the InLocCIIRC demo, takes up to **TODO** GB of disk space.

The dataset statistics are depicted in table 4.4. Notice that the horizontal field of view of database cutout images is widely different from the query FoVs. When I tried to generate the dataset, such that the cutouts have horizontal FoV of 60 degrees, the resulting pose estimation accuracy became 0%. **TODO: that might been caused by that densePE bug, re-run it.** I have spent a significant time investigating why this is happening, and came to the conclusion that the problem is in the data. When one creates a cutout of a lower FoV, smaller portion of the 360° panorama gets rendered. This also means that the visual quality of the image decreases. I believe that the quality of such cutouts is not good enough for the convolutional neural network to generate reasonable feature descriptors. Figure 4.5 illustrates this problem. It seems that there is nothing we can do about it, since the pixel density of each 360° panorama is determined by Matterport. It is, however, true that one could experiment with other FoV values. Such experiments were not conducted here, as regenerating the dataset and then uploading it to an evaluation server takes a lot of time (one day is not an exception).

TODO: how are reflective surfaces handled? TODO: describe the steps taken in dataset construction tool, maybe also some technical details.

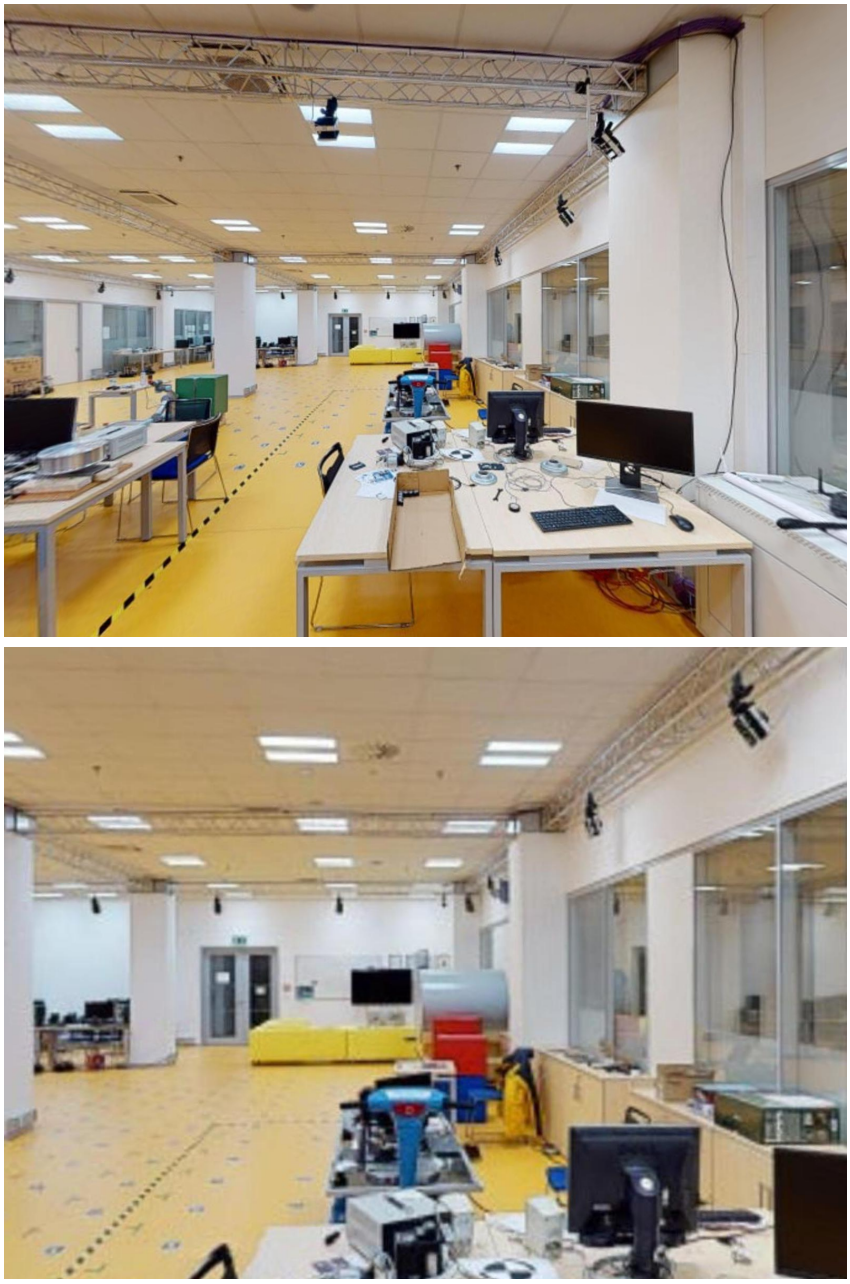


Figure 4.5: Visual quality comparision of the same cutout under different FoV. Top: horizontal FoV: 106.26°. Bottom: horizontal FoV: 60.00°. The image with a lower FoV contains a lot of artifacts and is of lower visual quality.

Chapter 5

Implementation

InLoc [1] authors provide a demonstration in MATLAB that operates on the InLoc dataset. I have taken this demonstration and adjusted it, so that it works on the InLocCIIRC dataset instead. I have added an evaluation script, that was missing from the original code. Although the evaluation of InLoc is handled by `visuallocalization.net`, this tool of course doesn't handle the newly created InLocCIIRC dataset yet.

The entire InLocCIIRC implementation should run on a multi-core machine with a GPU. The number of processing CPU threads can be up to 45 at a time. In order to do this, I was running the program on a CMP server. However, the GPU node prohibited the use of more than 8 CPU threads per user. So I had to split the implementation into 2 parts: in the first run, the GPU is used. In the latter run, no GPU is required, but a CPU with a lot of cores is used. The need for a GPU comes from the fact that we are using inference of NetVLAD neural network, which would take much longer on a CPU. This GPU restriction is present in InLoc implementation as well.

The original InLoc implementation uses point cloud projection in the pose verification step. The code for point cloud (PC) projection did not support variable point size. Because the models in my datasets are not dense (compared to those taken with the Faro 3D scanner), the projection can sometimes see through pillars or objects that are close to the camera. This is not desirable, as seeing what is behind the object can result in a different NetVLAD descriptor that is not similar to the query image. At first I have implemented PC projection with a point size parameter, but the problem is that it does not support headless[1] rendering (**TODO: or is**

[1]Headless rendering is rendering on a computer where the rendering program is not attached to

it software rendering actually?). I ended up using a mesh model projection instead of a point cloud projection in the point verification step. I am using existing software packages to achieve this (pyrender, trimesh, open3D). My projectMesh method supports headless rendering. Unfortunately, it is very demanding - requires 14 GB RAM and it also takes time to load the dataset into memory. Of course, one would cache the model in memory and call the render functions. However, this would require non-trivial implementation changes, at least in the demo, because the implementation is in MATLAB and the projectMesh routine is in Python.

A major change to the implementation was adding support for sequential queries. Currently the code supports specifically sequential queries from HoloLens. **TODO: In fact, the name InLocCIIRC_demo is not very accurate, because it is not really a implementing the InLoc paper** [1]. But the pose estimation algorithms are indeed based on InLoc. To estimate poses of sequential queries from HoloLens, poses from HoloLens must be provided. These poses are estimates of the ground-truth camera poses, and are computed by HoloLens itself. There are two approaches how the sequential nature of query sets is leveraged. Both approaches depend on a parameter k. We want to estimate the camera pose for each query in the query sequence. At each such query, we consider a segment of queries, such that the last query in the segment is the currently processed query. Constant k defines how long the segment is.

The first approach is called SequentialPV. It only leverages the other queries (in the segment) in the pose verification step. This approach aims to be more robust than the non-sequential one, by providing more evidence: the projection quality for all queries in the segment is considered and compared to the input query images. How is this done? We have top 10 camera poses (given by P3P in the pose estimation step). These poses are the estimated poses of the current query. Next, we have camera pose estimates for every query in the segment, provided by HoloLens. Those poses are wrt Omega. Therefore, I convert the poses from HoloLens from Omega to World, by aligning the two poses of the last query in the segment. The two poses are:

- the camera pose estimate (wrt World) provided from pose estimation step,
- the camera pose estimate (wrt Omega) provided from HoloLens.

To match the two poses, we just need to compute a linear transformation (rotation, translation). With this transformation, the other poses from HoloLens are converted from Omega to World. With all camera pose estimates being with respect to World coordinate system, I run the pose verification step. The pose verification step returns

a physical display.

a score, symbolizing the quality of the input query image and the reconstructed query image. I sum all the scores in the segment. **TODO: mention other ways, like mean, maximum**. This is done for those top 10 poses from the pose estimation step. At the end, I choose the candidate with highest score to represent the final camera pose estimate. This approach is a basic way to leverage the fact that the queries were taken in a sequence (and captured with HoloLens).

Approach two is called MultiCameraPose. We want to estimate pose of each query in the sequence by taking into account all poses and correspondences in the current segment. The camera pose estimates (wrt Omega) are taken from HoloLens. The 2D-3D correspondences are computed using the geometric verification step (and 2D to 3D transformation in parfor_densePE. **TODO: explain this better**) in InLoc. Given these data, an external program called *MultiCameraPose* [13] processes them and returns the camera pose estimates wrt World. The program contains an implementation of gsP4P [11]. For details on the MultiCameraPose program and gsP4P, please see Chapter 3. I store all returned camera rig poses, as the main result of the pose estimation step. In the pose verification step, all the estimated poses within a particular segment are evaluated (score is computed). Again, the candidate segment with the highest cumulative (summed up) score is selected. The last pose from the estimated poses in the segment is selected to be the final camera pose estimate for current query.

There is an important change when MultiCameraPose is used, compared to processing non-sequential queries. To understand that, let me first describe how the pose estimation step works in the non-sequential case:

1. We are given top 100 candidate cutouts for each query. These cutouts aim to be visually similar to the query. They were constructed using the input score matrix.
2. Query and cutout features are extracted.
3. Geometric verification is executed for all query-cutout pairs. This gives us 2D-2D correspondences aka “inliers” (some of which are inaccurate).
4. The top 100 candidates are re-ranked and sorted, so that query-cutout pairs with the highest number of inliers are preferred. If the number of inliers is the same, the original input score is used on top of it (floating point value between zero and 1).
5. Top 10 candidate cutouts for each query are chosen.
6. Each query-cutout pair and its 2D-2D correspondences are processed. Because one of 2D corresponding point sets lies in the cutout image, we can extract its corresponding 3D points (the dataset provides depth and 3D point of every

cutout pixel). The query-cutout 2D-3D correspondences are "fed" into P3P. The camera pose is estimated.

7. We now have 10 candidate pose estimates for every query.

In the MultiCameraPose approach, the segments have length k>1. We need to decide how to choose, for each query, top 10 *query-cutout segments*. The candidates will then be processed further using pose estimation and verification. Recall that the last query in the segment is always the one currently being processed (the one for which we want the camera pose). My current implementation does the following:

1. We have the re-ranked and sorted top 10 candidate cutouts for each query, as described in step 5 of the non-sequential pose estimation approach.
2. Generate all possible query-cutout segments of length k. There are 10^k possibilities.
3. Because k is expected to be no more than 5, we can easily generate all the combinations.
4. Every query-cutout has a score assigned, as described in step 4 of the non-sequential algorithm. I simply choose the combinations which have the cumulative (summed up) score the highest. Top 10 combinations are selected.

The algorithm in step 4 may be a bit problematic for two reasons. First, some query-cutout pairs may naturally have more inliers (on average) than others. It might be sub-optimal to sum those scores. Instead, e.g. an average or a median should be considered. The second issue is **TODO: in evaluation, where I discuss the source of errors, mention that issue where suboptimal query-cutout pairs are chosen in the top 10 combinations having a (small) negative impact on the accuracy**. The problem is that selecting 10 combinations from 10^k is not enough. But increasing the number of chosen top combinations is currently not possible, because pose verification is so slow. **TODO: Viz OneNote**.

TODO: How I am dealing with queries-cutouts not having the same aspect ratio. TODO: describe other changes from the original demo.

5.1 Pseudocode

```
mdoe = 'non-sequential' or 'sequentialPV' or 'MultiCameraPose'
segmentLength = 3 # aka 'k'; considered in 'sequentialPV', 'MultiCameraPose' modes
topRetrieval = 100
topGV = 10
topPE = 10
topPV = 1
neuralNet = NetVLAD()
coarseLayer = 'conv5'
fineLayer = 'conv3'

def addSecondaryQueries(ImgList, score, queryNames, cutoutNames):
  # primary query is a query user requested to perform pose estimation on.
  # secondary queries are part of the k-segments of primary queries.
  # they need to be added in 'MultiCameraPose' mode to be processed by
  # poseEstimation() onward
  implementationDetail()

def retrieval(score, queryNames, cutoutNames):
  ImgList = list()
  for i in len(queryNames):
    queryName = queryNames
    ImgList[i].queryname = queryName
    sortedScores, ind = sort(score[queryName].scores, 'descend')
    ImgList[i].topNname = cutoutNames[ind[0:topRetrieval]]
    ImgList[i].topNscore = sortedScores[0:topRetrieval]
  if mode == 'MultiCameraPose'
    addSecondaryQueries(ImgList, score, queryNames, cutoutNames)
  return ImgList

def extractFeatures(image):
  image = neturalNet.averagingImageNormalization(image)
  allLayerResults = neuralNet.forward(image)
  features = allLLayerResults # we need features from different layers
  return features

def loadQueryImageCompatibleWithCutouts(queryImage):
  queryImage = padImageByAddingRowsToMatchCutoutAspectRatio(queryImage)
  queryImage = scaleImageToMatchCutoutDimensions(queryImage)
  return queryImage

def adjustInliersToMatchOriginalQuery(queryTentatives, queryDimensions, cutoutDimensions):
  # reverts loadQueryImageCompatibleWithCutouts(...)
  return implementationDetail(...)

def buildFeatures(ImgList):
  features = list()
  for i in range(len(ImgList)):
    queryName = ImgList[i].queryname
```

```
    thisQueryFeatures = list() # query image features followed by
        'topRetrieval' cutout features
    queryImage = loadImage(queryName)
    queryImage = loadQueryImageCompatibleWithCutouts(queryImage)
    thisQueryFeatures.append(queryImage)
    for j in topRetrieval:
      cutoutName = Imglist[i].topNname[j]
      cutoutImage = loadImage(cutoutName)
      thisQueryFeatures.append(extractFeatures(cutoutImage))
    features.append(thisQueryFeatures)
  return features

def coarseToFineMatching(queryFeatures, cutotuFeatures):
  queryCoarseFeats = getFeaturesAtLayer(queryFeatures, coarseLayer)
  cutoutCoarseFeats = getFeaturesAtLayer(cutoutFeatures, coarseLayer)
  queryFineFeats = getFeaturesAtLayer(queryFeatures, fineLayer)
  cutoutFineFeats = getFeaturesAtLayer(cutoutFeatures, fineLayer)
  f1 = queryFineFeats
  f2 = cutoutFineFeats
  match12 = findNearestMatches(queryCoarseFeats, cutoutCoarseFeats)
  return f1, f2, match12

def sortImgListRowByHighestScores(ImgListRow)
  for i in len(queryNames):
    sortedScores, ind = sort(ImgListRow[i].topNscore, 'descend')
    ImgListRow[i].topNname = ImgListRow[i].topNname[ind]
    ImgListRow[i].topNscore = ImgListRow[i]topNscore[ind]
  return ImgListRow

def geometricVerification(ImgList, features):
  NewImageList = ImgList.copy()
  for i in range(len(ImgList)):
    thisQueryFeatues = features[i]
    queryName = ImgList[i].queryname
    parfor j in range(topRetrieval):
      cutoutName = Imglist[i].topNname[j]
      queryImgFeatures = thisQueryFeatures[0]
      cutoutImgFeatures = thisQueryFeatures[1+j]
      match12, f1, f2 = coarseToFineMatching(queryImgFeatures,
          cutoutImgFeatures)
      inls12 = denseRansac(f1, f2, match12)
      save(queryName, cutoutName, f1, f2, match12, inls12)
      NewImgList[i].topNscore[j] += len(inls12) # NOTE: the previous
          scores were between zero and one
    NewImageList[i] = sortImgListRowByHighestScores(NewImgList[i])
  return NewImageList

def getActualSegmentLength(idx, desiredSegmentLength, ImgList):
  return
      getSegmentLengthSuchThatSegmentQueriesAreWithinSequenceBounds(idx,
      desiredSegmentLength, ImgList)

```

```
96  def getCandidateIdx():
97    # for each query, we have multiple candidate solutions.
98    # parfor_densePE and parfor_densePV functions must be executed on
99    # all of those candidates
100   return implementationDetail()
101
102 def poseEstimation(ImgList):
103   features = buildFeatures(ImgList)
104   ImgList = geometricVerification(ImgList, features)
105   treatQueriesSequentually = mode == 'MultiCameraPose'
106   if not treatQueriesSequentually:
107     desiredSegmentLength = 1
108   ImgListSequential = keepPrimaryQueriesOnly(ImgList)
109   for i in range(len(ImgListSequential)):
110     actualSegmentLength = getActualSegmentLength(i, desiredSegmentLength,
          ImgListSequential)
111     combinations = permuteIndices([0:topGV], actualSegmentLength)
112     scores = computeScoresForSegmentCombinations('cummulative-sum')
113     ind = findBestCombinations(scores, topPE)
114     updateTopCutoutsAnsScoresInTheSegment(ImgListSequential[i], scores,
          ind)
115   if treatQueriesSequentually:
116     posesFromHoloLens = getPosesFromHoloLens()
117   else:
118     posesFromHoloLens = list()
119
120   parfor i in len(ImgListSequential):
121     candidateIdx = getCandidateIdx()
122     parfor_densePE(ImgListSequential, i, posesFromHoloLens, candidateIdx)
123
124   for i in len(ImgListSequential)
125     ImgListSequential[i].Ps = list(size=topPE) # estimated poses in the
          segment, for topPE combinations
126     for j in topPE:
127       ImgListSequential[i].Ps[j] = load_parfor_densePE_segment_poses(i, j)
128
129   return ImgListSequential
130
131 def parfor_densePE(ImgList, idx, posesFromHoloLens, candidateIdx):
132   actualSegmentLength = getActualSegmentLength(idx,
        implementationDetail(), ImgList)
133   Ps = list(size=actualSegmentLength)
134   useP3P = segmentLength == 1
135   if invalidPosesDueToDelay(posesFromHoloLens):
136     useP3P = True
137   for j in segmentLength:
138     f1, f2, match12, inls12 = load(queryName, cutoutName, candidateIdx)
139     queryTentatives = f1[inls12[0]]
140     cutoutTentatives = f2[inls12[2]]
141     queryTentatives = upscale(queryTentatives, cutoutSize)
142     queryTentatives = adjustInliersToMatchOriginalQuery(queryTentatives,
          queryDimensions, cutoutDimensions)
```

```
143       correspondences = build2D3DCorrespondences(queryTentatives,
              cutoutTentatives)
144
145     if useP3P:
146       P, inls = P3P(correspondences)
147       Ps[end] = P
148       save(inls, candidateIdx=candidateIdx)
149     else:
150       Ps = multiCameraPose(correspondences, posesFromHoloLens)
151
152     save(Ps, candidateIdx=candidateIdx)
153
154   def convertHLPosesToBeWrtCurrentQueryPoseEstimate(posesFromHoloLens):
155     # it should be clear how to do this from my textual description in the
          Implementation Chapter 5
156     return implementationDetail()
157
158   def parfor_densePV(ImgList, queries, idx, candidateIdx):
159     parentQuery = queries[idx]
160     queriesInSegment = getQueriesInSegment(parentQuery)
161     cutouts = getCutoutsInSegment(parentQuery)
162     Ps = load(parentQuery, candidateIdx=candidateIdx)
163     for i in range(len(queriesInSegment))
164       query = queriesInSegment[i]
165       cutout = cutouts[i]
166       P = Ps[i]
167       queryImage = loadQueryImage(query)
168       synthQueryImage = projectPose(P)
169       error = compute_DSIFT_error(queryImage, synthQueryImage)
170       save(parentQuery, query, cutout, error, synthImage,
              candidateIdx=candidateIdx)
171
172   def poseVerification(ImgList):
173     PV_list = setUpListForPoseVerificationProcessing(ImgList)
174     if mode == 'sequentialPV':
175       posesFromHoloLens = getPosesFromHoloLens()
176       posesFromHoloLens =
              convertHLPosesToBeWrtCurrentQueryPoseEstimate(posesFromHoloLens)
177       addPosesFromHoloLensForPoseVerificationProcessing(PV_list,
              posesFromHoloLens)
178     spaces = getSpacesAtWhichPoseEstimatesAre(ImgList)
179     for space in spaces:
180       queries = getQueriesInSpace(ImgList, space)
181       parfor i in len(queries):
182         candidateIdx = getCandidateIdx()
183         parfor_densePV(ImgList, queries, i, candidateIdx)
184
185     ImgList = reRankSortAndChooseTop(topPV)
186
187   def evaluate(ImgList):
188     # chooses top 1 poseVerification results for each query
189     visualEvaluationQueryByQuery(ImgList)
```

```
190     visualEvaluationQuerySegments(ImgList)
191     computeTranslationAndOrientationErrorsWrtReferencePoses(ImgList)
192     showLocalizationAccuracyGivenThresholds()
193     showErrorStatistics()
194
195   def main()
196     score, queryNames, cutoutNames = initialize()
197     assertNonSequentialModeUsedIfQuerySetIsNonSequential()
198     # score represents query-cutout score matrix
199     ImgList = retrieval(score, queryNames, cutoutNames)
200     ImgList = poseEstimation(ImgList)
201     ImgList = poseVerification(ImgList)
202     evaluate(ImgList)
```

Algorithm 5.1:]InLocCIIRC pseudocode. **TODO: pseudocode review.**

Note that in the current actual source code, I do not have the 'non-sequential' mode. Instead, it is determined by choosing 'MultiCameraPose' mode and setting segmentLength to 1.

Chapter 6

Evaluation

6.1 Experiment design

Performance of the implemented solution had to be evaluated quantitatively for all the three main methods: Non-sequential method, sequentialPV method and MultiCameraPose method. The two latter methods are designed to work with segments of queries, therefore different segment lengths, denoted by contant $k > 1$ were evaluated. Some of the promising methods were also visualized for human-friendly qualitative evaluation. The visualizations are used to better understand the sources of errors, which are described in section 6.4.

In order to measure how the InLocCIIRC algorithm is performing, I have measured the percentage of correctly localized poses within a threshold from a reference pose. Absolute position difference threshold is one of the following values, with decreasing difficulty: 0.25m, 0.50m, 1.00m. Angular threshold is set to 10°. Another metric is to compute statistics on the translation and orientation errors. For this, the mean, median, and standard deviation (std) were chosen.

A descriptive way to compare multiple methods is to compare percent of correctly localized queries, as the translation threshold increases (the orientation threshold is fixed).

I provide two kinds of visualization. The first one shows, for a subset of queries, how they are processed - the closest cutout found, the inliers used to reconstruct

the camera pose, and an error map. This is also useful when determining why InLocCIIRC performs poorly on certain queries. The second visualization shows a top level view of the physical environment. The queries, estimated query poses and sweeps are drawn.

6.2 s10e query set

Evaluation results of the non-sequential s10e query set are shown in this section. Table 6.1 shows the errors in pose estimation for individual queries. Rows with a NaN entry mean that densePE returned a NaN P matrix, we do not have a reference pose for the query, or the estimated pose was in a different space than the reference query pose. Table 6.2 shows the performance under the various thresholds. The InMap/OffMap performance is also shown. Error statistics are shown in table 6.3. Figure 6.2 shows how the localization accuracy changes given increasing translation error threshold.

Figure 6.1 shows example queries, how they are being processed and what is the localization result.

Figures 6.3 and 6.4 depict the dataset including the localization results.

Query ID	InMap	Translation [m]	Orientation [°]
1	Yes	0.0880	1.5386
2	Yes	0.1832	1.2987
3	No	0.1709	1.4430
4	Yes	0.1065	1.2635
5	Yes	0.2185	0.4119
6	No	0.1324	1.1850
7	Yes	0.0752	0.9260
8	No	0.2015	1.2524
9	No	0.1167	1.0074
10	No	0.1302	0.4764
11	No	0.1152	0.7651
12	No	0.2927	1.0733
13	No	0.9610	13.8282
14	No	0.1324	2.0652
15	No	0.1320	1.6251
16	No	0.3284	4.1566
17	No	0.0745	1.6222
18	No	0.1053	0.6402
19	No	0.0259	1.4572
20	No	0.0788	0.3069
21	No	0.1652	1.5283
22	No	0.2209	1.3378
23	No	0.1552	2.8651
24	No	0.6788	2.7907
25	No	0.0944	1.0374
26	No	0.4796	1.6020
27	No	0.1465	2.4818
28	No	0.0779	0.8901
29	No	0.0538	1.4261
30	No	0.0305	2.0371
31	No	0.1258	1.1690
32	No	0.1369	1.9361
33	No	0.2868	2.6586
34	No	0.2834	3.3971
35	No	2.3826	0.4984
36	No	0.1939	2.0936
37	No	0.1471	2.6406
38	No	0.0761	1.4153
39	No	0.2338	2.6006
40	No	7.8370	153.0940

Table 6.1: Pose estimation errors on query images.

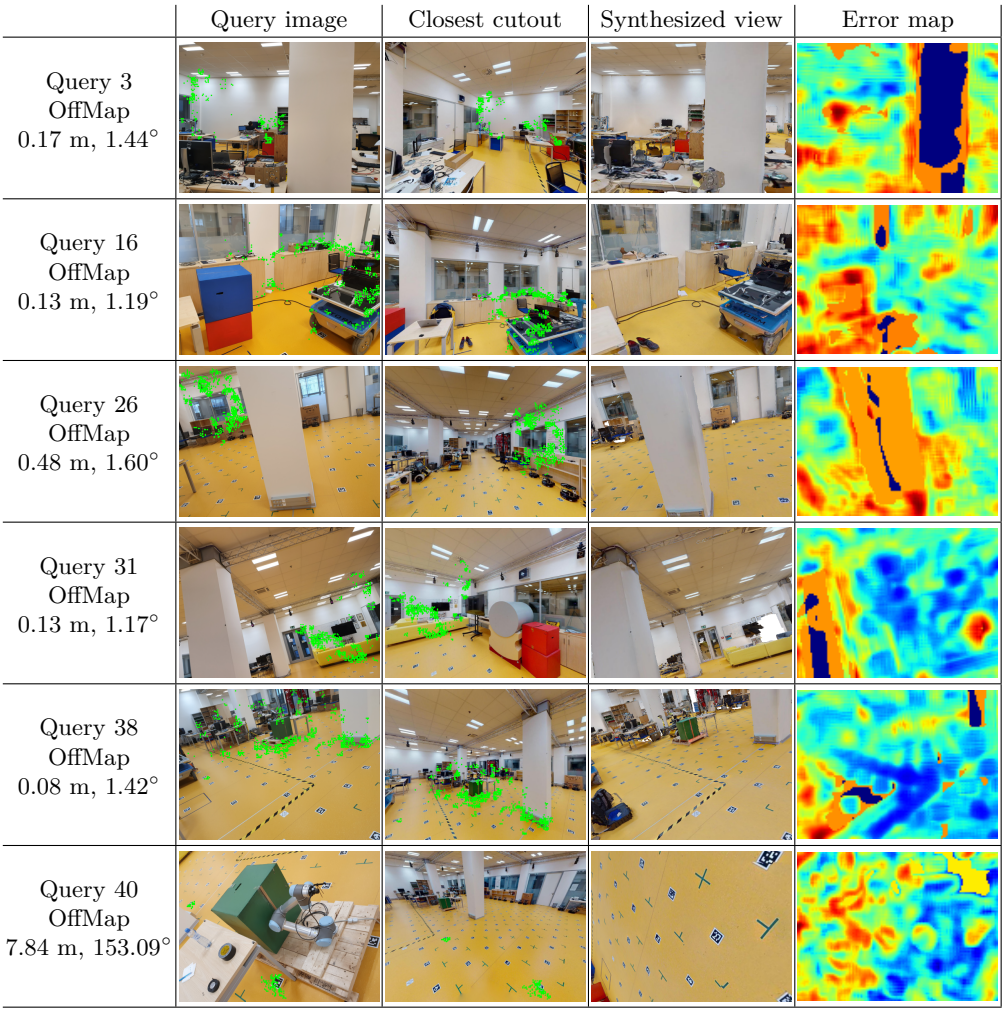


Figure 6.1: Qualitative comparison of query localization. From left to right: Query name and localization error (meters, degrees), query image, the best matching database image, synthesized view at the estimated pose, error map between the query image and the synthesized view. Green dots are the inlier matches obtained by P3P-LO-RANSAC. The majority of query images shown here are well localized within 0.5 meters and 5.0 degrees. All of the shown queries are OffMap, to test challenging estimation scenarios. InLocCIIRC struggles to find correct inliers on query 40, see section 6.4 for an investigation (**TODO**).

Threshold	InLoc	**InLocCIIRC**	InMap	OffMap
0.25m	38.9%	**77.50%**	100.00%	74.29%
0.50m	56.5%	**90.00%**	100.00%	88.57%
1.00m	69.9%	**92.50%**	100.00%	91.43%

Table 6.2: Evaluation of performance of localization methods. The method in the first column was run on InLoc dataset. The second column method was run on InLocCIIRC dataset. Percentage rate of correctly localized queries within given threshold is shown. Angular threshold is equal to 10° in every row. The last two columns belong to InLocCIIRC method. InMap queries are queries for which we have a similar cutout in the dataset. **TODO: evaluate estimated poses by procrustes with poses from HoloLens** (if the queries are from HoloLens).

Statistics \ Error type	Translation [m]	Orientation [°]
Mean	0.44	5.70
Median	0.14	1.45
Standard deviation	1.26	24.00

Table 6.3: Statistics of the s10e pose estimation errors. InLocCIIRC got completely lost 0 out of 40 times. Not included in the mean/median/std errors. Errors are computed by comparing InLocCIIRC pose estimates with reference poses. Notice that the deviations are high. This is caused by the query 40 performing extraordinarily poorly.

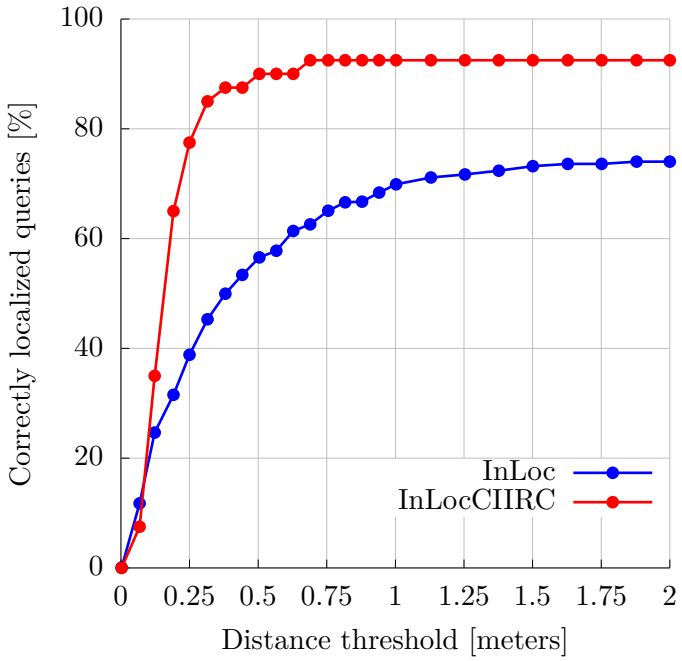


Figure 6.2: Comparison between InLoc and InLocCIIRC on their respective datasets. The x-axis describes the maximum allowed translation error. The angular threshold is set to 10°.

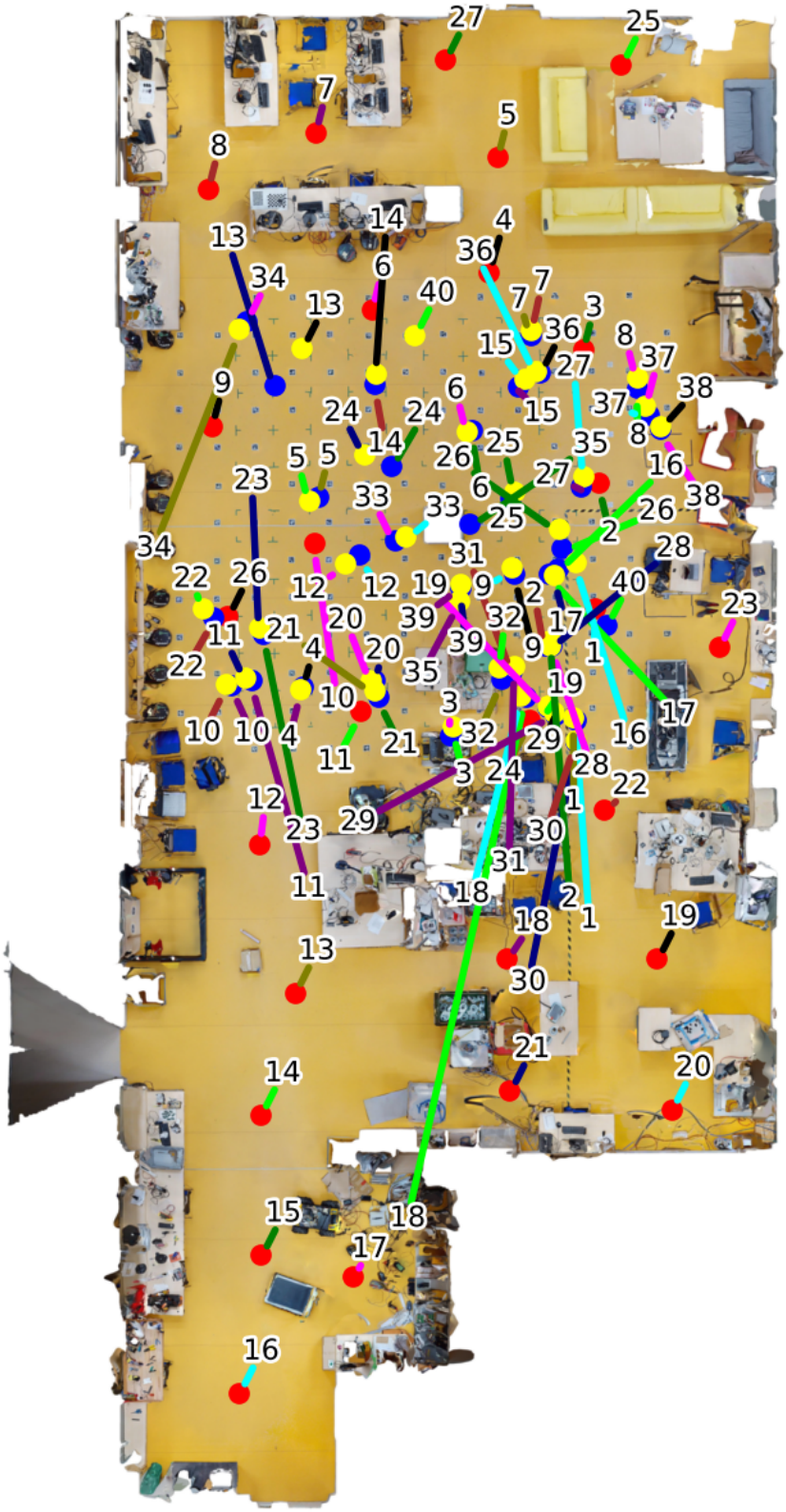


Figure 6.3: View on the floor plan of room B-315. Red dots: sweeps. Blue dots: queries. Yellow dots: estimated query poses.

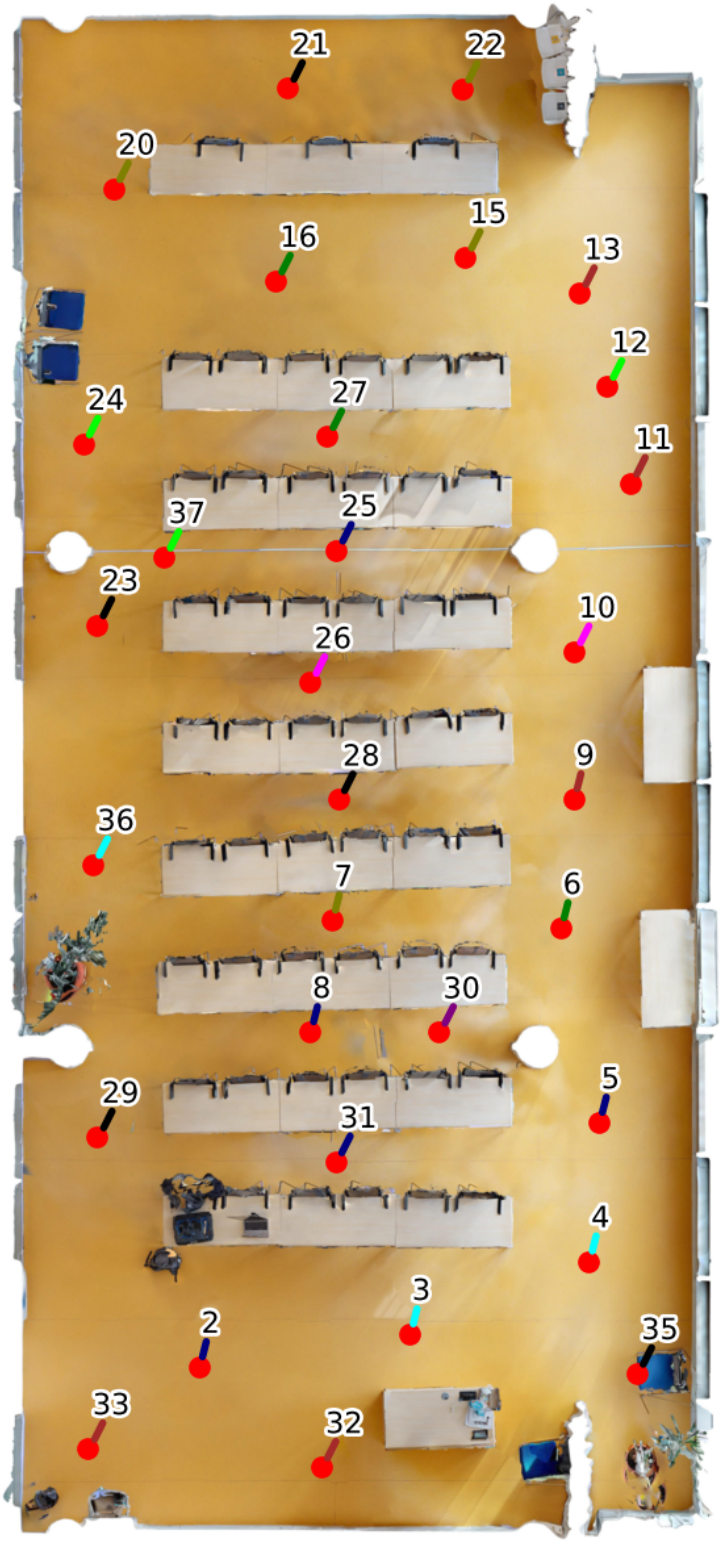


Figure 6.4: View on the floor plan of room B-670. Red dots: sweeps. Blue dots: queries. Yellow dots: estimated query poses. No s10e queries were incorrectly localized to this room.

6.3 HoloLens1 query set

6.3.1 Summary

Method \ Threshold	0.25m	0.50m	1.00m
k=1 (non-sequential)	63.80%	81.90%	85.89%
sequentialPV, k=2	63.80%	82.52%	86.50%
sequentialPV, k=3	63.80%	83.44%	86.50%
sequentialPV, k=4	61.96%	82.52%	85.28%
MultiCameraPose, k=2	**68.41%**	**83.74%**	**87.12%**
MultiCameraPose, k=3	**68.41%**	81.60%	86.20%
MultiCameraPose, k=5	67.18%	80.67%	85.58%
HoloLens	**84.36%**	**97.55%**	**97.55%**

Table 6.4: Evaluation of performance of localization methods on HoloLens1 query set. Ran on the InLocCIIRC dataset obviously. Percentage rate of correctly localized queries within given threshold is shown. Angular threshold is equal to 10° in every row. The HoloLens method are the poses provided by HoloLens tracking itself, after being converted to be wrt World coordinate system. As can be seen, it is superior to all the custom methods I have tried.

Method \ Statistics	Mean		Median		Std	
k=1	0.52m	3.62°	0.18m	2.01°	1.64m	11.33°
sequentialPV, k=2	0.52m	3.05°	0.17m	2.06°	1.64m	5.21°
sequentialPV, k=3	**0.45m**	3.04°	0.18m	2.02°	**1.36m**	5.21°
sequentialPV, k=4	0.60m	3.61°	0.19m	2.02°	1.93m	11.47°
MCP, k=2	0.53m	2.76°	**0.16m**	**1.85°**	1.84m	4.41°
MCP, k=3	0.54m	**2.68°**	**0.16m**	1.99°	1.85m	**3.06°**
MCP, k=5	0.60m	3.59°	0.17m	2.02°	1.92m	11.26°
HoloLens	**0.15m**	**2.09°**	**0.14m**	**1.51°**	**0.08m**	**1.69°**

Table 6.5: Statistics of the HoloLens1 pose estimation errors. InLocCIIRC got completely lost 29 out of 350 times for all methods (except the HoloLens method). The HoloLens method got completely lost 6 out of 350 times, which is caused by the HoloLens delay (see table 4.3). The *completely lost* cases are not included in the mean/median/std errors. Errors are computed by comparing InLocCIIRC pose estimates (or the pose estimates from HoloLens converted to be wrt World CS) with reference poses. The errors in [m] units are translation errors and the errors in [°] units are orientation errors. Lowest errors are highlighted in bold. MCP stands for MultiCameraPose. The original HoloLens method is superior to all the custom methods I have tried. Note that if the estimated poses were compared to the (unknown) ground-truth poses, the errors would likely be even lower, as discussed in the Reference poses section 4.1.

6.3.2 Best custom method

Judging from the results above, the best performing custom method is MultiCameraPose with sequence length $k = 2$.

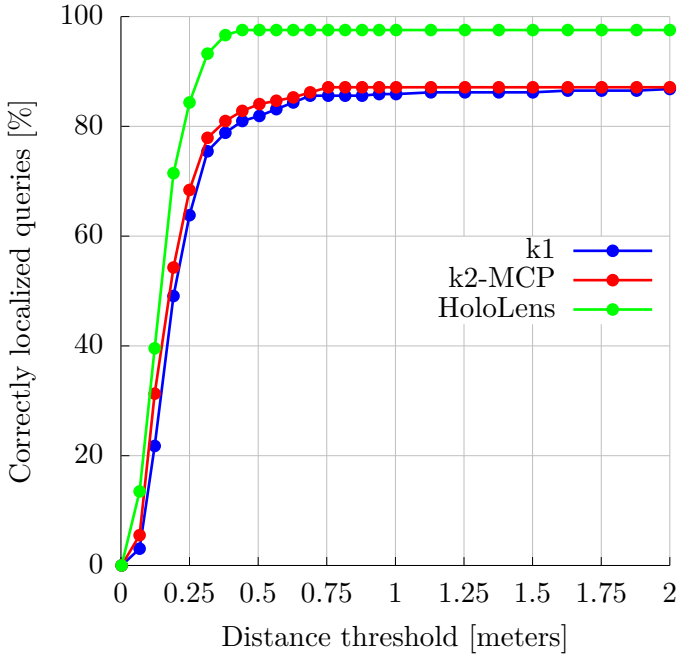


Figure 6.5: Evaluation of methods on the HoloLens1 query set. Comparison between the baseline method ($k = 1$, i.e. non-sequential) with the best performing custom method ($k = 2$, MultiCameraPose). The original HoloLens method, that we are aiming to surpass is also shown. The x-axis describes the maximum allowed translation error. The angular threshold is set to 10°.

Figure 6.7 depicts the dataset including a subset of localization results (every 20th HoloLens query is rendered). View of room B-670 is not shown, as for this subset of results, it looks the same as in case of s10e query set, see figure 6.4. This means that none of the queries in the subset were incorrectly localized in room B-670.

Figure 6.6: Qualitative comparison of query localization. From left to right: Query name and localization error (meters, degrees), query image, the best matching database image, synthesized view at the estimated pose, error map between the query image and the synthesized view. Green dots are the inlier matches obtained by geometric verification. The pose estimation of query 84 is not completely wrong by human standards. InLocCIIRC matched the query image with a very similar cutout image, that is, however, at another location. Although this query is InMap, the chosen cutout is not the one that forms the InMap propery. Note that the query images have a different aspect ratio than the cutout images. The error maps not shown to save space.

Figure 6.7: View on the floor plan of room B-315. Red dots: sweeps. Blue dots: queries. Yellow dots: estimated query poses. Every 20th HoloLens1 query rendered.

6.4 Sources of errors

6.4.1 Previous queries have meaningful correspondences but current query does not have any correspondences

This was observed on MultiCameraPose, k=2 experiment. This results in InLocCIIRC completely being lost (returning NaN estimated pose), thus limiting the number of correctly localized queries given translation/orientation thresholds. In this scenario, the queries in the segment prior to the current query being processed have 2D-3D correspondences (found using geometric verification). Furthermore, those queries look meaningful upon manual inspection. However, we are interested in the current query, which does not have any correspondences. MultiCameraPose does not support a rig containing a camera for which there are no correspondences. Of course, we cannot use P3P on the current query, without knowing the query-cutout correspondences. Potential solution to this problem is: use the last estimated non-NaN pose in a sequence of queries ending with the current query. Limit the number as to how far into history to go. I did not have time to implement it. Known affected queries in HoloLens1 query set: 88, 122, 148, 231, 233, 236, 315, 319, 341. Why did we find no correspondences at those affected queries? For query 122 it is understandable - there was a very fast movement. For query 174 - it is somehow difficult, even the preceding queries 170-173 were hard to estimate (resulting poses were not NaN, but the errors from reference poses were high). However, for some affected queries, namely query 88, 148 and 231, a problem was discovered. The next subsection describes the problem.

6.4.2 Bad input score matrix

This was observed on non-sequential (k=1) and MultiCameraPose, k=2 experiments. Known affected queries: 88, 148, 231[1]. This issue probably affects more queries than is currently known by me. It causes no 2D-3D corresponences to be found. It is a problem, that currently causes NaN pose estimate for the affected queries. But it can also lower estimation accuracy for the sucessor queries, if the affected query is considered within its segment. This is because currently, P3P is used (non-sequential pose estimation), if some of the queries in a segment have no correspondences. For those 3 known affected queries, investigation revealed that the chosen cutout (i.e. the top one in pose verification output) from the previous query was not even considered in the top 100 cutouts in the pose estimation step. The reason the previous query's

[1]This query is somewhat blurry, which may also have an inpact.

chosen cutout was picked is that the queries have not changed much during the two frames. The score for those cutouts was:

Query ID	Ranking of the previous query's chosen cutout
88	714
148	138
231	518

Table 6.6: The ranking after sorting all cutouts for a given query by highest score. Only 100 make it to the pose estimation step, others are not considered. This suggests that the scores are not completely correct.

6.4.3 Hard to pick top 10 combinations for non-trivial segments

Geometric verification step chooses top 10 cutouts for each query based on the highest number of inliers. The wrong ones would normally be filtered by pose verification. However, if segments of length k>1 are used, the good query-cutout pairs simply won't make it to the top 10 list. This is because we are only choosing top 10 combinations from 10^k possible combinations. **TODO: use of the word "combinations" is technically incorrect/inaccurate.** As a result, average correspondences in queries within a non-trivial segment have lower quality (many matches are incorrect/imprecise) **TODO: show the examples from OneNote - I had to call queryPipeline again. Cannot reproduce! Honestly cannot tell which method has more non-corresponding matches!**. There is no easy solution to this problem. Making pose estimation return significantly more than top 10 candidates for each query will have a performance impact, because the pose verification step is already time consuming.

6.4.4 No HoloLens poses

Due to the delay (see table 4.3), some of the queries by the end of the HoloLens1 and HoloLens2 sequences do not have a pose estimated from HoloLens. In such a case we have to resort to using standard P3P, which performs (on average) worse than MultiCameraPose. Hopefully the delay was only caused by the software extracting the data from HoloLens and the delay is not actually present in real use. If it is present, it is a problem as the techniques based on InLoc described in this paper would not work in real-time.

Chapter 7

Conclusion

I have created a new dataset suitable for indoor visual localization either on single RGB images or on a sequence of query images and localization data from HoloLens. I have adjusted the original InLoc implementation **TODO: github citation** and made it work on the newly acquired dataset. The performance on the non-sequential s10e query set is very good (compared to results in InLoc paper). This is likely caused by the fact that my dataset is much smaller than the InLoc dataset. I have also implemented two novel methods that are based on InLoc [1] - the sequentialPV method and the MultiCameraPose method. It was expected that the sequentialPV method would not to perform very well compared to HoloLens tracking. The MultiCameraPose method is more accurate than both the baseline InLoc method and the sequentialPV method. The resulting estimated poses are usable. However its performance is still significantly below the precision of HoloLens tracking itself. It is not clear why the new MultiCameraPose method is not performing that well. In the previous chapter, I have described known sources of errors, a lot of which can be targeted in a future work. This will certainly improve the evaluation performance.

7.1 Future work

Improve the accuracy of the MultiCameraPose method by fixing known sources of errors. Spend extra time to analyze why there are inaccurate poses for certain queries and suggest an enhancement. The work on the HoloLens2 sequence should be continued - we need to compute reference poses. The code is there, but currently we are missing more manually set-up 2D-3D correspondences. Also, the work on synthetic dataset generation using Habitat AI shall be continued. Although it

cannot give us data from HoloLens tracking, it can be used to generate new indoor localization datasets, without the need for expensive equipment (such as a Matterport scanner). There are extra parameters such as setting up the cutout horizontal field of view, **TODO: dslevel**, MultiCameraPose software [13] parameters and more. The accuracy of the reference poses wrt ground truth poses shall be also improved.

Appendix A

Bibliography

[1] Taira, H.; Okutomi, M.; et al. InLoc: Indoor Visual Localization with Dense Matching and View Synthesis. In *CVPR*, 2018.

[2] Pajdla, T. *Elements of Geometry for Computer Vision*. 2020.

[3] Arandjelović, R.; Gronat, P.; et al. NetVLAD: CNN architecture for weakly supervised place recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[4] Wijmans, E.; Furukawa, Y. Exploiting 2D Floorplan for Building-scale Panorama RGBD Alignment. In *Computer Vision and Pattern Recognition, CVPR*, 2017. Available from: `http://cvpr17.wijmans.xyz/CVPR2017-0111.pdf`

[5] Fischler, M.; Firschein, O. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. In *Readings in Computer Vision*, Morgan Kaufmann, 1987, ISBN 978-0-08-051581-6, pp. 726 – 740, doi:https://doi.org/10.1016/B978-0-08-051581-6.50070-2.

[6] Arandjelović, R.; Zisserman, A. Three things everyone should know to improve object retrieval. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 2911–2918.

[7] Liu, C.; Yuen, J.; et al. SIFT Flow: Dense Correspondence across Scenes and Its Applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 33, no. 5, 2011: pp. 978–994.

[8] Fukushima, K. Neocognitron: a self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, volume 36, no. 4, 1980: p. 193—202, ISSN 0340-1200, doi:10.1007/bf00344251. Available from: `https://doi.org/10.1007/bf00344251`

[9] Schmidhuber, J. Deep learning in neural networks: An overview. *Neural Networks*, volume 61, 2015: pp. 85 – 117, ISSN 0893-6080, doi:https://doi.org/10.1016/j.neunet.2014.09.003.

[10] Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*, 2015.

[11] Kukelova, Z.; Heller, J.; et al. Efficient Intersection of Three Quadrics and Applications in Computer Vision. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[12] Walk, J.; Sattler, T.; et al. Beyond Controlled Environments: 3D Camera Re-Localization in Changing Indoor Scenes. In *Proceedings IEEE European Conference on Computer Vision (ECCV)*, 2020.

[13] Sattler, T.; Lučivňák, P. [online], 2020, [cit. 2020-09-14]. Available from: `https://github.com/lucivpav/MultiCameraPose`

[14] Hübner, P.; Clintworth, K.; et al. Evaluation of HoloLens Tracking and Depth Sensing for Indoor Mapping Applications. *Sensors*, volume 20, 02 2020: pp. 1021:1–23, doi:10.3390/s20041021.

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Lučivňák Pavel** Personal ID number: **435627**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence**

II. Master's thesis details

Master's thesis title in English:

Visual Localization with HoloLens

Master's thesis title in Czech:

Vizuální lokalizace pro HoloLens

Guidelines:

1) Review the state of the art in indoor visual localization, see [1,2] and references therein.
2) Adjust method [2] to local environment and image acquisition using HoloLens. Create new 3D data set for the local environment and evaluate the accuracy of the localization w.r.t. a ground truth in that environment.
3) Apply InLoc localization method on data from HoloLens, evaluate behavior and inaccuracies of the localization on this data. Investigate a possibility of using multiple images for improving the localization.
4) Demonstrate and evaluate the improved method for HoloLens localization.

Bibliography / sources:

[1] Arandjelović, R.; Gronat, P.; et al. NetVLAD: CNN architecture for weakly supervised place recognition. In IEEE Conference on Computer Vision and Pattern Recognition, 2016.
[2] Taira, H.; Okutomi, M.; et al. InLoc: Indoor Visual Localization with Dense Matching and View Synthesis. In 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, June 2018, ISSN 1063-6919, pp. 7199–7209, doi:10.1109/CVPR.2018.00752.
[3] Garg, R.; Kumar, B. V.; et al. Unsupervised CNN for single view depth estimation: Geometry to the rescue. In European Conference on Computer Vision, Springer, 2016, pp. 740–756.
[4] Zhang, Y.; Funkhouser, T. Deep Depth Completion of a Single RGB-D Image. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.
[5] Van Gansbeke, W.; Neven, D.; et al. Sparse and Noisy LiDAR Completion with RGB Guidance and Uncertainty. In 2019 16th International Conference on Machine Vision Applications (MVA), IEEE, 2019, pp. 1–6.

Name and workplace of master's thesis supervisor:

doc. Ing. Tomáš Pajdla, Ph.D., Applied Algebra and Geometry, CIIRC

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **04.02.2020** Deadline for master's thesis submission: **14.08.2020**

Assignment valid until: **30.09.2021**

doc. Ing. Tomáš Pajdla, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

