Mapping Building Outlines to 3D Point Clouds

Anurag Sai Vempati
Autonomus Systems Lab
ETH Zurich
avempati@ethz.ch

Wolf Vollprecht
wolfv@student.ethz.ch

Abstract

While point clouds are easy to obtain through processes such as Structure-From-Motion they only exist in a space for themselves. GPS coordinates, recorded at capture time of the images, can give a sparse and noisy reference to the true position of the point cloud. In this paper we present a method to register a point cloud on reference map data.

1. Introduction

With current state of the art technology, obtaining point clouds through structure from motion has become an easy task. Several attempts deal with a similar problem, such as ...

2. Segmenting the Point Cloud and generating 2D outline

Point clouds that we will be operating on are obtained through Structure from Motion (SfM) techniques. SfM techniques generate 3D structure from a sequence of 2D images that can either be manually taken or collected from online images warehouses like Google images, Flikr etc. We will be using one such software called Bundler [??]. Bundler does incremental reconstruction using Sparse Bundle Adjustment. It updates the structure as additional images are available, eliminating the need to wait for the system to generate structure from scratch which might take several hours. We used the Aachen dataset [??] containing point cloud generated by bundler using 4479 images taken with different consumer cameras.

For getting a 2D outline of the buildings, we need to be able to segment out the point clouds into parts that belong to a plane which is up-right (and hence shows up as a line in the 2-D map) and the parts that are outliers to these planes. It's also helpful to get rid of noisy points in the point cloud to eliminate drifts and errors further down the pipeline (corner detection, ICP etc.).

2.1. Bundler Parser

The aachen.out file is the output generated by Bundler (please see http://phototour.cs.washington.edu/bundler/bundler-v0.4-manual.html#S6 for a description of the output format) after registering all the images. It contains the camera calibration for every one of the 4479 images. Bundler file consists of the estimated scene geometry and has the following format:

```
# Bundle file v0.3
<num_cameras> <num_points>   [two integers]
<camera1>
<camera2>
   ...
<cameraN>
<point1>
<point2>
   ...
<pointM>
```

Each camera entry <cameraI> contains camera intrinsics and extrinsics int he following form:

```
<f> <k1> <k2>   [focal length, followed by distortion coeffs]
<R>             [camera rotation]
<t>             [camera translation]
```

Each point entry <pointJ> has the following form:

```
<position>      [3D position of the point]
<color>         [RGB color of the point]
<view list>     [a list of views the point is visible in]
```

A parser is written that can efficiently read the aachen.out file and store a part or the whole structure into a point cloud (along with RGB color attributes) that can be processed using point cloud library (PCL ??). We also store the camera locations $\mathbf{c} = -R^{'}t$ for estimating the ground plane.

2.2. Ground Plane and Point normal estimation

To be able to generate building 2D outlines, we first estimated the up-vector which is coplanar with the facades of the building and is normal to the ground. Our approach is based on the assumption that majority of the images are taken from ground level and thus fitting a plane through all the camera centers **c** should be a reasonable estimate of the ground plane. The smallest eigen vector of covariance matrix generated from camera positions is chosen as the up-vector. Fig 1 shows the camera positions and the estimated ground plane.

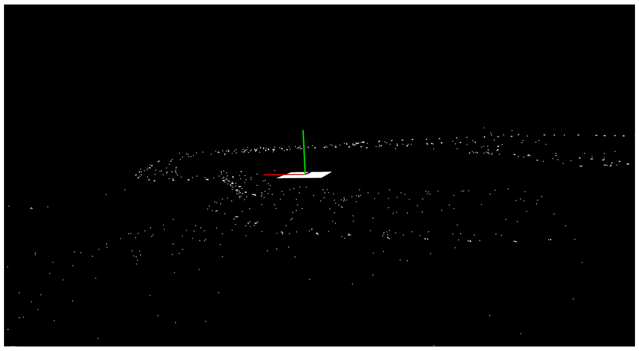


Figure 1: Camera positions and estimated ground plane

The point normals are required to distinguish points that belong to the building facades and the rest. It also helps to filter out the noisy points from the point cloud. point normals are estimated in a similar fashion as the up-vector by evaluating smallest eigen vector of the covariance matrix generated from k nearest neighbours of each point in the point cloud. Fig 2 shows point normals of a section of the point cloud.

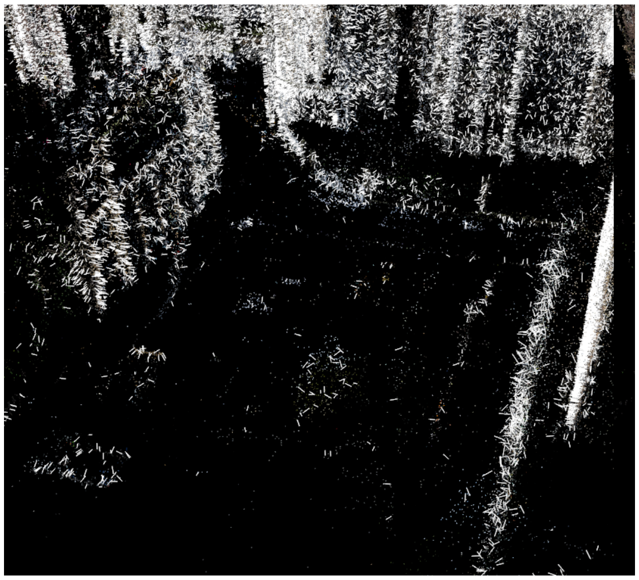


Figure 2: Point normals

2.3. Segmenting the point cloud

With the knowledge of the ground plane (up-vector) we can calculate the probability for a given point to be on a vertical plane, which makes it a candidate for being part of a facade. Projecting these points onto a ground plane should give us a 2D footprint of the building, similar to what is manually drawn in OSM. This probability P_i for each point i is estimated as $\alpha\ 1 - Dot(point\_normal_i, up\_vector)$, where $Dot(.)$ indicates vector dot product. The point i is assumed to be belonging to a facade if $P_i > threshold$. Fig 3 shows the input point cloud (on the left) and the one obtained after segmenting out points belonging to facades (on the right).

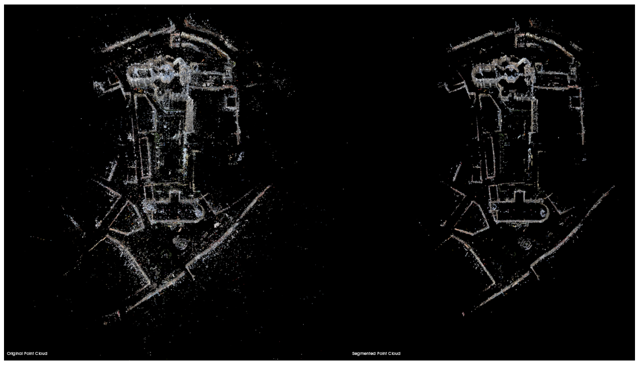


Figure 3: Segmented Point Cloud

The points of the segmented cloud (on bottom right) are now down-projected onto the ground plane to get a 2D outline (on top) indicating the building footprint as shown in Fig 4

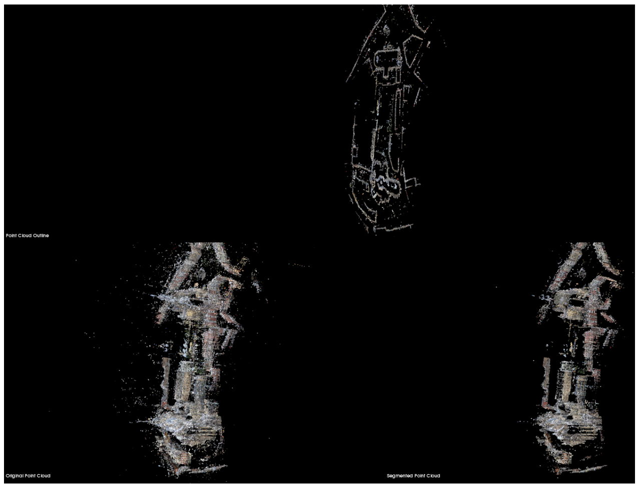


Figure 4: Building footprint obtained from point cloud

3. Obtaining the ground truth

In order to obtain the ground truth building outlines, we used OpenStreetMaps[1], a collaborative mapping effort

[1]http://www.openstreetmap.org/

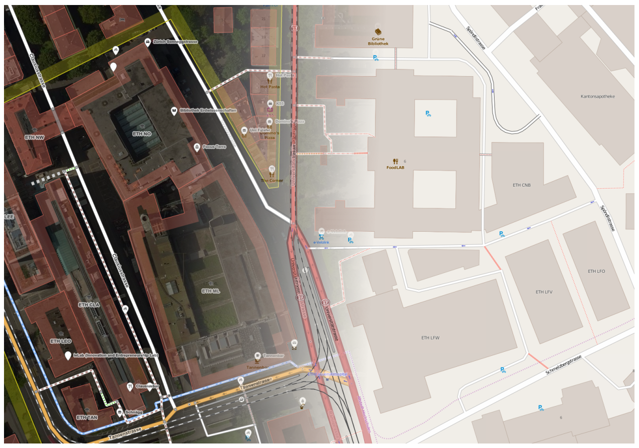


Figure 5: OSM overlaid with editor interface

that takes place globally. Volunteers are mapping their surroundings and upload it to a central database, where all changes to the map are stored. Of concern for us is only one mapped entity, the building. All points in OSM are stored as nodes. The entity consists of one or more node (in the case of a building it's multiple nodes). By parsing the XML response and mapping all node values to the corresponding building entity, we can obtain all corner points of the closed polygon that describes a building outline in the real world.

In order to adjust the point cloud to the obtained ground truth, we discretized the polygons to point clouds themselves by creating points along the polygon edges in a certain distance. During the Iterative Closest Point matching these points will be matched against the point cloud and provide an error distance.

The OSM data is in the standard latitude/longitude coordinate format. This representation maps coordinates to the globe, which has a spherical shape. However, we would like to work on a 2D surface. The size of the registration is reasonably small to obtain a 2D representation that falls into the boundaries of acceptable error.

There are a number of methods on how to convert coordinates to a 2D representation, which is a topic that has been explored for a long time since large-scale mapping has taken place globally. An exhaustive overview can be found in Snyder (1987) [**?**]. The methods can be divided into two different categories:

- **Conformal** This category of mappings preserves angles as observed in the real world (ie. the local angles are preserved).
- **Equiareal** Retains equal area
- **Equidistant** Preserving distance
- **Azimuthal** Preserving Direction
- **Shortest Route** Shortest route is observable

Since the sphere cannot be flattened out without distortion (ie. the sphere is a non-developable surface) not all of the above properties can hold for a projection at the same time.

For our application of registering a 2D point cloud on the map, we chose to use the Universal Transverse Mercator (UTM) projection, which preserves the angles and shapes, a property we deemed useful for our application.

Contrary to other map projections, UTM is not a single projection but rather divides the globe into 60 different zones.

4. Registering the Point Cloud

After obtaining the ground truth and the 2D projection of the 3D cloud we are now able to register the point cloud on the OSM reference.

Therefore we use "Iterative Closest Point" (ICP) matching algorithm. As the name suggest, ICP is an iterative algorithm to minimize the distance between two (or more) point cloud measurements. ICP is a widely used algorithm for example in robotics (Simultaneous Localization and Mapping [SLAM]).

The input to the ICP algorithm in general is two pointclouds, one called the reference and the other source. The source should be aligned to the reference cloud. The output of the algorithm is a transformation matrix in 2 or 3 dimensions. Furthermore, the algorithm needs to be supplied with a stop condition, for example a maximum number of iterations or a minimal size of change between iterations, which indicates that the algorithm has converged to a minima.

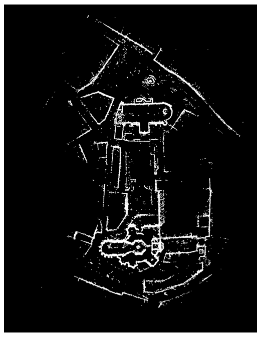
Several important drawbacks of ICP are that it usually only provides rigid transformations (i.e. scale and shear factors are not affected). Allowing infinite scale the ICP solution would scale down to only one point with a distance of zero, as all points would be incident with that one. Another problem is that ICP, without good initialisation, tends to convertge to a local minima. Therefore it is paramount to have an initial rotation and translation that matches the reference somewhat.

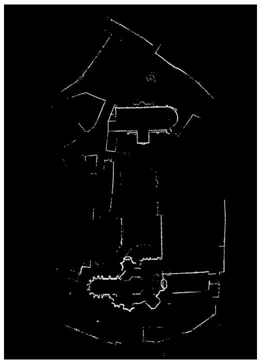
For our implementation, the existing `libpointmatcher`[**?**] library was utilized to provide a framework for customizable ICP algorithms.

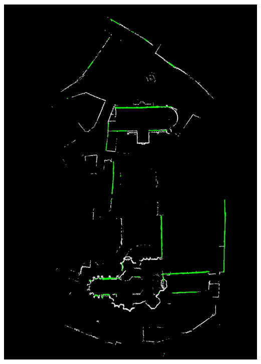
Both the map data as well as the point cloud is normalized and scaled to an arbitray scale, so that the leftmost point is coincident with the (0, 0) coordinate.

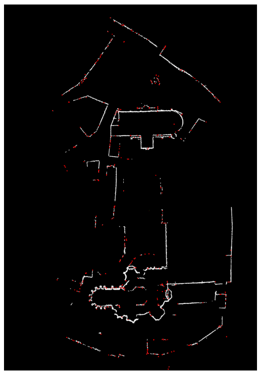
4.1. ICP process

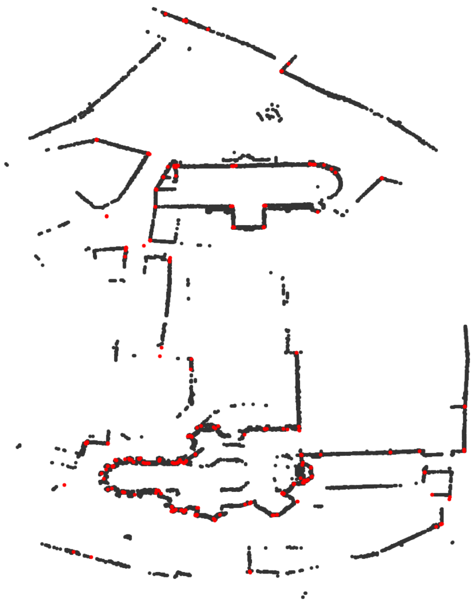
As implemented, the ICP implementation uses a KD-Tree with a variable number of neighbors it considers as potential neighbors. The nearest neigbor is accepted as neigh-


(a) Original input


(b) Output after filter


(a) Hough corner detector


(b) Harris corner detector


(c) RANSAC based corner detection

bor and a transformation is searched that decreases the distance (error) between all neighbors.

Generally, there are two different popular ICP variants: Point-to-Point and Point-to-Plane, with the latter usually performing better. However, since we are operating in the 2D space, there are no planes except for the ground plane to be found. Thus we decided to use the more approachable point-to-point algorithm.

One ICP iteration consists of 4 steps:

- We search a KD-Tree structure for the nearest neighbors and weight or reject them depending on the distance
- The error is calculated
- Using the error, the singular value decomposition (SVD) is calculated
- The rotation matrix is $\mathbf{R} = \mathbf{U} * \mathbf{V}^\intercal$
  The translation vector is $\mathbf{T} = \mathbf{A} - \mathbf{R} * \mathbf{B}$
- Repeat process until either only a very small change between iterations is observed or the error is sufficiently small

4.2. Filtering data

To further reduce the amount of noise, we implemented a filtering chain, consisting of

- Any given point needs at least 50 neighbors in a radius of 2 meters to be considered a valid measurement
- Statistical outliers are removed with a mean of 8 and a standard deviation of 1. This further removes outliers
- Only 1/8 of all remaining points are randomly chosen to reduce the computational load

4.3. Initial alignment

As stated before, the initial alignment of the two point-clouds is crucial for successful ICP matching. In order to find a good initial alignment, we try to find corner points in the measured data.

There exist a number of corner detectors that are well researched. Some of the most notable ones, the Hough-transform [?] and the Harris-detector [?] were tried, but didn't deliver sufficient results. The properties of our data are also not extremely well suited for these algorithms which were primarily developed for images.

We developed an approach based on the Random Sample Consensus algorithm (RANSAC) [?]. RANSAC is able to detect lines in a robust way. Our process looks like this:

- Choose random point from point cloud
- Select nearest 100 neighbors from kd-tree and insert into sub point cloud

- Find best line with RANSAC
- Remove all inliers from sub point cloud
- If more than threshold points remaining, find second best line by running RANSAC again
- Check angle between both lines (if angle to small or wide, reject)[2]
- If line is found, calculate intersection point
- Reject if intersection point is not close to points in sub point cloud

After a certain, arbitray numbers of corners is found, we use the map of corners to match the OpenStreetMap data. Therefore we randomly select 3 points from the corner map, and calculate a corresponding transformation to three points on the OSM data. This makes it possible to calculate an error and improve or reject several initial transformations.

We then choose the best transformation and use it as intial transformation for the ICP.

[2]Rejecting intersection points based on angle is possible for us since we are primarily interested in corners that meet at about 90 degree. This is due to the fact that we are working with city scapes.