Efficient In-Memory Point Cloud Query Processing

Balthasar Teuscher, Oliver Geißendörfer, Xuanshu Luo, Hao Li, Katharina Anders, Christoph Holst, and Martin Werner

Technical University of Munich, School of Engineering and Design, Department Aerospace and Geodesy
balthasar.teuscher@tum.de, o.geissendoerfer@tum.de, xuanshu.luo@tum.de, hao_bgd.li@tum.de, k.anders@tum.de, christoph.holst@tum.de, martin.werner@tum.de

Abstract. Point cloud data acquired via laser scanning or stereo matching of photogrammetry imagery has become an emerging and vital data source in an increasing research and application field. However, point cloud processing can be highly challenging due to an ever-increasing amount of points and the demand for handling the data in near real-time. In this paper, we propose an efficient in-memory point cloud processing solution and implementation demonstrating that the inherent technical identity of the memory location of a point (e.g., a memory pointer) is both sufficient and elegant to avoid gridding as long as the point cloud fits into the main memory of the computing system. We evaluate the performance and scalability of the system on three benchmark point cloud datasets (e.g., ETH 3D Point Cloud Dataset, Oakland 3D Point Cloud Dataset, and Kijkduin 4D Point Cloud Dataset) w.r.t different point cloud query patterns like k nearest neighbors, eigenvalue-based geometric feature extraction, and spatio-temporal filtering. Preliminary experiments show very promising results in facilitating faster and more efficient point cloud processing in many potential aspects. We hope the insights shared in the paper will substantially impact broader point cloud processing research as the approach helps to avoid memory amplifications.

Keywords: 3D Point Cloud, In-Memory Processing, Spatio-temporal, Nearest Neighbor

1 Introduction

Point clouds significantly differ from other geospatial data in terms of their computational nature rendering efficient processing of point clouds in traditional geospatial infrastructures such as relational database management systems (RDBMS) or distributed key-value stores complex [9, 20, 23]. The reason is rooted in the imbalance of data and metadata when a huge number of comparably small objects is to be organized. In relational databases, for example, a natural mapping would let each point occupy a row in a table inducing at

least a primary key on top of the points. Otherwise, relational mapping of attributes would not be easily possible. For big data systems, key-value patterns dominate the organization of processing in line with MapReduce [8, 19] as well as the theory of (almost) infinitely scaling systems. In both cases, managing the identities (primary keys in databases, keys in a key-value system) is similarly complex to manage the data itself (e.g., points represented as tuples of floating point values).

One proposal often seen in this context is spatial gridding or voxelization. If we would go ahead and cut the point cloud into grid cells, we can organize the data – both in distributed key-value stores or in relational data management systems – in terms of these blocks or 3D tiles. However, this implies that queries are first defined only up to these artificial data boundaries and refinement steps are needed if, for example, a query's data demand extends beyond the current 3D tile.

Furthermore, this approach has limitations when it comes to varying point densities, which is, however, a very common aspect for terrestrial point clouds where density degrades with the distance from the sensor to the object and objects (after, for example, ground removal) are scattered in space. For focal point cloud analysis (e.g., point cloud analysis where a bound on the distance of relevant other points is given), this has been done by extending the grid to a grid overlapping neighbors [4].

The challenge of three-dimensional point cloud data and their volumes is amplified when the time dimension is added through repeated point cloud acquisitions. Neighborhood queries are then extended into the temporal domain [1, 18], for example, for spatiotemporal point cloud filtering.

In this paper, we propose and demonstrate that the inherent technical identity of the memory location of a point (e.g., a memory pointer) is both sufficient and elegant to avoid gridding as long as the point cloud fits into the main memory of the computing system. With modern servers with commonly more than 1TB of main memory, this requirement is, however, not a severe limitation for many use cases. We demonstrate an open-source implementation of complex point cloud queries following this approach and give performance insights on selected queries. We test the feasibility and performance of these queries on multiple benchmark 3D point cloud datasets and share insights into a future deployment for high-performance computing (HPC) based point cloud analysis.

The remainder of the paper is structured as follows: The next Section 2 reviews related work on point cloud data management and point cloud processing. In Section 3, we derive an architecture for scalable point cloud data management in the Python environment. In Section 4, we classify various data modalities and common query patterns in order to derive an architecture for a point cloud processing engine. In Section 5, we evaluate scalability and performance on a variety of datasets. Section 6 comments on future potential directions and Section 7 concludes the paper.

2 Related Work

Point cloud data (containing x,y,z information) can be acquired via laser scanning and stereo matching of photogrammetry imagery, and has become an emerging and vital data in increasing aspects of real-world applications, ranging from forestry investigation [9, 24], 3D city modeling [15, 35], precision agriculture [32], to autonomous drones/cars [36]. Especially, point cloud acquisition using the Light Detection And Ranging (LiDAR) method has been investigated in diverse levels of scale and accuracy, from satellite-based (e.g., the Geoscience Laser Altimeter System (GLAS) on ICESat [38]), regional-scale Airborne Laser Scanning (ALS), to fine-grained Terrestrial Laser Scanning (TLS). The emerging availability of such sensors and data acquisition techniques resulted in a continuing growth in the amount of point cloud data, which poses great challenges for efficient data storing, processing, and management to the conventional spatial computing algorithms designed for 3D point data.

Point cloud processing is a highly challenging task due to an increasing point rate of available LiDAR scanners and the demand for handling the data in near real-time. The processing mainly involves registration to existing point clouds with the ICP [2], RANSAC [28] algorithms and segmentation tasks [26, 27]. Both purposes require features that are computed on images [6] and point clouds [5] directly. Additionally, tasks are often solved with Deep Learning approaches [7, 13, 37]. Specifically, geometric features based on the mean, variance, number of points, density, and normal vectors [16] are used to describe local properties. Eigen-based features [3] have gained interest for multiple years since they offer a higher-level specification of the surroundings. All features computed from the local neighborhood increase the information content of the respective environment. Therefore, using robust and unique features are utilized for an unambiguous matching [30], which is needed to identify corresponding regions.

Computing meaningful features in local neighborhoods is often a challenging piece of work. Structuring a point cloud with an octree [10, 17] is a common approach used to describe point surroundings. Therein, respective tree leaf nodes summarise the information content of all points within its node boundaries of a pre-defined size. However, points lying close to the border of adjacent leaf nodes will not contribute to a high-level feature at the same time. An alternative is shown in [14] with building a graph structure to compute a triangulation to find line and edge patterns. Creating a graph enables a faster and more flexible search in a point neighborhood of an unsorted point cloud. Furthermore, using graphs is applicable in point cloud deep learning approaches [11, 33] as well.

In summary, quite challenging algorithms in terms of very large loops and many neighborhood retrievals are commonly needed to process point clouds.

3 Architecture

The point cloud query engine is designed following a few modern programming paradigms especially related to big geospatial data management and differs in

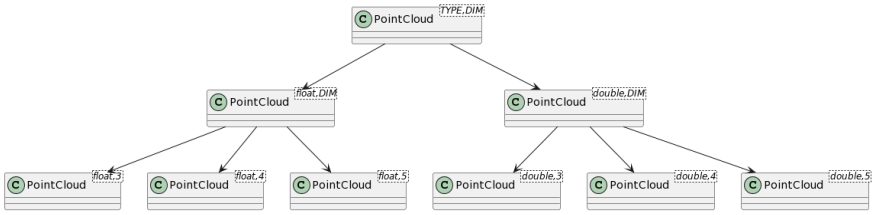


Fig. 1: Generic Class Architecture

its architecture from other approaches quite a bit. In this section, we highlight architectural choices, how they affect the performance of the system and give hints on future extensions and the path forward for this system.

3.1 Computational Infrastructure

The first and most important decision is to choose a generic programming platform that provides us with a good blend of performance, portability, usability, existing peer-reviewed libraries for geometry and linear algebra, flexibility, simple parallel programming features, and excellent Python interfacing. With these attributes, the choice for the core language has been made to be modern C++ as it provides us with type-generic programming, excellent OGC geometry implementations using boost::geometry, fast and powerful linear algebra using Eigen3, and annotation-based parallel programming in a shared-memory-parallel setting (SMP) based on OpenMP.

While not yet implemented, the choice of standard C++ containers throughout the implementation facilitates the use of distributed versions of these containers as well, such as provided by the DASH library for partitioned global address space model of computing, where a global address space across a supercomputer is simulated [12]. In this way, the library might work across large supercomputers coordinated using the Message Passing Interface (MPI).

All functionality has been implemented as a single C++ header file relating only to the boost and Eigen3 libraries. The implementation itself is almost fully generic as depicted in Figure 1.

The main pointcloud class is implemented for all possible numeric types and all dimensions in the same code base. Template metaprogramming is used to implement functions for different dimensionalities, while the system is flexible enough to partially specialize certain dimensions. For example, a spatiotemporal algorithm can only be run on point clouds with at least four dimensions. Of course, such functions cannot be implemented as member functions of the generic PointCloud class, as any attempt to access the fourth coordinate in a 3D instantiation of a pointcloud will lead to a compilation error. But why do we go this way, if it is limited in this way? The reason is that in contrast to most other libraries, the type and dimension of the point cloud are constant at compile time and all loops or branches based on dimension (e.g., a loop going over the x,y,z

coordinate in a 3D point cloud, but over x,y,z,t in 4D) can be easily unrolled by the compiler for unprecedented performance gains.

As said, we still want to implement special functions for less generic point clouds. In order to do so, we use free functions that get references to the point cloud. As we make sure that we request these functions only in a valid context, they will happily compile and at the same time be able to apply dimension and type-based optimizations otherwise not possible in this stage.

3.2 Algorithmic Infrastructure for Geometry and Spatial Indexing

Instead of reinventing the wheel or using non-generic implementations, we rely on the most modern family of peer-reviewed libraries for scientific computation in C++. Boost Geometry provides us with a complete and reasonably correct implementation of almost all OGC simple feature types and operations. It is generic in dimension and coordinate type in itself and we can forward the type-specifications with which the point cloud was instantiated with ease.

In addition, the indexing library of boost geometry provides us with a generic implementation of an R-tree indexing tree and associated query algorithms which we instantiate with a default R^* balancing algorithm. This is a good approach for dynamic insertion of point especially in the case when many nearest neighbor queries are going to be performed. We initialize the tree using an STR bulk load algorithm, but in many situations, we will want to stream batches of points into the index to avoid the need for large contiguous areas of main memory for holding points while building the index. In this case, the first batch is used to STR bulk load a tree while subsequent points are inserted using the R^* approach of reinserting points on node splits.

This R-tree implementation provides a query interface with an output iterator (e.g., the query is actually a typical tree scan and emits one indexed value at a time) which we adapt to a C++ function call. The query predicate is built as a Boolean combination of contains, covered_by, covers, disjoint, intersects, overlaps, within and two special predicates. The first special predicate is the "satisfies" predicate which can be used to call a custom Boolean predicate during tree traversal that decides on whether points qualify for the result set in a rather arbitrary way and a "nearest" predicate which ensures that only the k nearest neighbors are retrieved. Note, however, that for performance reasons the output of the k nearest neighbors is not sorted by distance as in distance browsing - an aspect to keep in mind when using the system. It is worth noting that this setup has a huge advantage stemming from functional programming capabilities of modern C++ : when we implement the function that takes one result set element after another as a lambda function, we can capture variables from the calling scope. This can be used to, for example, aggregate the mean of all results without ever materializing the result set in memory.

3.3 Algorithmic Infrastructure for Linear Algebra

One of the query patterns we investigated while setting up this system is a quite classical one: from a given definition of the surroundings of a point, one builds up a $N \times 3$ matrix M, extracts the corresponding three eigenvalues $\lambda_0, \lambda_1, \lambda_2$ of the 3×3 self-adjoint matrix $M^t \cdot M$, normalizes them by their sum, orders them and calculates features including, but not limited to:

$$\begin{aligned}
\text{linearity} &= (\lambda_0 - \lambda_1)/\lambda_0 \\
\text{planarity} &= (\lambda_1 - \lambda_2)/\lambda_0 \\
\text{scattering} &= \lambda_2/\lambda_0 \\
\text{omnivariance} &= \sqrt[3]{\lambda_0 \lambda_1 \lambda_2} \\
\text{anisotropy} &= (\lambda_0 - \lambda_2)/\lambda_0 \\
\text{eigen-entropy} &= -\lambda_0 \log_2(\lambda_0) - \lambda_1 \log_2(\lambda_1) - \lambda_2 \log_2(\lambda_2) \\
\text{trace by largest} &= (\lambda_0 + \lambda_1 + \lambda_2)/\lambda_0 \\
\text{change of curvature} &= \lambda_2/(\lambda_0 + \lambda_1 + \lambda_2)
\end{aligned}$$

In our implementation, the Eigen3 library is used for eigenvalue extraction by QR decomposition, which could be further optimized by using a direct equation solution (with some loss in numerical precision) for compile-time constant-sized matrices. However, we skip such microoptimizations as the system is - at the moment - memory bound and a wall-clock performance improvement would not be possible.

3.4 Python Bindings

As Python is one of the most prominent glue languages, our system relies on pybind11 to efficiently (i.e., in most cases without copying memory or marshalling representations) communicate and share data between our library, numpy, and the user. However, as Python does not provide generic programming support, we instantiate the leaf nodes in Figure 1 as individual Python classes. For each of these classes, we first bind specific API calls related to the dimension and type, followed by binding a generic API that all pointcloud classes share. This finalizes the solution to the problem that some operations can only be defined for certain dimensions of point clouds, as we will require those functions only from the function implementing the 4D Python binding.

3.5 Data Handling

All data in the library is organized as named columns, where the current version supports exactly one point cloud, and a variable number of named 1D columns representing the results of computations.

At this moment, the point information is available twice: once in an array, accessible via index, and once in the R-tree index as the leaf nodes contain

the actual data. Fortunately, the array can be dropped if wanted, even more, a pattern of loading a dataset as a sequence of smaller numpy arrays (maybe a few hundred megabytes of data) always clearing the point cloud data structure afterward is possible. This has the effect that no large contiguous memory areas are occupied and the final memory consumption will be the cost of storing the points in the R-tree plus the size of the small numpy array that is used to provide the batches to the C++ implementation. A nice consequence is that the point cloud that you are going to use for queries can be a different one (e.g., a 3D grid if you are interested in a voxel-like computation). Most queries then loop over the given cloud, which can be either the original one or another one and compute features or other computations for each point in this cloud.

4 Data Modalities, Query Patterns and the API

Point clouds have different properties depending on how they are acquired and preprocessed. As we are providing a general implementation framework that is suitable for all point clouds, we select representative point cloud benchmark datasets of different natures for evaluation.

Photogrammetric point clouds are typically very dense 3D point clouds acquired from cameras in a 3D reconstruction setting where multiple images from different viewpoints are combined to a 3D point cloud structure.

Laserscanning point clouds show a comparably higher non-uniformity as many scanners are acquiring images steering the laser ray using a specific pattern. In addition, the spatial or angular resolution of laser scanners is often lower while the range and precision outperform photogrammetric methods.

In many cases, the sensor itself is mounted to a mobile platform and thus generates data along the trajectory of the mobile sensor platform. Methods of the family of Simultaneous Localization and Mapping (SLAM) use this mobile sensor information to jointly estimate the trajectory of the mobile sensor platform and a pointcloud representing the environment. While these approaches help to mitigate single scan weaknesses such as limited resolution and scan patterns, they introduce their own limitations due to strongly depending on the estimated pose of the sensor which is often subject to error accumulation through time, especially from integrating the movement using inertial measurement units.

Lately, neural rendering and especially Neural Radiance Fields (NeRFs) have been invented to generate a dense voxelized model of the environment from a limited number of observations. While the output is actually some average radiance observed for rays crossing a 3D volume, it can naturally be considered a point cloud by representing the volume with its center point. In a similar way, PointNeRF [34] directly models neural radiance fields with point clouds as opposed to vectors and we expect this exciting area of research to provide another category of point clouds in the future as occlusions can be modelled much better as opposed to scan-based methods.

Further, point clouds provide a way to represent geometry in a fully local way such that in some settings people sample point clouds from geometric models,

including CAD and GIS datasets. Such sampling can reduce the complexity and generalize the sampled objects' geometric representations. For example, a random point on the surface of a mesh is often required in shape similarity computation [22].

The general sampling nature of point cloud data and the individual samples being unbounded implies relating and aggregating the samples in order to draw meaningful information. The attributes of a single sample are almost never of immediate interest, but rather the neighborhood, which constitutes a segmentation of objects of interest. Hence range and neighborhood queries are at the core of point cloud analysis. Even more, given the importance of points contributing to the description of a neighborhood decreases with its distance to it [29].

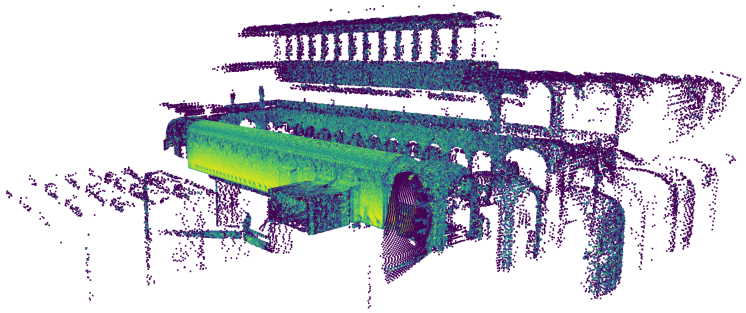


Fig. 2: ETH 3D Point Cloud Dataset.

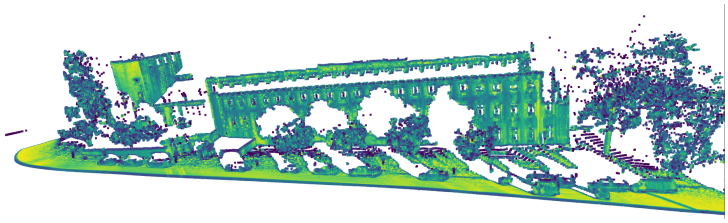


Fig. 3: Oakland 3D Point Cloud Dataset.

In this paper, we consider three 3D point cloud benchmark datasets (as shown in Figure 2, 3, and 4) of different sizes to evaluate the performance of our in-memory point cloud processing library w.r.t the time and efficiency of point cloud query and feature extraction:

ETH 3D Point Cloud Dataset: The first dataset was collected in the main building (Hauptgebaude) of ETH Zurich [25], where the laser scanner moved mainly along a straight line in a long hallway. This hallway is located on the

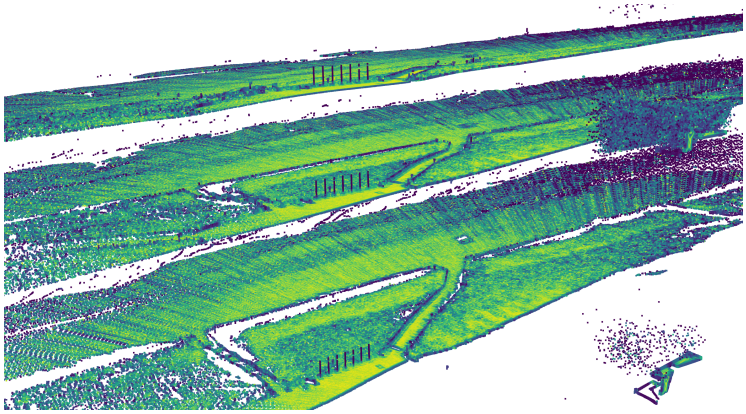


Fig. 4: Kijkduin 4D Point Cloud Dataset with first three timestamps shown from bottom to top.

second floor and surrounds the central place, which is then open over several floors. Herein, the hallway has a curved ceiling and a wall on one side. On the other side, there are pillars, aches and ramps. Those elements are the main interest of this dataset. ETH 3D point cloud datasets consist of in total 6,810,063 points.

Oakland 3D Point Cloud Dataset: This classic dataset was collected using Navlab11 equipped with side-looking SICK LMS laser scanners and used in push-broom [21]. The dataset was collected around the CMU campus in Oakland, Pittsburgh, PA, and contains labeled 3D point cloud derived from LiDAR data collected using a moving platform in an urban environment. Oakland 3D point cloud dataset contains 1,614,625 points.

Kijkduin 4D Point Cloud Dataset: The third dataset was recorded by a laser scanner observing one kilometer of coast at hourly intervals for about six months, in the Kijkduin dune system in the Netherlands [31]. This dataset includes more than 4,082 consecutive topographic scans of a varying size in the lower millions of points each, therefore, it can be regarded as a unique 4D spatiotemporal point cloud dataset in understanding the morphological behavior of the beach-dune system at hourly to monthly time scales. The total number of points in the dataset across all individual hourly scans adds up to about 6.6 billion spatiotemporal 4D points.

5 Evaluation

The following section gives hints on the performance and scalability of the system in typical scenarios. Specifically, we consider three types of point cloud queries: (1) the extraction of k nearest neighbors; (2) the extraction of eigen features; (3) spatio-temporal filtering of 4D point clouds. Moreover, we test the performance of these queries for smaller datasets on two different computing plat-

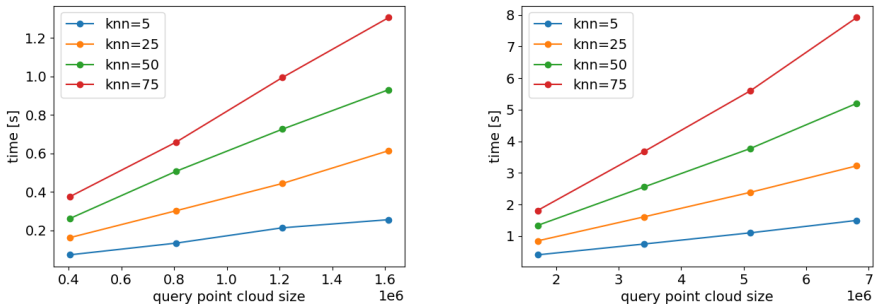
forms, namely a business laptop (with an Intel(R) Core(TM) i7-8565U CPU, 16.0 GB DDR4 2400MHz memory and a 512GB PCIe 3.0 M.2 SSD) and a gaming PC workstation (with an Intel(R) Core(TM) i9-10900K CPU, 32.0 GB DDR4 2933MHz memory and 1TB PCIe 3.0 M.2 SSD). The big Kijkduin cloud is processed on a medium performance AMD-based server (AMD EPYC 7702P 64-Core Processor, 1 TB main memory, 500 TB disk array).

5.1 Extraction of K Nearest Neighbors

As a first query, we consider the raw extraction of k nearest neighbors of any point and perform simple aggregation in terms of finding the maximum distance from the query point on two benchmark datasets, namely the Oakland 3D dataset (with around 1.6 million points) and ETH 3D dataset (with around 6.4 million points). In terms of pseudo-code, the considered algorithm for a parameter k is given as follows:

```
1 for p in points:
2     neighbors = knn(p, k)
3     d = max_dist(neighbors,p)
```

Figure 5 gives the performance of this query for varying datasets and values for parameter k. As expected, this query has an almost linear scaling behavior with respect to the size of the point cloud on a single machine, and this is observed by the straight lines in the figure. The increasing steepness of the lines is implied by the fact that not only more points are given, but also each neighborhood selection - despite the neighborhoods having the exact same size - traverses a deeper tree.

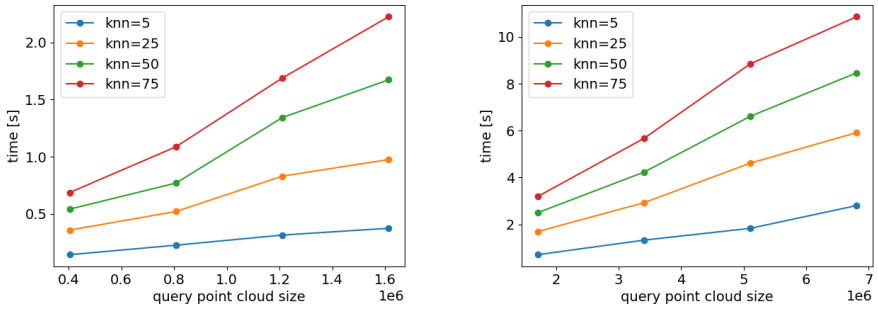


(a) Oakland 3D Dataset (1.6 million points) (b) ETH 3D Dataset (6.7 million points)

Fig. 5: Performance of simple kNN Queries over the whole dataset.

5.2 Extraction of Geometric Features

Geometric features are computed from a neighborhood. We define the following neighborhoods as they incur different query patterns: (1) kNN: the neighborhood consists of the k nearest neighbors; (2) cube: the neighborhood consists of a n-dimensional cube which might contain infinite coordinates, for example, a 2D box in 3D would be modeled as $(x_{\min}, y_{\min}, -\infty) \rightarrow (x_{\max}, y_{\max}, \infty)$; (3) sphere: all points that have a 2D or nD-Euclidean distance smaller than a given radius; (4) cylinder: all points within a cylinder.



(a) Oakland3D 3D Dataset (1.6 million points) (b) ETH 3D Dataset (6.7 million points)

Fig. 6: Performance of calculating 10 different eigenvalue-based features over the whole dataset.

However, due to the nature of the R-tree, the query processing of all of these neighborhoods is essentially the same: first, we take the minimum bounding rectangle (MBR) of the query object (box, cylinder, sphere, etc.) and determine all points within this box with a tree traversal. During query processing, we evaluate the more specific geometry through additional "satisfies" constraints. All valid points passing all geometry tests are then collected as the relevant neighborhood. This neighborhood is now shifted to the origin either by computing the centroid of the neighborhood or by directly using the query point that was used to create the neighborhood. All results are combined into an $N \times 3$ matrix M. A self-adjoint 3x3 matrix is generated using from $M^t \cdot M$. As the eigenvalues of selfadjoint matrices are always real numbers, we can extract them, order them, and normalize them to sum up to one. The extraction is currently performed with a QR composition leading to good numerical stability, but faster algorithms would be available for 3x3 self-adjoint matrices. However, as we will see, the algorithms are not compute-bound such that this optimization would not pay off in terms of wall-clock time (but may save energy or provide more compute time in different scenarios).

These eigenvalues are then used together with the various derived geometry features from Section 3.3 to compute ten features for each point of the point cloud. As the performance evaluation in Figure 6 shows, computing these features only adds a small overhead compared to bare kNN queries in Figure 5.

Table 1 shows the detailed comparison of different query times of our proposed in-memory point cloud processing library on two different computing platforms in order to better understand the efficiency and scalability of the system. Herein, we compare the processing time (in second) of mainly four steps: (1) the initialization, which refer to a data reading operation; (2) the indexing, which create the R-tree for the whole point clouds; (3) the extraction of k nearest neighbors with k=25; (4) the extraction of ten eigen features based on k nearest neighbor results. One can easily identify two key findings from this comparison: first, a general trend of fast and efficient in-memory point cloud processing can be observed across different datasets and computing platforms; second, though the extraction of eigen features is still the most time-consuming operations, this becomes much more affordable and with even incremental addition times in a workstation. Note that the laptop was running a Microsoft Windows operating system to model a typical geoscientists mobile computing device while the workstation was running Linux.

	Oakland 3D (1.6M pts.)		**ETH 3D (6.2M pts.)**	
Query Time(s)	Laptop	Workstation	Laptop	Workstation
Initialization	0.094	0.041	0.525	0.219
Indexing	0.483	0.226	3.327	1.174
kNN radius	3.137	0.646	30.458	3.511
Eigenfeatures	10.296	1.093	84.769	5.882

Table 1: Performance of point clouds queries with two benchmark datasets on different computing platforms (i.e., a business laptop running Microsoft Windows and a gaming PC running Linux).

5.3 Spatio-temporal Filtering

The third query pattern is about spatio-temporal filtering. Especially in long-term observations, noise and small perturbations to the observed geometry shall be differentiated from relevant signals representing true change of the geometry.

We implement a special query designed for a spatio-temporal point clouds that does compute for each point of a query point cloud the average Z coordinate of a space-time cylinder with a given spatial radius, unbounded spatial height, and a configurable time window. In this way, a very massive point cloud is analyzed locally in space and time using a query grid of points of interest for which this value shall be computed. This query is also used to highlight the simplicity of extending our framework by experts from the field not firm in

modern C++. Therefore, we discuss the function doing the computation right here:

```
1  namespace bg = boost::geometry;
2  template<typename pointcloud>
3  void boxfilter_4d(pointcloud &pcl, std::string prefix,
4                    double radius, double time_radius) {
5      pcl.named_attributes[prefix+"_boxfilter4d"] = std::vector<double>(pcl.cloud.size
           ↪ ());
6      auto &a = pcl.named_attributes[prefix + "_boxfilter4d"];
7      auto & cloud = pcl.cloud;
8      #pragma omp parallel for
9      for (size_t i=0; i< cloud.size(); i++) {
10         typename pointcloud::point_type minPoint, maxPoint;
11         bg::set<0>(minPoint, bg::get<0> (cloud[i]) - radius);
12         bg::set<1>(minPoint, bg::get<1> (cloud[i]) - radius);
13         bg::set<2>(minPoint, -std::numeric_limits<double>::infinity());
14         bg::set<3>(minPoint, bg::get<3> (cloud[i]) - time_radius);
15         bg::set<0>(maxPoint, bg::get<0> (cloud[i]) + radius);
16         bg::set<1>(maxPoint, bg::get<1> (cloud[i]) + radius);
17         bg::set<2>(maxPoint, std::numeric_limits<double>::infinity());
18         bg::set<3>(maxPoint, bg::get<3> (cloud[i]) + time_radius);
19
20         typename pointcloud::box_type query_box(minPoint,maxPoint);
21         double sum=0, N=0;
22         pcl.rt.query(bgi::intersects(query_box), boost::make_function_output_iterator
               ↪ ([&](typename pointcloud::value_type const& v) {
23             N++;
24             sum += bg::get<2>(v.first.min_corner()); // z coordinate
25         }));
26         double mean = sum/N;
27         a[i] = mean;
28     }
29 }
```

This listing shows how easy it can be for a domain scientist to implement their own computational kernels over the point cloud. The infrastructure hides all Python interfacing, and the generic nature allows the implementation of a function that can be immediately used with, for example, `float` or `double` values for coordinates. References are used to make shortcuts to the cloud itself and to the output attribute vector (lines 5-7). Then, parallel processing is required with a parallel for loop based on OpenMP. The body itself prepares a query box with both finite and infinite coordinates (lines 11-21). Then, accumulators (in each thread of parallel execution) are initialized as double variables N and sum (line 22). Then, we increment N (line 24) and accumulate the Z coordinate (the second coordinate) out of the R-tree box representing this point in line 25. Note, how handy the capture of lambdas has become: within a foreign query infrastructure given by the R-tree, we can write to local variables `N` and `sum`. As a last step in each loop, we store the computed mean value of the attributes

at this location. Hence, the coordinate point `cloud[i]` and the named attribute contain our results. Note that the estimation of the mean in this way is not optimal from a numerical perspective and can be replaced with an incremental form, but we aim for simplicity in this example.

Running the prepared query over the whole point cloud is not reasonable as spatiotemporal filtering is especially useful for standardizing geometry and reducing the amount of data. Therefore, we exploit the following design facts: (1) point cloud data is implicitly inside the R-tree including row number information to the original dataset when needed and (2) the point cloud can be overwritten with a different cloud.

For this approach, therefore, we create a spatiotemporal grid with 2D coordinates and a time coordinate and estimate the Z value of a 4D point cloud from the filtering result.

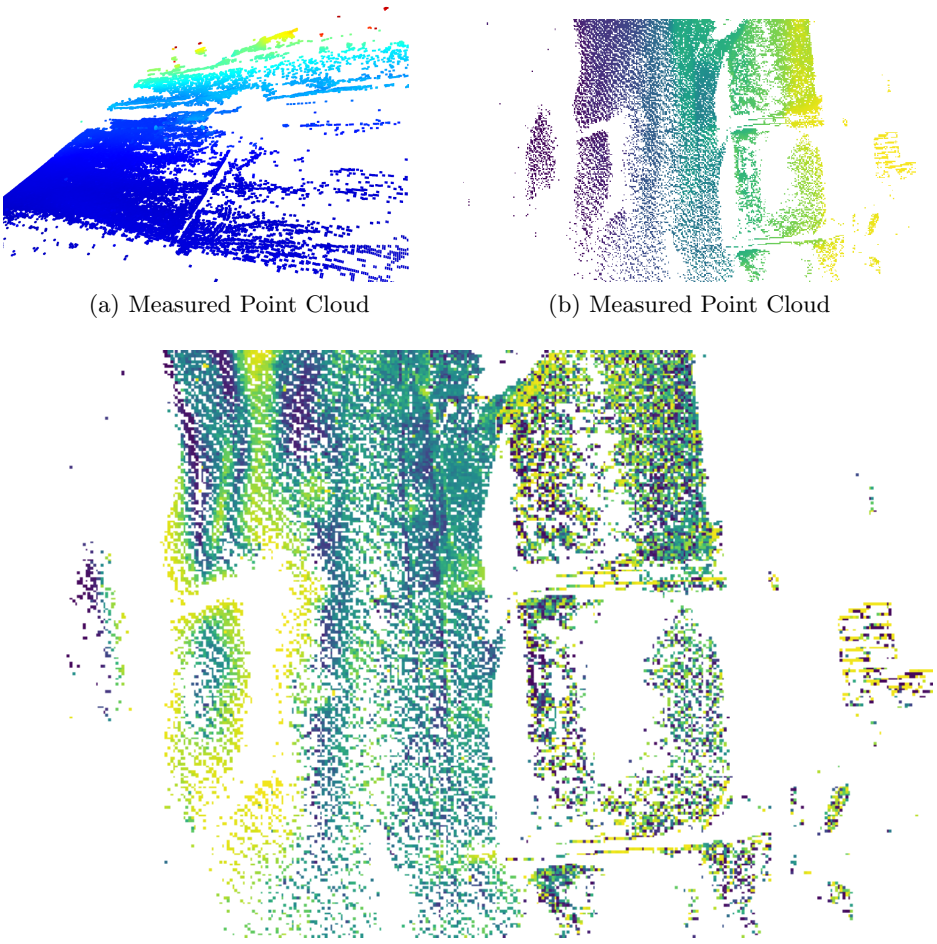
The query grid covers a bit more than the spatial region of the point cloud generated by a permanent laser scanner (PLS) station at the Kijkduin beach system in the Netherlands. For each timestep, this query point cloud contains 6.6 million points such that a query with 10 timesteps evaluates 66 million neighborhoods. Note that this query grid has been designed to be larger than the original point cloud to see the behaviour under missing data.

The original dataset [31] contains more than 4,000 timesteps, often spaced in one hour, taken from the rooftop of a nearby hotel. The total number of valid points sums up to 6.038 billion 4D points. In order to cope with such a massive point cloud in an in-memory setting, we used a moderate server that we expect to be available to most scientists working with permanent laser scanning data. The Kijkduin data has been put into an HDF5 container with a flat storage layout to speed up reading through memory mapping. The file contains 5 columns of data each with 6 billion double values summing up to a raw size of 121 GB.

With a query grid with a 50cm point distance and 10 selected timestamps, we run a filter with a 50cm radius and 1 week of temporal range as well and average the Z coordinate and a second query with a 2m spatial range and, again, one week of temporal range. Due to differences in the spatial extent of the different query times and our choice of a larger grid, some of these neighborhoods turn out to be empty generating a Not-a-Number (NaN) value implied by a zero division. We easily drop those NaN values by slicing in Python.

Parts of the results are visualized in Figure 7. The original measured point cloud is compared to the spatiotemporally filtered point cloud leading to a change map as depicted in the Figure. One can clearly see stronger (yellow) changes between a measured point cloud and the spatiotemporally filtered point cloud.

The overall performance of the query processing for this massive point cloud has been measured by wallclock time. Reading the dataset took half a minute (30s, 6 GB/s). This number is based on holding the data in a disk array with decent hardware and in a single HDF5 container with contiguous dataset layout. After reading the data, we transfer ownership from numpy to our library instantiating geometry points. This takes another 38s. Note that due to the design of the system, we could have streamed the points in smaller batches from disk



(a) Measured Point Cloud

(b) Measured Point Cloud

(c) Difference

Fig. 7: The measured point cloud and the smooth result as well as the deviation map computed from the difference of the spatiotemporally filtered cloud and the measured cloud.

to memory avoiding this duplication, but with 1 TB of main memory, it was an unneeded complication. However, when using a chunked storage layout, of course, the HDF5 data should be streamed in the chunks that have been used to organize the file on the disk. The spatial indexing now converts all points to boxes and organizes them with a sort-tile-recurse strategy. This takes 464s, about 7 minutes. Afterward, we drop the point cloud data and add our query point cloud into the library with negligible time cost. The filtering query with 50cm neighborhoods with a time extent of one week and 10 timesteps covering the whole observation duration then, was filtered to the average Z coordinate in another 945s, which is 15 minutes. Hence, the whole procedure took 1,478s or less than 25 minutes. The throughput of the approach in this case is an impressive number of nearly 45,000 neighborhoods per second.

6 Future Work

With this approach, we have shown that current hardware combined with a hardware-friendly programming environment like C++ is able to cope with the scale of modern point clouds without any programmatic handling of the big data nature. We can treat even the largest point cloud with more than 6 billion points as a simple in-memory dataset using a numpy array and work on it including spatial indexing, spatiotemporal filtering, and arbitrary user-defined filters. The system can be steered from Python in an intuitive and efficient way. However, successful big point cloud processing is not feasible in Python itself due to the need for parallelization and excessive loops when looping over all points of the data.

6.1 Interactive Analysis

Clearly, 25 minutes of work on a medium-performance server is not a problem for a point cloud acquired with months of measurements. However, we believe that speeding up the data science cycle for point cloud analysis with interactive performance of experiments can still speed up the research cycle as well.

Two aspects of future work are planned: First, to build a simple *distributed data infrastructure* on top of this paper's computational infrastructure such that cloud computing machines or compute clusters commonly available in universities can be used to further reduce the analysis time.

Second, we want to investigate a view-based computation model such that the computation is constrained by the requirements of a camera. That is, the query point cloud is actually generated by view culling of a camera and only "visible" query points are processed. In this way, we hope to interactively work across even larger point clouds like the AHN3 point cloud covering the whole of the Netherlands.

6.2 Desktop and Web Application

For many point cloud processing tasks, desktop applications are being used and are quite popular. The desktop application paradigm is important, especially for field work where an Internet connection is not available and powerful server-scale compute systems are not commonly used.

At the same time, the community starts collecting, integrating, visualizing and analyzing an ever-growing amount of spatiotemporal point cloud data. Many of these point clouds are not compatible with the assumptions made for desktop applications.

In this context we aim to extend the point cloud infrastructure proposed in this paper with both: a web-based interface for browser-based access, processing and visualization of tera- or even petascale point cloud collections and the ability to use the very same techniques, algorithms, and sketches during development and fieldwork on a local machine (with significantly smaller datasets, of course).

7 Conclusion

From a technical point of view, the proposed library is built around the Python buffer protocol which allows the communication of data structures between different Python libraries and the Python core without copying or transforming data. Thus, a point cloud starts to exist in Python, for example, as a NumPy array with three columns (3D point cloud) or even 4 columns (4D point clouds including time). Such a point cloud is communicated to our C++ library in terms of a memory pointer and some descriptive aspects for efficient slicing support. In this library, the point cloud is indexed as an R*-tree using the Sort-Tile-Recurse bulk loading algorithm. The resulting indexing tree consists of a well-balanced arrangement of possibly overlapping rectangles which is known to be extraordinarily performant for neighborhood queries such as small range queries or nearest-neighbor queries. Furthermore, we support additional query predicates of spatial nature like ST_Within, their negations, an ordering constraint, or whether the candidate points satisfy a user-implemented unary predicate.

With such an efficient access structure for neighborhoods of points (both spatial neighborhoods like within 30cm as well as topological neighborhoods like the ten nearest neighbors or temporal neighborhoods like within 100 seconds), we implement a fast singular value decomposition optimized for small environments with not more than a few hundred points. From this singular value decomposition, we can read the three real-valued eigenvalues (for 3D) and extract the structure tensor features such as linearity, planarity, scattering, omnivariance, anisotropy, eigenentropy, normalized trace, and change of curvature. The library is publicly available in Github[1].

To summarize, given the fact that there are nearly no overheads between a point cloud in numpy and processing in our library except for spatial indexing, we expect that this work will have a strong impact on point cloud processing

[1] https://github.com/tum-bgd/pointcloudqueries

research as the approach helps to avoid typical memory amplifications and scales to datasets that have been difficult to process before. Moreover, we aim to provide the 3D research community with a simple, extendable, performant, portable environment to formulate even complex computing tasks based on a mixture of OGC geometry, linear algebra, artificial intelligence libraries such as PyTorch and TensorFlow, and HPC platform for visualization, essentially without data transformation cost. This paper presents the first step into this direction.

References

1. Anders, K., Winiwarter, L., Lindenbergh, R., Williams, J.G., Vos, S.E., Höfle, B.: 4d objects-by-change: Spatiotemporal segmentation of geomorphic surface change from lidar time series. ISPRS Journal of Photogrammetry and Remote Sensing 159, 352–363 (2020), https://www.sciencedirect.com/science/article/pii/S0924271619302850
2. Besl, P.J., McKay, N.D.: Method for registration of 3-D shapes. In: Schenker, P.S. (ed.) Sensor Fusion IV: Control Paradigms and Data Structures. vol. 1611, pp. 586 – 606. International Society for Optics and Photonics, SPIE (1992), https://doi.org/10.1117/12.57955
3. Blomley, R., Weinmann, M., Leitloff, J., Jutzi, B.: Shape distribution features for point cloud analysis and a geometric histogram approach on multiple scales. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences II-3, 9–16 (2014), https://isprs-annals.copernicus.org/articles/II-3/9/2014/
4. Brenner, C.: Scalable estimation of precision maps in a mapreduce framework. In: Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. SIGSPACIAL '16, Association for Computing Machinery, New York, NY, USA (2016), https://doi.org/10.1145/2996913.2996990
5. Bueno, M., Martínez-Sánchez, J., González-Jorge, H., Lorenzo, H.: Detection of geometric keypoints and its application to point cloud coarse registration. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences 41, 187–194 (2016)
6. Chen, Y., Medioni, G.: Object modeling by registration of multiple range images. In: Proceedings. 1991 IEEE International Conference on Robotics and Automation. pp. 2724–2729 vol.3 (1991)
7. Choy, C.B., Dong, W., Koltun, V.: Deep global registration. CoRR abs/2004.11540 (2020), https://arxiv.org/abs/2004.11540
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
9. Eitel, J.U., Höfle, B., Vierling, L.A., Abellán, A., Asner, G.P., Deems, J.S., Glennie, C.L., Joerg, P.C., LeWinter, A.L., Magney, T.S., et al.: Beyond 3-d: The new spectrum of lidar applications for earth and ecological sciences. Remote Sensing of Environment 186, 372–392 (2016)
10. Elseberg, J., Borrmann, D., Nüchter, A.: Efficient processing of large 3d point clouds. In: 2011 XXIII International Symposium on Information, Communication and Automation Technologies. pp. 1–7 (2011)
11. Fischer, K., Simon, M., Olsner, F., Milz, S., Gross, H.M., Mader, P.: Stickypillars: Robust and efficient feature matching on point clouds using graph neural networks. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 313–323 (June 2021)

12. Fürlinger, K., Fuchs, T., Kowalewski, R.: DASH: A C++ PGAS library for distributed data structures and parallel algorithms. In: Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016). pp. 983–990. Sydney, Australia (Dec 2016)
13. Gojcic, Z., Zhou, C., Wegner, J.D., Wieser, A.: The perfect match: 3d point cloud matching with smoothed densities. CoRR abs/1811.06879 (2018), http://arxiv.org/abs/1811.06879
14. Gumhold, S., Wang, X., MacLeod, R.: Feature extraction from point clouds. Proceedings of 10th international meshing roundtable 2001 (11 2001)
15. Huang, R., Xu, Y., Hoegner, L., Stilla, U.: Semantics-aided 3d change detection on construction sites using uav-based photogrammetric point clouds. Automation in Construction 134, 104057 (2022)
16. Ioannou, Y., Taati, B., Harrap, R., Greenspan, M.: Difference of normals as a multi-scale operator in unorganized point clouds. In: 2012 Second International Conference on 3D Imaging, Modeling, Processing, Visualization & Transmission. pp. 501–508 (2012)
17. Kammerl, J., Blodow, N., Rusu, R.B., Gedikli, S., Beetz, M., Steinbach, E.: Real-time compression of point cloud streams. In: 2012 IEEE International Conference on Robotics and Automation. pp. 778–785 (2012)
18. Kromer, R.A., Abellán, A., Hutchinson, D.J., Lato, M., Edwards, T., Jaboyedoff, M.: A 4d filtering and calibration technique for small-scale point cloud change detection with a terrestrial laser scanner. Remote Sensing 7(10), 13029–13052 (2015), https://www.mdpi.com/2072-4292/7/10/13029
19. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with mapreduce: a survey. AcM sIGMoD record 40(4), 11–20 (2012)
20. Lokugam Hewage, C.N., Laefer, D.F., Vo, A.V., Le-Khac, N.A., Bertolotto, M.: Scalability and Performance of LiDAR Point Cloud Data Management Systems: A State-of-the-Art Review. Remote Sensing 14(20), 5277 (2022), https://www.mdpi.com/2072-4292/14/20/5277
21. Munoz, D., Bagnell, J.A., Vandapel, N., Hebert, M.: Contextual classification with functional max-margin markov networks. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2009)
22. Osada, R., Funkhouser, T., Chazelle, B., Dobkin, D.: Shape distributions. ACM Transactions on Graphics 21(4), 807–832 (Oct 2002)
23. Pajić, V., Govedarica, M., Amović, M.: Model of Point Cloud Data Management System in Big Data Paradigm. ISPRS International Journal of Geo-Information 7(7), 265 (2018), http://www.mdpi.com/2220-9964/7/7/265
24. Polewski, P., Yao, W., Heurich, M., Krzystek, P., Stilla, U.: Learning a constrained conditional random field for enhanced segmentation of fallen trees in als point clouds. ISPRS Journal of Photogrammetry and Remote Sensing 140, 33–44 (2018), geospatial Computer Vision
25. Pomerleau, F., Liu, M., Colas, F., Siegwart, R.: Challenging data sets for point cloud registration algorithms. The International Journal of Robotics Research 31(14), 1705–1711 (Dec 2012)
26. Qi, C.R., Su, H., Mo, K., Guibas, L.J.: Pointnet: Deep learning on point sets for 3d classification and segmentation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (July 2017)
27. Qi, C.R., Yi, L., Su, H., Guibas, L.J.: Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.)

Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017), https://proceedings.neurips.cc/paper_files/paper/2017/file/d8bf84be3800d12f74d8b05e9b89836f-Paper.pdf
28. Rusu, R.B., Blodow, N., Beetz, M.: Fast point feature histograms (fpfh) for 3d registration. In: 2009 IEEE International Conference on Robotics and Automation. pp. 3212–3217 (2009)
29. Tobler, W.R.: A computer movie simulating urban growth in the detroit region. Economic Geography 46, 234–240 (1970), http://www.jstor.org/stable/143141
30. Vetrivel, A., Gerke, M., Kerle, N., Nex, F., Vosselman, G.: Disaster damage detection through synergistic use of deep learning and 3d point cloud features derived from very high resolution oblique aerial images, and multiple-kernel-learning. ISPRS Journal of Photogrammetry and Remote Sensing 140, 45–59 (2018), https://www.sciencedirect.com/science/article/pii/S0924271616305913, geospatial Computer Vision
31. Vos, S., Anders, K., Kuschnerus, M., Lindenbergh, R., Höfle, B., Aarninkhof, S., de Vries, S.: A high-resolution 4d terrestrial laser scan dataset of the kijkduin beach-dune system, the netherlands. Scientific Data 9(1), 191 (2022)
32. Weis, M., Gutjahr, C., Rueda Ayala, V., Gerhards, R., Ritter, C., Schölderle, F.: Precision farming for weed management: techniques. Gesunde Pflanzen 60(4), 171–181 (2008)
33. Xie, L., Furuhata, T., Shimada, K.: Multi-resolution graph neural network for large-scale pointcloud segmentation. CoRR abs/2009.08924 (2020), https://arxiv.org/abs/2009.08924
34. Xu, Q., Xu, Z., Philip, J., Bi, S., Shu, Z., Sunkavalli, K., Neumann, U.: Point-nerf: Point-based neural radiance fields. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 5438–5448 (2022)
35. Xu, Y., Boerner, R., Yao, W., Hoegner, L., Stilla, U.: Pairwise coarse registration of point clouds in urban scenes using voxel-based 4-planes congruent sets. ISPRS Journal of Photogrammetry and Remote Sensing 151, 106–123 (2019)
36. Yue, X., Wu, B., Seshia, S.A., Keutzer, K., Sangiovanni-Vincentelli, A.L.: A lidar point cloud generator: from a virtual world to autonomous driving. In: Proceedings of the 2018 ACM on International Conference on Multimedia Retrieval. pp. 458–464 (2018)
37. Zhang, R., Li, G., Wiedemann, W., Holst, C.: Kdo-net: Towards improving the efficiency of deep convolutional neural networks applied in the 3d pairwise point feature matching. Remote Sensing 14(12) (2022), https://www.mdpi.com/2072-4292/14/12/2883
38. Zwally, H.J., Schutz, B., Abdalati, W., Abshire, J., Bentley, C., Brenner, A., Bufton, J., Dezio, J., Hancock, D., Harding, D., et al.: Icesat's laser measurements of polar ice, atmosphere, ocean, and land. Journal of Geodynamics 34(3-4), 405–445 (2002)