DOI: 10.1111/tgis.12458

WILEY Transactions in GIS

RESEARCH ARTICLE

Real-time inverse distance weighting interpolation for streaming sensor data

Qinghan Liang[1] | Silvia Nittel[1] | John C. Whittier[1] | Sytze de Bruin[2]

[1]School of Computer and Information Science, University of Maine, Orono, Maine

[2]Laboratory of Geo-Information Science and Remote Sensing, Wageningen University and Research, Wageningen, Netherlands

Correspondence
Silvia Nittel, School of Computer and Information Science, University of Maine, Orono, Maine 04469.
Email: silvia.nittel@maine.edu

Funding information
National Science Foundation, Grant/Award No: 1527504

Abstract

With advances in technology and an increasing variety of inexpensive geosensors, environmental monitoring has become increasingly sensor dense and real time. Using sensor data streams enables real-time applications such as environmental hazard detection, or earthquake, wildfire, or radiation monitoring. In-depth analysis of such spatial fields is often based on a *continuous* representation. With very large numbers of concurrent observation streams, novel algorithms are necessary that integrate streams into rasters, or other continuous representations, continuously in real time. In this article, we present an approach leveraging data stream engines (DSEs) to achieve scalable, high-throughput inverse distance weighting (IDW). In detail, we designed and implemented a novel stream query operator framework that extends general-purpose DSEs. The proposed framework includes a two-panel, spatio-temporal grid-based index and several algorithms, namely the Shell and *k*-Shell algorithms, to estimate individual grid cells efficiently and adaptively for different sampling scenarios. For our performance experiments, we generated several different spatio-temporal stream data sets based on the radiation deposits in the Fukushima region after the nuclear accident of 2011 in Japan. Our results showed that the *k*-Shell algorithm of the proposed framework produces a raster based on 250k observation streams in under 0.5 s using a state-of-the-art workstation.

1 | INTRODUCTION

Environmental monitoring becomes increasingly sensor dense and real time, supported through advances in technology and a variety of inexpensive (geo)sensors. Geosensor networks are deployed in various environments, such as urban observation (Mead et al., 2013; Murty et al., 2008; Resch, Mittlboeck, Girardin, Britter, & Ratti, 2009; Sanchez et al., 2011; Xiao et al., 2017), smart forests (Zhong, Kealy, Sharon, & Duckham, 2015), precision agriculture (Agrible, Inc., 2016), earthquake monitoring (Faulkner et al., 2011; Hudnut, Bock, Galetzka, Webb, & Young, 2002; Kong et al., 2015), or radiation monitoring (Safecast, 2016). The geosensors of a network, mobile or stationary, sample concurrently and often at high temporal frequency; geosensor networks in smart cities, emergency monitoring or precision agriculture, can reach up to millions of concurrently sampling sensors. The observations are streamed continuously to the cloud and servers (Nittel, 2009; Sanchez et al., 2011; Zhou, Chen, & Chen, 2017). With such sensing environments at hand, interest in real-time analysis of phenomena and events has increased. Dense geosensor networks allow us to observe regional spatial fields such as urban air pollution, flooding, or radioactive fallout at much higher resolution than ever before. However, these opportunities also pose novel challenges to existing processing and analysis software, as well as to geostatistics.

Using spatial fields (Liang, Nittel, & Hahmann, 2016), their in-depth analysis is often based on a carefully created *continuous* representation. Therefore, the observation streams need to be integrated automatically on-the-fly and in real time into rasters, tessellations, or other continuous representations. This requirement poses several significant challenges: first, the continuous process of data collection might change over time if sensors are mobile, yet the creation of an accurate representation of the spatial field is desired; second, novel spatial interpolation algorithms need to be able to process potentially very large numbers of samples efficiently and produce new rasters under a few seconds for real-time GIS applications. While the density of geosensors and their sampling rate depends on an application, for this article we set a target throughput rate for the real-time spatial interpolation algorithms to a quarter of a million updates/second using a state-of-the-art workstation.

Rasters are a common representation of spatial fields. They are often created using kriging (Krige, 1951), since kriging calculates both an unbiased estimate and the kriging variance. However, kriging is computationally expensive, since it requires computing a variogram, fitting a model to the variogram, and the kriging process itself, which has a computational complexity of $O(n^3)$ or $O(n^4)$ with n observations. Efforts have been made to adapt kriging for data streams (Lorkowski & Brinkhoff, 2015a; Srinivasan, Duraiswami, & Murtugudde, 2010; Zhong, Kealy, & Duckham, 2016), but still the computational complexity of the kriging step itself severely limits high throughput. Thus, kriging is a less likely candidate for integrating up to 250k sensors in real time. On the other hand, inverse distance weighting (IDW) (Lam, 1983; Mitas & Mitasova, 1999) has a computational complexity of $O(n)$, and has a higher potential to scale to very large observation sets. In recent years, work has been proposed to increase throughput for IDW (Hennebohl, Appel, & Pebesma, 2011; Huang, Bu, Tao, & Tan, 2016; Mei, Xu, & Xu, 2016); these approaches focused on designing algorithms that exploit GPU (graphics processing unit) architectures and parallelize IDW. However, these approaches are not directly applicable to data *stream* processing. Stream processing requires algorithms that both scale up to high throughput as well as automatically and adaptively keep up with massive streams that continuously push updates with real-time query results.

A promising approach to real-time analytics over sensor data streams is to use data stream engine (DSE) tools. Much like database management systems (DBMSs), DSEs are designed to do the heavy lifting of data management while the user can focus on formulating queries over streams. DSEs have been developed for the throughput needs of real-time data stream applications. Compared to DBMSs with a throughput of 500 updates/s (Stonebraker, Çetintemel, & Zdonik, 2005), DSEs scale up to 1.5 million updates/s per CPU core (Carbone et al., 2015). DSEs simplify the programming of analysis tasks over streams by providing a stream query language, and also include a library of customizable functions. Today, many commercial and open source DSEs are available (Whittier, Nittel, & Subasinghe, 2017). DSEs have been successfully applied in traffic and critical infrastructure

monitoring (Ali, Chandramouli, Raman, & Katibah, 2010; Galic, Baranovič, Krízanovič, & Mésković, 2014); they are also promising candidates for monitoring continuous phenomena based on large numbers of sensor data streams and real-time requirements for query results.

In this article, we propose a query stream operator framework for DSEs that performs scalable, real-time IDW for large numbers of moving sensor streams. The framework consists of a set of stream-based query operator implementations that execute in parallel and in a pipelined way. The first component is a main-memory-based, two-panel grid index that indexes each new sensor tuple spatially and supports efficiently locating samples via the shell list template-based search. The next operator (i.e., the interpolator manager) consumes the indexed data and controls a collection of parallelized cell interpolator operators that are built to minimize memory footprint and decrease execution time. We implemented several cell interpolator algorithms—the single shell (Shell), *k*-adaptive shell (*k*-Shell), and limited *k*-adaptive shell (*ak*-Shell) approaches—which deliver improved estimation results for different sampling scenarios. We tested the framework's performance using different data streams generated based on the radiation deposit event in the Fukushima region following the nuclear accident of March, 2011 in Japan. The streams simulate sensors moving with varying location patterns, such as dense (urban) and sparse (cross-country) road networks, as well as several random sampling patterns. In the first part of the performance test, we evaluate the impact of algorithm parameters such as search radius, IDW power parameter, and *k* on the root mean square error (RMSE). In the second part, we test the run-time and throughput for the parameters with the lowest RMSE. The tests show that the *k*-Shell approach provides the highest throughput and the best RMSE under all sampling scenarios. It achieves the generation of rasters based on 250k moving sensors updating every second into a 500 × 500 grid in under 0.4 s.

The remainder of the article is structured as follows. Section 2 includes the background, discussing the problems of high-throughput and stream-based spatial interpolation. In Sections 3 and 4, we present our approach. Section 5 discusses our experimental data sets, and Section 6 contains the performance tests and results. Section 7 summarizes the results and discusses future work.

2 | BACKGROUND

2.1 | Scalability of spatial interpolation methods

The computational complexity of a spatial interpolation method has a significant impact on its scalability (i.e., its potential to write algorithms that still perform efficiently although the number of input points increases significantly). For instance, typical implementations of kriging (Krige, 1951) have a computational complexity of $O(n^4)$ or $O(n^3)$ if improved (Lorkowski & Brinkhoff, 2015b; Zhong et al., 2016). Although kriging delivers an unbiased estimate and a type of error map (kriging variance), its computational complexity makes it an unlikely spatial interpolation method for integrating up to 100k moving sensor streams in real time without using massive computational resources. Srinivasan et al. (2010), Lorkowski and Brinkhoff (2015a), and Zhong et al. (2016) have adapted kriging for data streams; this work has confirmed that kriging does not scale to large numbers of sensors, with kriging taking 2 s to process 20 tuples using a 50 × 50 raster grid (Zhong et al., 2016) and 30 s to process 10 tuples using a 150 × 150 raster grid (Lorkowski & Brinkhoff, 2015a, 2015b), respectively.

On the other hand, IDW (Lam, 1983; Mitas & Mitasova, 1999; Shepard, 1968) has a computational complexity of $O(n)$; its potential to scale to significantly larger observation sets efficiently is more promising. IDW is calculated based on the following equation. To interpolate value u at a cell center x based on samples $u_i = u(x_i)$ for $i = 1, 2, \ldots, n$, the following is calculated:

$$u(\mathbf{x}) = \begin{cases} \dfrac{\sum_{i=1}^{n} w_i(\mathbf{x}) u_i}{\sum_{i=1}^{n} w_i(\mathbf{x})}, & \text{if } d(\mathbf{x},\mathbf{x}_i) \neq 0 \text{ for all } i \\ u_i, & \text{if } d(\mathbf{x},\mathbf{x}_i) = 0 \text{ for some } i \end{cases} \tag{1}$$

where $w_i(\mathbf{x}) = \frac{1}{d(\mathbf{x},\mathbf{x}_i)^p}$

The parameter x denotes the interpolated point location, x_i is a sample location, d is the given distance from the known point x_i to the unknown point x, and n is the total number of known points used in the interpolation; p is a positive real number, called the power parameter.

2.2 | Scaling inverse distance weighting interpolation to high throughput

In recent years, increased interest has been placed on revising algorithms for IDW (Lam, 1983; Mitas & Mitasova, 1999) so that software can be developed that can cope with larger data sets in near real time. The main task of increasing throughput for IDW is to parallelize time-consuming tasks such as finding the relevant, spatially neighboring observation points for an estimated point. Then, the samples' distances to the estimated point and, thus, their weights for their input to the estimation need to be calculated. For instance, a raster consists of a million grid points for a 1,000 × 1,000 cell grid, and estimating each cell fast is the key to increased throughput.

Current approaches can roughly be divided into two categories: first, work that is based on using GPUs or grid technology to parallelize processing IDW (Guan, Kyriakidis, & Goodchild, 2011; Hennebohl et al., 2011; Huang et al., 2016; Mei et al., 2016) and second, work that is data stream based (Nittel, Whittier, & Liang, 2012; Whittier, Nittel, Liang, & Plummer, 2013). The first category exploits GPU architectures to speed up processing large observation sets. These redesigned methods are used for stored data sets. In the domain of "big data," this type of processing falls under methods for dealing with the *volume* (of data). The second category of high-throughput spatial interpolation consists of stream-based approaches, which fall under the category of methods dealing with the *velocity* (of data). Stream-based implementations are different, in that they need algorithms that continuously and adaptively keep up with the pushed updates from devices as they come in, can absorb all data, and produce rasters in near real time. Often, stream-based implementation is designed to be integrated in data stream engines to take advantage of their generic stream data management support and reduce overall application development cost.

2.3 | Streaming inverse distance weighting

Informally, a sensor data *stream* is a continuously updating spatio-temporal relation with append-only tuples created by a single geosensor or an entire geosensor network if all geosensors have a homogeneous schema (Nittel, 2015). Representing the updates of an individual sensor as a time series of data records has been common practice in geographic and scientific applications. However, streams are different from stored time series in that streams are continuously produced and processed immediately as they arrive, that is, stream processing is tied to the stream delivery, is adaptive to it, and is expected to deliver results in real time.

In the database community, DSEs have been developed with a similar intention as DBMSs, that is to provide a powerful data management tool that does much of the heavy lifting of query execution over streams in real time. DSEs simplify application development by focusing on simply writing stream queries that plug in to the remainder of the DSE processing engine. While queries in a DBMS are written over relations, queries in a DSE are written over streams. Since a stream is unbounded and potentially infinite, queries are executed over well-defined portions of streams, so-called *stream windows*. A window can be the data of the last 5 s or 5 min, or any user-defined time interval, depending on the application requirements for windows. The data within a stream query window is a finite set of data, and all processing has to finish before the end of a query window, that is, before the data of the next window arrives. A query is automatically executed again and again for each new stream window (i.e., continuous query processing).

To use DSEs effectively, algorithms need to adhere to the stream query processing paradigm, that is, each query operator must be implemented as a data-driven, purely main-memory operator. Such an operator grabs tuples from an input queue that stores tuples in the order in which they arrive; an operator has only

temporary access to each tuple for processing (one pass) and it must perform adaptively and efficiently to meet the processing deadlines for each query window. On the other hand, a traditional algorithm typically fetches the necessary data from the disk and often uses disk-based spatial indexes to perform tasks such as interpolation. Such an algorithm also assumes that it has random access to all the data, as long as it is necessary to perform a task. However, the assumption of disk I/O, disk-based spatial indexing, and the lack of memory compactness becomes a severe bottleneck in such algorithms for very high throughput (e.g., 100k data points/s).

The initial approach to implementing stream queries performing spatial functions such as nearest-neighbor search for mobile objects was to implement a DSE query operator graph, feed the stream into the data stream engine, but instead of implementing a complete set of operators implemented as fully as stream-based operators, some operators were implemented as wrappers and internally called a traditionally implemented spatial library as available in spatial DBMSs. Kazemitabar, Demiryurek, Ali, Akdogan, and Shahabi (2010) and Miller et al. (2011) report that this approach significantly limits the throughput of DSE queries. Venkatesan (2013) experimented with a similar approach, and implemented a stream-based spatial interpolation query in Oracle Streams and called the Oracle Georaster application programming interface to transform spatio-temporal point clouds per query window into a raster. The results showed that generating a 512 × 512 raster using 60 sensors and a search radius of 10 cells takes around 6 min on a state-of-the-art workstation, which cannot be considered real time and only works for applications with infrequent updates and very long query windows (e.g., a query result every 30 min). Processing 100k updates took too long for testing.

Thus, to achieve a throughput of streams up to 100k tuples/s and a query response time of about 1–2 s, spatial functionality such as spatial interpolation needs to be algorithmically redesigned for stream-based processing, and the following requirements hold:

- The algorithms use main memory only, since disk I/O is a bottleneck. On the other hand, main memory is a limited resource in a DSE since all necessary incoming data and query operators share the available memory. Therefore, the memory footprint of a spatial interpolation method needs to be kept small while retaining the interpolation quality, even for large grids and large data sets. The algorithms need to be implemented as one-pass algorithms (i.e., assuming only sequential access to data). Furthermore, they need to be scalable and adaptable to the amount of incoming data.
- Sampling depends on varying underlying spatial constraints, such as movement of sensors on road networks, data skew due to lack of sampling in water bodies (rivers, lakes, oceans), number of participating sensors at any point in time, and other parameters. This can lead to sampling skew over both space and time. This poses new requirements on how to support interpolation quality within automatic, repeated execution over continuously new data.[1]
- The computational complexity of a spatial interpolation method has a significant impact on the scalable performance for very large numbers of data points. Overall, stream queries consist of pipelined operators that work autonomously, use input queues, and place their results into output queues. All operators work simultaneously and should be non-blocking, that is, not assume that all data needs to be available before processing starts. Bottlenecks in an algorithm need to be parallelized to increase throughput.

To the best of our knowledge, this work is the first of its kind adapting IDW to high-throughput, real-time spatial interpolation of real-time sensor streams. As mentioned previously, Srinivasan et al. (2010), Lorkowski and Brinkhoff (2015a), and Zhong et al. (2016) have adapted kriging for data streams processing using a stream-based operator paradigm. However, due to the computational complexity of kriging, the approaches do not scale well to very high throughput, and these approaches are limited to small data sets (20 tuples/s).

3 | DSE FRAMEWORK FOR REAL-TIME SPATIAL INTERPOLATION

Our objective was to design a scalable, high-throughput, stream-based framework that computes IDW on-the-fly in real time using mobile sensor data streams. To set a concrete target line, we aimed at developing a framework that is able to transform up to 250k samples/s continuously into rasters, with a raster produced every 1–2 s.

3.1 | Framework overview

Figure 1 shows an overview of the proposed framework. In the lower left part of the figure, geosensors in a geographic area and a query over that region are depicted. The upper half of the figure depicts the proposed stream query framework (described below); the lower right part shows a single output of the stream query (i.e., a continuous representation of one query window). The stream query framework for continuous stream-based IDW consists of a collection of cooperating, non-blocking stream query operators. Each operator performs a sub-function of the stream-based transformation of sensor data streams into a stream of rasters. The framework consists of the following operators:

- **Index operator**: tuples arrive ordered by time in streams, but for spatial interpolation tuples that need to be searched by location to find relevant tuples for a cell estimation fast. The index operator's task is to spatially index arriving updates (i.e., to insert them into a two-panel spatial index). It also needs to rapidly eliminate outdated tuples at the end of a window.
- **Cell interpolator manager:** the cell interpolator manager is aware of window semantics, cooperates with the index operator to select the correct time panel of the spatial index, and manages a pool of cell interpolator operators that perform grid cell interpolation in parallel.
- **Cell interpolator operator**: a cell interpolator operator estimates the value of a single grid cell of the output raster. It locates all relevant tuples in the index, calculates their distances from the cell center, and computes the value.
- **Raster assemble operator**: the raster assemble operator keeps track of all arbitrary interpolated grid cells for a single raster, since the interpolated cells can arrive in various orders from the interpolators, and produces the output grid.

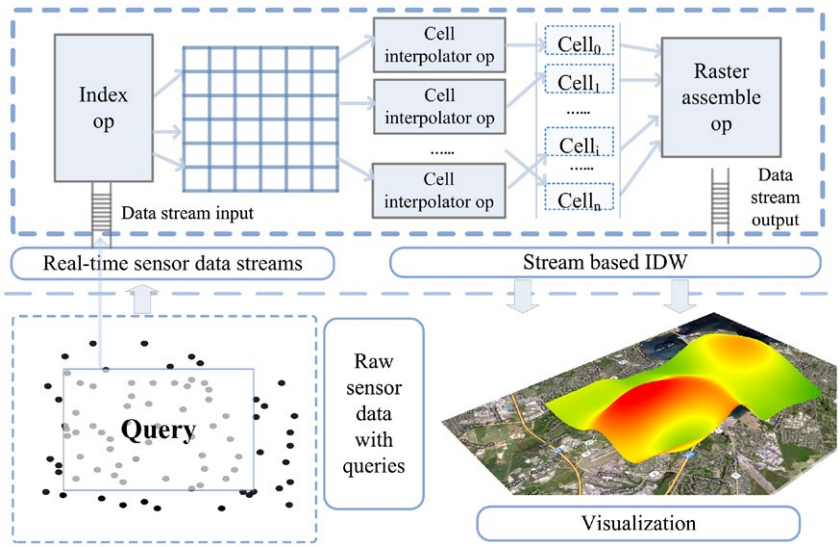


FIGURE 1 Stream query operator framework for IDW

Overall, the sensor data streams flow into the query operator framework; the spatial grid index operator incrementally indexes all the tuples of a query window. A part of the index is pushed to the next set of operators, the cell interpolators and their manager. Each cell interpolator places an estimated cell in the output queue, which is used by the grid assemble operator. All data are stored in main memory to achieve fast access and high throughput. Tuples that are no longer relevant to a query are usually stored to disk or discarded, depending on the application.

3.2 | Spatial indexing operator

The index operator maintains a scalable spatial index that is flexible in scale and throughput in a stream-based environment. This means that it needs to know the length of user-defined stream query windows (i.e., varying window length) and is able to index a massive number of updates efficiently within one query window while minimizing memory space. Once tuples are outdated, they need to be flushed efficiently from the index (for the index to be filled again with the tuples of the next window). The index operator continuously inserts and deletes tuples with massive rates into the index. Search performance is also a crucial performance parameter.

Today, several entity-based spatial index structures such as R-trees (Guttman, 1984) are considered state-of-the-art spatial index structures due to their height balance; however, such a tree-based spatial index structure does not perform well in this type of environment. An R-tree's performance rapidly deteriorates with massive numbers of incremental inserts and deletes, due to the constantly required reorganization of the tree. A space-based grid index that is stored purely in main memory provides better performance for the necessary creation, deletion, and search performance for massive inserts and deletes, since it never requires a reorganization of the grid structure itself.

Algorithm 1 Spatial Grid Indexing Algorithm

```
1: Initialize grid index panel P_A;
2: Initialize grid index panel P_B;
3: Insert tuples into first panel, Set currentIndexWindow ← P_A;
4: while True do
5:     At the end of current query window
6:     if timer.endCurrentwindow() = True then
7:         Tuples in the current window to buffer in queue for next operators:
8:         bufferQueue.insert(currentIndexWindow);
9:         Open the second grid panel and reset the query window:
10:        currentIndexWindow ← SwapGridIndexPanelsAB();
11:        currentIndexWindow.resetTimer();
12:    Fetch next tuple from stream: t ← inputQueue.dequeue();
13:    Insert tuple to grid index panel:
14:    if t.isInCurrentwindow() = True then
15:        Locate grid cell index: gridCell_ij ← t.locateCell();
16:        currentIndexWindow.gridCell_ij.dataList.insert(t);
```

In our architecture, the grid index has the following algorithms:

- **Initialization:** the grid index operator dynamically creates a spatial grid in the size of the output raster, for instance 1,000 × 1,000 cells (see Algorithm 1). The grid index has two identical panels, P_A and P_B, that is, two data structures with empty grid cells of the same size that are used concurrently, but for different tasks (see Figures 2 and 3). The index operator initializes the grid index with the user-defined windows size (e.g., 10 s).

- **Insertion:** the index operator processes the incoming stream by opening a query window, and inserts the tuples in their arrival order into the first panel P_A of the grid index. Each grid cell covers a geographic region, and each observation pertains to its original location. To keep the index compact, only pointers to the original tuples are managed in the index. Also, since up to 250k tuples need to be inserted in the overall grid index, tuples *within* an individual grid cell are not sorted by distance to the grid cell center or sorted in any other way; they are just added to the grid cell tuple list.

The spatial index operator inserts all tuples for window w_i into panel P_A. At the end of query window w_i (e.g., w_i = [0 s, 10 s]), all tuples that arrived during the window have been inserted and grid panel P_A is pushed to the following operators for consumption (see Figure 2). Meanwhile, the spatial index operator opens the second grid panel, P_B. During window w_{i+1} (w_{i+1} = [10 s, 20 s]), the index operator incrementally inserts all tuples of this window into the index as they arrive (see Figure 3). Once this window is closed, the index operator pushes panel P_B to the following operators, and it flushes the first spatial grid panel P_A with the data of w_i and initializes it for the window w_{i+2} (we assume a tumbling window only in this article).

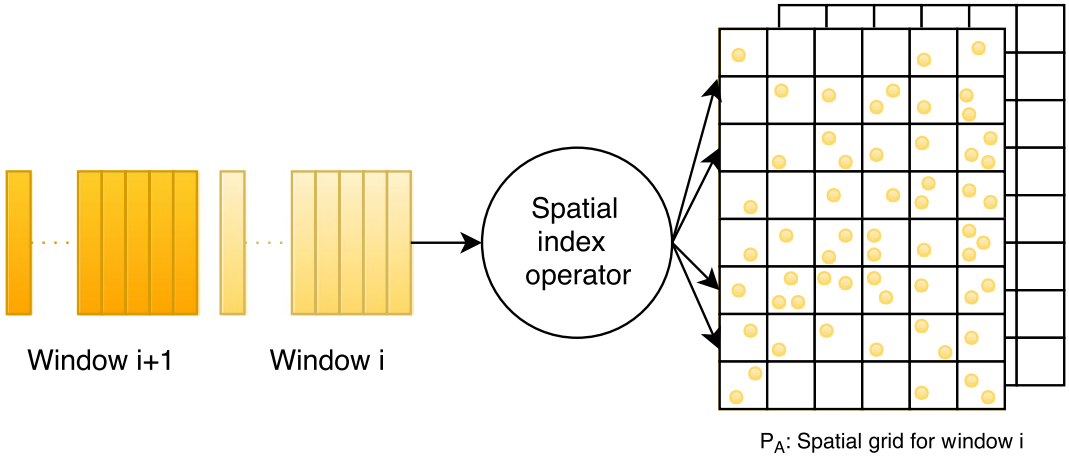


FIGURE 2 Spatial index operator for window w_i

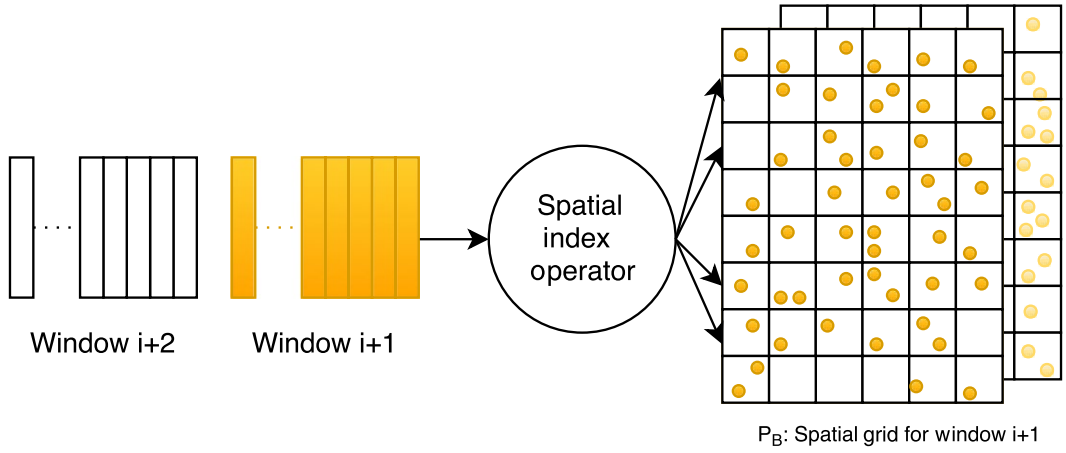


FIGURE 3 Spatial index operator for window w_{i+1}: (a) search radius; and (b) cells by expanding radius distance increments

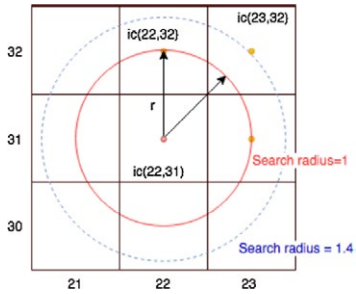
- **Deletion:** the spatial grid index contains two panels, one for the current window and one for the previous window. This design was chosen since having only a single panel would block all the following operators and they would have to wait until indexing was completed before they could resume interpolation. With the two-panel design, the spatial index operator and the grid cell interpolator operators can work in parallel (non-blocking). The spatial grid operator recycles the panels continuously and flushes a panel before reusing it for a new window.

3.3 | Spatial search using the shell list template

Each spatial interpolation operator estimates a cell ec_i (estimated cell) at a time. To do so, the interpolation operator accesses the corresponding index cell ic_i (index cell) in the spatial grid index. A search radius r is defined in order to identify the necessary samples for estimation around each ic_i. For instance, Figure 4 shows different search radii for cell $ic_{(22,31)}$ (center). The cells within search radius $r = 1$ are determined by the *center* of cell $ic_{(22,31)}$ and include all cells whose cell center distance to the center of $ic_{(22,31)}$ is equal to 1. In this case, only the cell centers of the four cells to the north, south, east, and west (red circle) are equal to $r = 1$, but the samples of, for example, $ic_{(23,32)}$ are not included.

The spatial grid index makes sample points efficiently searchable by distance from a cell center of any ic_i. The search starts with the cell ic_i and expands iteratively outward. A cell interpolator needs to visit all the relevant grid index cells within a given search radius. To speed up the identification of those cells, we introduce the shell list template (SLT). The SLT is a data structure that is created once per long-running stream query and shared between all cell interpolator instances. It is a look-up structure that allows each cell interpolator to calculate the grid index cells it needs to visit to interpolate an estimated cell. The SLT is based on (x, y) offsets and is compacted to use minimal main-memory storage, that is, it contains a quarter of the look-up cells and the remaining cells are calculated by mirroring. The memory size of a SLT for $r = 32$ is around 4 KB.

The SLT consists of a list of shells and the shells are ordered by ascending radius (see Figure 5a). Since the SLT is a *template*, it is not tied to any particular cell; therefore, it starts with a so-called center cell. A **shell** $sh_d(cc)$ is a collection of offset pairs, all of which have the same distance d to the center cell (the cell to be estimated); it identifies the cells j whose cell centers are all equal to the spatial distance d from the central cell. For instance, in Figure 5b, two shells are shown, $sh_1(cc)$ and $sh_{1.4}(cc)$, around the central cell. The construction of a SLT starts with the central cell; a shell with $d = 0$ has offsets (0, 0) to the center cell's (x, y) coordinates. The $sh_1(cc)$ contains the cells with offsets (1, 0) and (0, 1) from cc (as well as (0, −1) and (−1, 0), however they can be calculated by mirroring, and thus are not stored in the SLT). The basic idea of the SLT is that the relevant index cells for estimation of a



(a) Search radius

4.2	3.6	3.2	3	3.2	3.6	4.2
3.6	2.8	2.2	2	2.2	2.8	3.6
3.2	2.2	1.4	1	1.4	2.2	3.2
3	2	1	0	1	2	3
3.2	2.2	1.4	1	1.4	2.2	3.2
3.6	2.8	2.2	2	2.2	2.8	3.6
4.2	3.6	3.2	3	3.2	3.6	4.2

(b) Cells by expanding radius distance increments

FIGURE 4 Search radius for grid index cells: (a) nested list structure of shell list template; and (b) center cell, shell $sh_1(cc)$ (red), and shell $sh_{1\cdot4}(cc)$

certain distance can be calculated with this additive offset template. Revisiting the example in Figure 4, the shell $sh_0(22, 31)$ is used to retrieve the samples of cell (22, 31) itself; the shell $sh_1(22, 31)$ retrieves the samples of cells (23, 31), (22, 32), (21, 31), (22, 30), and so on.

Algorithm 2 Shell List Template Algorithm

```
1: shellElemList ← LinkedList < ShellElem > ()
2: searchRadius ←  desired maximum radius
3: radiusSquared ← searchRadius * searchRadius
4: for x = 0 to searchRadius do                                  ▷ Create flat list of all ShellElem
5:         for y = 0 to searchRadius do
6:                 if x * x + y * y < = radiusSquared then
7:                         shellElemList.add(ShellElem(x, y));
8: sortedShellElems ← shellElemList.sort();                      ▷ Sort shellElemList in ascending distance
9: highDistance ← - 1;
10: shellList ← LinkedList < LinkedList < ShellElem > > ();      ▷ Initialize list of Shells
11: currentShell ← NULL; ▷ Declare currentShell variable
12: for all shellElem ∈ sortedShellElems do                      ▷ Group all equidistant shellElem
13:         x ← shellElem.x;
14:         y ← shellElem.y;
15:         squaredDistance ← x * x + y * y;
16: if squaredDistance = highDistance then                       ▷ If the Shell for this distance exists
17:                 currentShell.add(shellElem);                 ▷ Continue to fill the current Shell
18:         else
19:                 currentShell ← LinkedList < ShellElem > ();  ▷ Create a new shell
20:                 currentShell.add(shellElem);                 ▷ Begin populating the created Shell
21:                 shellList.add(currentShell);                 ▷ Add Shell to shellList
22:                 highDistance ← squaredDistance;              ▷ Update distance of Shell being populated
```

In Algorithm 2, a list of shell elements (shellElemList) is created; this is a "flat" list of shell elements, and each shell element is an offset pair. The user-defined search radius is given, and the offset pairs of all cells that fall within the search radius are added to the "flat" shell element list. Next, this shell element list is sorted by ascending distance from the center cell. Now, the shell list itself is created; it is a two-level list of lists describing concentric shells of successive distances from a central cell *cc* (see Figure 5a). The variable current shell is declared, and the sorted list of shell elements is processed element by element. Logically, the first shell element is (0, 0); its squared distance is 0 and thus, 0 > *highDistance*. The current shell is created as a list of shell elements (line 19). The current shell *element* is added to the new shell (line 20), and the shell itself is added to the shell list (line 21). The high distance is updated. Next, the shell element (1, 0) is processed. The squared distance is 1, and thus greater than the current high distance. A new shell is created, (1, 0) is added, the new shell is added to the shell list, and the algorithm proceeds with the shell element (0, 1). Here, the squared distance is equal to the high distance, and the shell element is added to the current shell. The algorithm proceeds until all shell elements are sorted into equidistant shells that are ordered by *r*. Since the SLT is symmetric, only coordinates with positive *x* and *y* are stored in each shell; the full shell can be constructed through mirroring.

As mentioned, the SLT serves as a look-up structure; it is never copied and all samples only exist once in the grid index itself. All cell interpolators work in parallel and share the data. Thus, the memory consumption only depends on the number of arrival tuples and is constant otherwise.

4 | STREAM-BASED INTERPOLATION OF GRID CELLS

In this section, we present the stream-based interpolation operator, which estimates the value of a cell using IDW. Since sampling patterns can vary greatly in moving-sensor scenarios, the objective is to come up with both a uniform and high interpolation quality across the raster, independent of the sampling patterns. Additionally, main-memory constraints and execution time are considered. We introduce three alternative algorithms: the Shell algorithm, the *k*-Shell algorithm, and the *ak*-Shell algorithm.

4.1 | Shell-based IDW interpolation algorithm

Interpolating the estimated value u_{ec_i} of a grid cell (center) ec_i consists of three computational steps: (1) finding collocated samples; (2) calculating the distance $d(ec_i, x_j)$ between the cell center ec_i and each sample x_j; and (3) computing the IDW formula.

In the Shell approach, the Shell cell interpolator is set with a user-defined search radius *r*. To locate the samples within the search radius around the cell *c*, the Shell approach uses the SLT. First, the cell interpolation operator that implements IDW *incrementally* processes all tuples in an index cell at a time. The operator calculates the distance of each tuple from the cell center of ec_i, computes the weight for the tuple, and adds it to a running intermediate value of the interpolation function.[2] It proceeds with the following tuple in the current index cell (or if all tuples in the cell are processed, it uses the SLT to determine the next cell). The cell interpolation operator terminates when all tuples contained in index cells determined by the search radius are processed and the estimated value is calculated (see Algorithm 3 for the complete algorithm).

This algorithm uses variable numbers of samples for each interpolated cell, since the distribution of samples in cells depends on the sampling pattern of the moving sensors at t_i, and this distribution changes from time stamp to time stamp. With dense sampling patterns, many tuples are considered during interpolation if a fixed search radius is used; this required an increasing amount of processing time while the estimation quality might not increase.

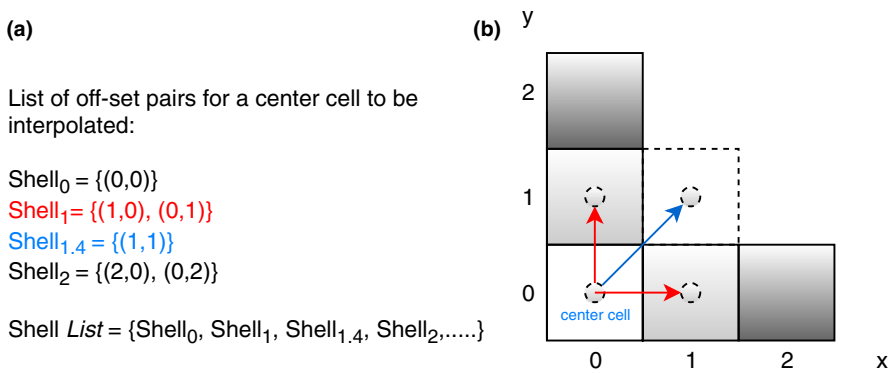


FIGURE 5 Shell list template

4.2 | *k*-Shell interpolation algorithm

As mentioned, the Shell algorithm leads to variable numbers of observations per cell since it uses a fixed search radius, and regions with sparse sampling will potentially result in lower estimation quality. To alleviate the drawbacks, we designed an alternative algorithm, the *k*-Shell approach. In *k*-Shell, a parameter *k* is chosen to determine a minimum number of observations that need to be identified until the search terminates. Here, the number of samples is less dependent on sampling around an estimated cell, and a more uniform estimation quality for the entire raster can be achieved.

In previous work (Nittel et al., 2012), we introduced an algorithm to find the *k*-nearest-neighbor (*k*NN) tuples for each cell. Here, the interpolation operator scans the tuples within and around the cell c_i, and incrementally creates a *sorted* list of *k*NN tuples. In detail, it scans all available tuples in an index cell, which are maintained in an unordered list. While processing the tuple list, it creates a *sorted* list in which tuples are ordered by distance from the cell center. Then, it determines which tuples are considered as *k*-nearest ones. While this method was exact when using the correct *k*NN sample, creating sorted lists requires significant computational cost. Furthermore, creating lists leads to increased memory consumption in the first place.

In this article, we propose a simplified, faster algorithm. Still, the search for samples is capped by the parameter *k*. The *k*-Shell interpolation operator starts with the shell $sh_0(c_i)$ and processes the tuples within the cell c_i (in the grid index). All tuples are processed, independently of how many tuples are actually present. If, after processing $sh_0(c_i)$, the parameter *k* is satisfied, the search terminates. Otherwise, the *k*-Shell interpolation operator expands the search to the next larger shell, $sh_1(c_i)$, using the SLT (see Section 3.3). Again, *all* tuples that can be found in the cells determined by $sh_1(c_i)$ are processed within the interpolation, and if *k* is satisfied, the search terminates, and so on. Not sorting tuples by distance, as well as not copying any data, eliminates processing steps and memory consumption, leading to significant improvement in run-time as well as stable memory consumption.

For regions that are sparsely sampled, such as observations collected on rural road networks, *k*-Shell can lead to a vast expansion of the search radius until *k* samples are found. The spatial distance of those found samples from cell ec_i might be so significant that the interpolation quality is decreased if they are considered. Therefore, we propose a third algorithm, the *ak*-Shell approach. This algorithm is identical to *k*-Shell, besides using a r_{max} parameter. In *ak*-Shell, the operator terminates if it has found *k* samples or if it has reached r_{max}, whichever comes first.

Algorithm 3 ***k*-Shell IDW Algorithm**

```
1: for all idwThread ∈ interpolationThreads do
2:     gridCell_ij ← bufferQueue.take();
3:     for all shell ∈ gridCell_ij.STL.list do
4:         for all shellListElem ∈ shell.list do
5:             for all t_i ∈ shellListElem.datalist do
6:                 numerator + = gWeight(t_i.getDistance(), power) * t_i.value;
7:                 denominator + = gWeight(t_i.getDistance(), power);
8:                 totalTupleCount + + ;
9:         if totalTupleCount > = k then
10:            gridCell_ij.value ← numerator/denominator;
11:            outputQueue.insert(⟨i, j, gridCell_ij.value⟩);
```

In summary, *k*-Shell has the advantage of dealing with data skew adaptively by using a minimum bound of samples to be considered and potentially a maximum search radius. Especially for densely sampled regions, the algorithm is adaptive to data skew in sub-regions. Furthermore, the processing time of *k*-Shell is significantly improved.

5 | EXPERIMENTAL SETUP

5.1 | Implementation and run-time environment

The stream-based algorithms were implemented in Java in a simulated DSE environment (i.e., operators are connected via queues, and work in a pipelined fashion), but we do not consider any other DSE components. The experiments were run on an HP Z440 workstation with a 3.5 GHz, six-core Intel Xeon E5-1650v3 processor and 32 GB of DDR3 1866 memory; the system run Ubuntu 14.04 and Java 1.8.0_131. The Java heap size was set at 3 GB, which was used to buffer the input stream and all processing.

5.2 | Data streams

Since sufficient numbers of live data streams for this type of research are not available today, we simulated streams by creating moving agents that simultaneously sense a common phenomenon over a geographic region with Netlogo (Wilensky, 1999). We created different population sizes of moving agents, since each agent is the originator of a sensor stream. The population sizes that we use for testing are the following: 2^{14}, 2^{15}, ..., 2^{19}. The sensor agents of each population move in different spatial patterns, as discussed in the next section.

In this article, we have used the 5 days after the Fukushima Dai-ichi nuclear accident as input for the simulation of moving sensor data streams. Calendar time corresponds to different "ticks" (t) in the agent simulation. For the experimental results in Section 6, we primarily used "tick" stamp t = 24, which corresponds to March 14, 2011 00:00 UTC, 9:00 JST (day 3 after the nuclear accident) and "tick" stamp t = 40 (March 16, 2011 00:00 UTC, 9:00 JST), since the phenomenon changes significantly during these days. Figure 6 shows the radiation deposits at t = 40.

5.2.1 | Movement patterns

The sensor agents move randomly and generate environmental sensor measurements along their paths. All agents sampled the phenomenon concurrently, and observations were logged. Initially, the agents were uniformly distributed over the entire area, but their movement was based on three different spatial movement constraint

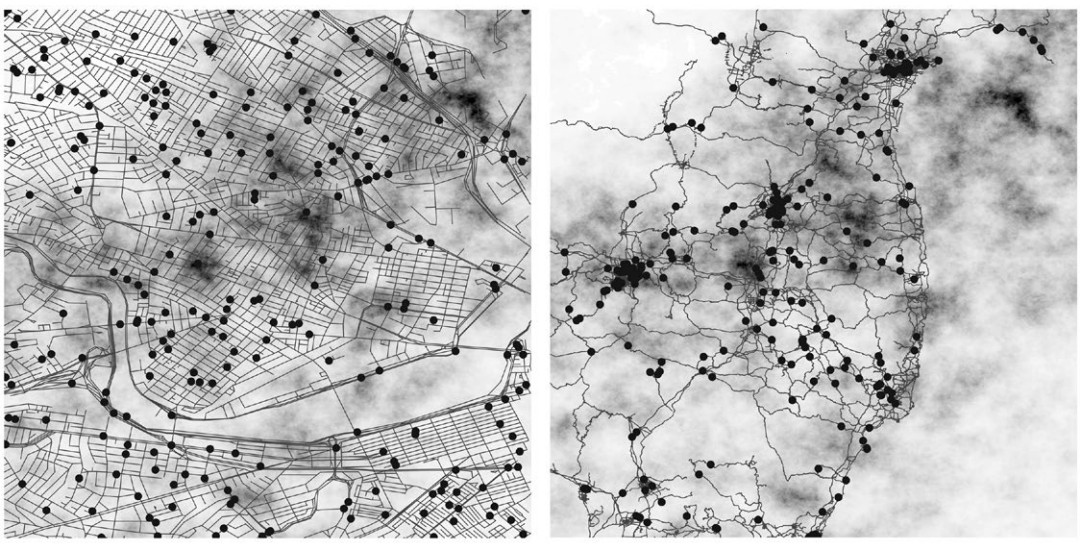


FIGURE 6 Radiation plume and simulated sensors over: (a) Cambridge, MA; and (b) Fukushima, Japan on March 16, 2011 00:00 UTC, 9:00 JST (t = 40)

models: (a) movement in a *dense* urban road network (the Cambridge road network data set [Cambridge data set]); (b) movement in a rural, *sparse* road network (the Japan road network data set [Japan data set]); and (c) spatially *unconstrained* random walk movements (the unconstrained randomly sampled data set [randomly sampled data set]) (see Figure 6).

5.2.2 | Radiation data set

We considered a realistic spatio-temporal phenomenon to be sampled by the moving agents. In detail, we used an approximation of the radiation deposits in the 5 days after the Fukushima Dai-ichi nuclear accident on March 11, 2011. The predicted radiation levels were calculated in R using data from ZAMG (Wotawa & Skomorowski, 2012) and SPEEDI (Chino, Ishikawa, & Yamazawa, 1993), and a sequence of snapshots of the event between the fourth and fifth day after the disaster in 15 min sampling increments was generated. For each snapshot a 2,000 × 2,000 cell raster for a 2° × 2° study area surrounding Fukushima was used.

The data set was created in the following way: soon after the start of the Fukushima Dai-ichi nuclear accident, the Meteorological and Geophysical Service of Austria (ZAMG) started computing and publishing data on the behavior of the radioactive cloud released by the accident (Wotawa & Skomorowski, 2012). Calculations relied on the model FLEXPART (Stohl, Forster, Frank, Seibert, & Wotawa, 2005). For this study, ZAMG kindly provided accumulated cesium-137 deposition data for the period March 11, 21:00 UTC till April 3, 21:00 UTC in 3 h intervals. These data are strongly predictive with respect to measured radiation levels, although locally varying weather conditions as well as radionuclides and other variables not accounted for by the model will cause departures from locally expected radiation levels.

Similar to Heuvelink, Jiang, De Bruin, and Twenhofel (2010), we represented true measured radiation levels as the sum of a deterministic trend based on the ZAMG data and a stochastic residual:

$$\log(Z(x)) = f[\log(Cs_{ZAMG}(x))] + \epsilon(x) \tag{2}$$

where $\log(\cdot)$ denotes a natural log transform, $Z(x)$ denotes the radiation measured at a location x, Cs_{ZAMG} is the modeled accumulated cesium-137 deposition, f is a (deterministic) regression function, and $\varepsilon(x)$ is the difference between the true and the deterministic radiation levels. Since the latter residual is unknown at non-measured locations, it was modeled as a second-order stationary and isotropic Gaussian random function (GRF), see Heuvelink et al. (2010). The regression coefficients as well as the parameters of the GRF were obtained by analyzing the Cs_{ZAMG} data in combination with measurements acquired by the System for Prediction of Environment Emergency Dose Information (SPEEDI), which are available via http://www.sendung.de/japan-radiation-open-data/, as well as data acquired by the global sensor network Safecast (http://blog.safecast.org/). All analyses were performed in R (The R Project for Statistical Computing, 2017) using the contributed packages gstat (Pebesma, 2004), sp (Pebesma, 2004), and akima, along with custom scripts.

6 | PERFORMANCE EVALUATION

We followed a two-step process for the performance tests: first, we tested different parameters (power parameter p for IDW, search radius r, k for k-Shell) and selected the configuration that produced the best estimation quality for the different algorithms, sampling characteristics strategies, and phenomena; second, we assessed the run-time and throughput performance for the selected parameters from step 1 and tested different sensor stream population sizes (population sizes).

6.1 | Investigating RMSE impact of chosen parameters

Each configuration that we tested generated predicted output rasters during interpolation. To assess the interpolation error of each configuration, we compared each cell in the estimated raster, *e*, with the corresponding cell in the ground truth raster, *o*, that was sampled by the simulated mobile sensor agents. Since the agents are sampling a phenomenon that changes over time, *e* must be compared with *o* from time stamp *t*. The RMSE was computed as follows:

$$\mathrm{RMSE} = \sqrt{\frac{\sum_{i=1}^{n} (e_i - o_{t,i})^2}{n}} \tag{3}$$

The RMSE was normalized by dividing the RMSE by the mean of o_t to allow the RMSE for snapshots from different times to be compared with each other:

$$\mathrm{NRMSE} = \frac{\mathrm{RMSE}}{\overline{o_t}} \tag{4}$$

Two of the data sets contain water bodies (the Sea of Japan and the Pacific Ocean for the Japan data set; the Charles River for the Cambridge data set); our simulation did not sample over water bodies and any estimated cells over water bodies depend on the sampling over land. We used two different methods to assess RMSE: in Method 1, we compared only cells for which a value was actually calculated and *empty cells were ignored* in the RMSE calculation; in Method 2, the empty cells from Method 1 were *replaced with the mean value* of all predicated cells, and RMSE was computed over all cells. We normalized RMSE for the different data sets by dividing the RMSE by the mean of the ground truth images at the same time stamp. We used both methods; Method 1 is used for the random sampling data set, and Method 2 for the Japan and Cambridge data sets.

We determined the kriging baselines as best case RMSE for the different data sets. We determined it by using the largest population (256k), the same time stamps as in the tests ($t = 24$ and $t = 40$), and a local kriging configuration with 32 neighbors. The kriging baselines are the following: (1) for the randomly sampled data set the RMSE is 0.1 for $t = 24$ and 0.08 for $t = 40$; (2) for the Cambridge data set the RMSE is 0.27 for $t = 24$ and 0.18 for $t = 40$; and (3) for the Japan data set the RMSE is 0.15 for $t = 24$ and 0.24 for $t = 40$.

6.2 | Identifying the power parameter *p* for IDW

IDW assumes that the variable being assessed decreases in influence with distance from its sampled location. A higher power gives more emphasis to nearer points, and significantly lower weight to more distant samples. The type of sampling we test is different from traditional IDW scenarios, that is, using sensor data streams in high spatial density, IDW encounters a much higher number of observations in close proximity; a high power parameter *p* is expected to perform better, since many samples are already likely in spatial proximity and should receive higher emphasis during calculations.

In this test, we assessed the influence of the power parameter *p* for both the Shell and *k*-Shell algorithms for the Cambridge data set. Four different population sizes, $n = 32k, 64k, 128k, 256k$, were selected. For this test, the radius was set to $r = 32$ and we chose the radiation phenomena at $t = 24$ (March 14, 2011 00:00 UTC, 9:00 JST) and $t = 40$ (March 16, 2011 00:00 UTC, 9:00 JST). For each population size, we varied the power value between 1 and 4. Figure 7 shows that for the Shell approach, the power $p = 4$ has the lowest RMSE. For the *k*-Shell approach ($k = 4$), the power had limited impact; this is due to the fact that likely *k* samples are found in close proximity and the search terminates.

6.3 | Impact of selected algorithm parameters

Next, we investigated the influence of several parameters on the interpolation *quality* of the algorithms described in Section 4.

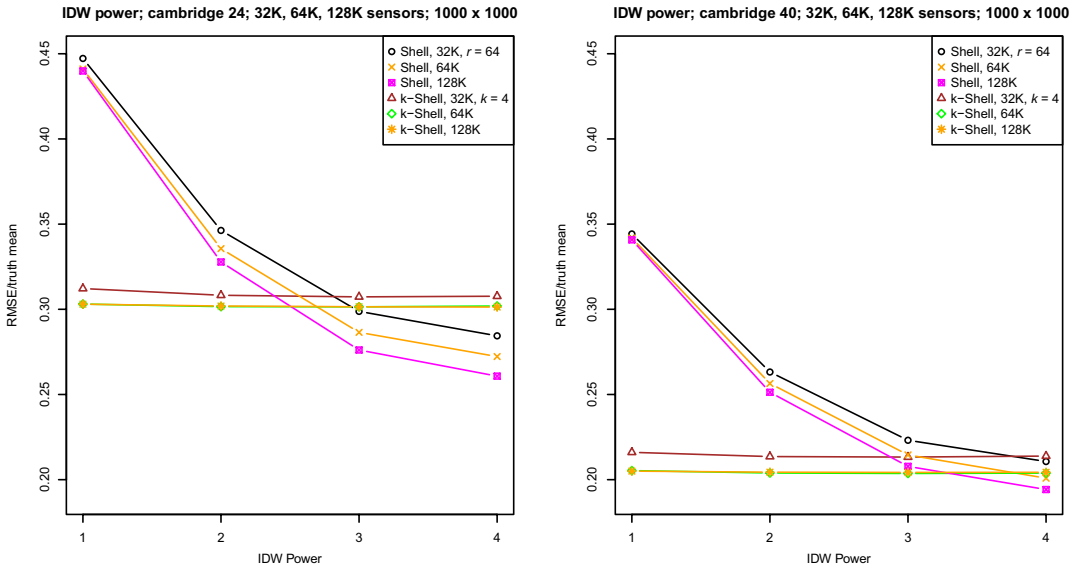


FIGURE 7 Impact of power parameter *p* on RMSE

6.3.1 | Impact of the neighborhood radius

We explored the impact of the search radius *r* around a cell. Each algorithm starts by locating samples in an incrementally increasing search radius around the candidate cell. How far should we extend the radius to cap the number of samples when considering the estimation of a cell? We tested the following radii: 0, 1, 2, 4, 8, 16, 32, 48, 64, 80. Further, we tested the population sizes *n* = 32K, 64K, 128K, 256K and the grid size is 1,000 × 1,000.

The results for the Shell approach (*t* = 24) are depicted in Figure 8. As expected, the RMSE decreases with increasing search radius. For the randomly sampled data set, *r* = 4 is the radius size that can be considered as delivering an acceptable RMSE (and larger radii, naturally). For the Cambridge data set, radius sizes starting with *r* = 32 start producing results with the lowest RMSE. In the Japan data set, acceptable results are also achieved with *r* = 32, since the RMSE does not decrease significantly after increasing the radius further. Results are similar for *t* = 40, but here a starting radius of *r* = 16 was set for the Cambridge data set, since this is a data set that is sampled spatially denser and therefore a smaller radius is sufficient.

In the next test, we compared the impact of the power parameter *p* versus the radius *r* for sensor population size *n* = 128K. In Figure 9, we can see that starting with *r* = 4 and *p* = 4, the lowest RMSE is achieved for the randomly sampled data set. On the other hand, using *p* = 1 leads to increasingly distorted results with increasing radius. The other tests confirm the assessment of *r* from above.

6.3.2 | Impact of *k* for the *k*-Shell approach

In the *k*-Shell approach, we investigated the impact of the parameter *k* (Figure 10). *k* was varied as follows: *k* = 1, 2, 4, 6, 8, 16, 32, *r* = 64, and the grid size is 1,000 × 1,000. It is expected that the RMSE decreases with larger *k*, which is the case in all tests. For the randomly sampled data set, *k* = 4 delivers a stable RMSE, and increasing *k* further does not decrease the RMSE significantly. For the Cambridge data set and the Japan data set, the parameter *k* = 8 using *k*-nearest-neighboring samples delivers a RMSE that is not significantly improved by using a higher *k*.

The motivation for *k*-Shell is that the memory footprint stays uniform for interpolation and thus is robust against skewed sampling patterns. In a second test, we compared power *p* versus *k* for *k*-Shell (Figure 11). This test confirms the results for *k* from above and using *p* = 4. We chose *k* = 4, 8, 16 as the basis for run-time tests.

6.3.3 | Impact of r_{max} and k for the *ak*-Shell algorithm

The *ak*-Shell algorithm is designed to perform well in sparse or non-urban road network sampling scenarios and it limits the search radius to a fixed r_{max} so that less computational time is spent in sparse areas. We varied k and tested different cutoff radii r_{max} = 8, 16, 32. We also varied the population size from 32K to 256K. The results show (as expected) that if the cutoff radius r_{max} and k are increased, the RMSE decreases. However, the influence of r_{max} is significantly higher than that of the chosen k (Figure 12). We chose r_{max} = 32 for the run-time tests.

Having determined the selected parameter for an acceptable RMSE, we performed run-time and throughput tests.

6.4 | Assessing run-time performance

6.4.1 | Impact of r on run-time

Figure 13 shows the run-time measurements for the Shell approach. We tested population sizes of n = 32K, 64K, 128K, 256K and radii r = 8, 16, 24, 32, 40; the grid size is 500 × 500. As is to be expected, the run-time of this algorithm significantly increases with larger r and population size, since massively more samples are processed per grid cell. For r = 40 and sample size n = 256K, a raster is created in ca. 1 min. As we assessed in Section 6.3.1, r = 16 is sufficient for the Cambridge data set, which can be computed for sample size n = 256K in around 10 s. Nevertheless, the Shell approach is not efficient for real-time processing as defined previously, in that it should produce a raster within 1–2 s.

Next, we tested the *k*-Shell approach, which is adaptive to sample distributions and skew, but is not less affected by population size.

6.4.2 | Impact of k on run-time

In the *k*-Shell approach, we investigated the impact of the k parameter on the run-time performance (shown in Figures 14 and 15) since k determines how many samples are used for estimation. We tested population sizes n = 32K, 64K, ..., 256K and k = 4, 6, 8, 16; the grid size is 500 × 500. This test showed that the *k*-Shell approach has

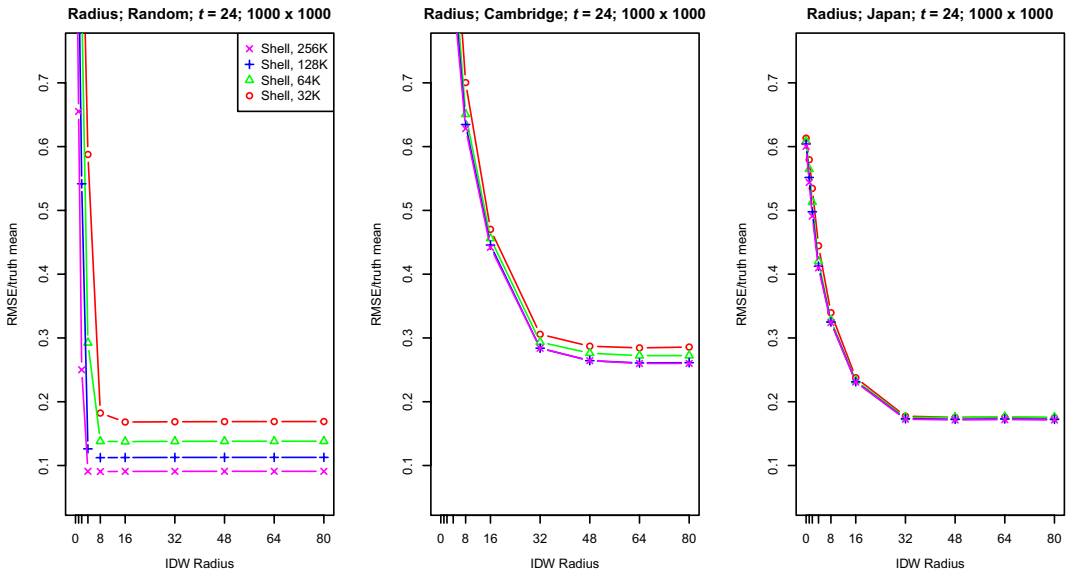


FIGURE 8 Impact of radius r on RMSE for the Shell algorithm (t = 24)

the best run-time performance of the algorithms proposed, and consistently reaches real-time computation. Even for the largest sample size of n = 256K and k = 16, the k-Shell approach runs in under 0.5 s. Previously, we had determined that for the data set at hand, k = 8 provides sufficiently low RMSE. Note that the run-time performance for n = 32K is higher than that for n = 64K; this is due to the fact that n = 32K is a sparser sample set, and thus the cell interpolators need to search in a wider radius (i.e., more cells) to find sufficient k samples.

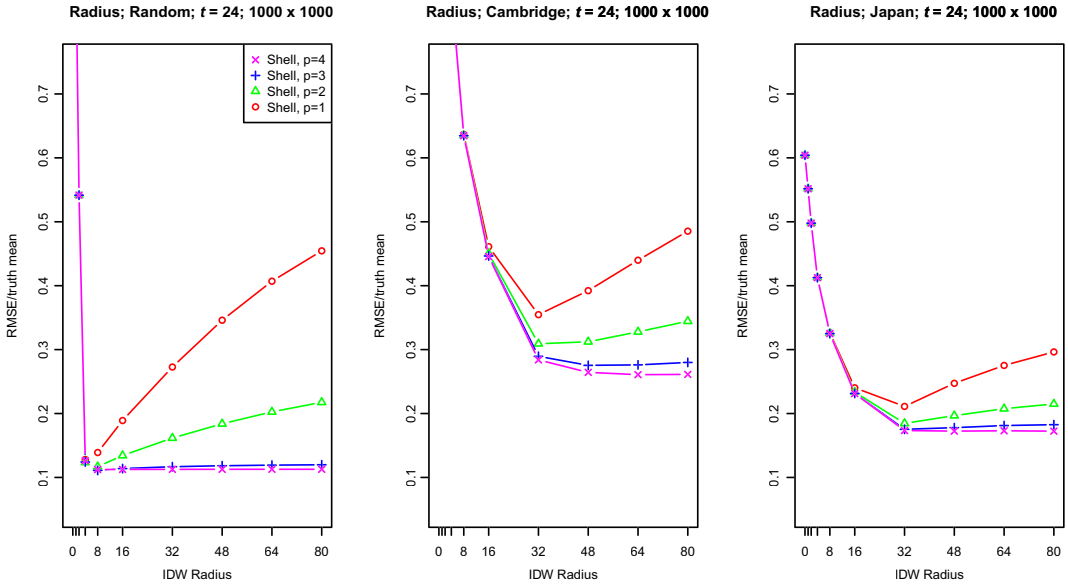


FIGURE 9 Radius vs. power for the Shell algorithm (t = 24)

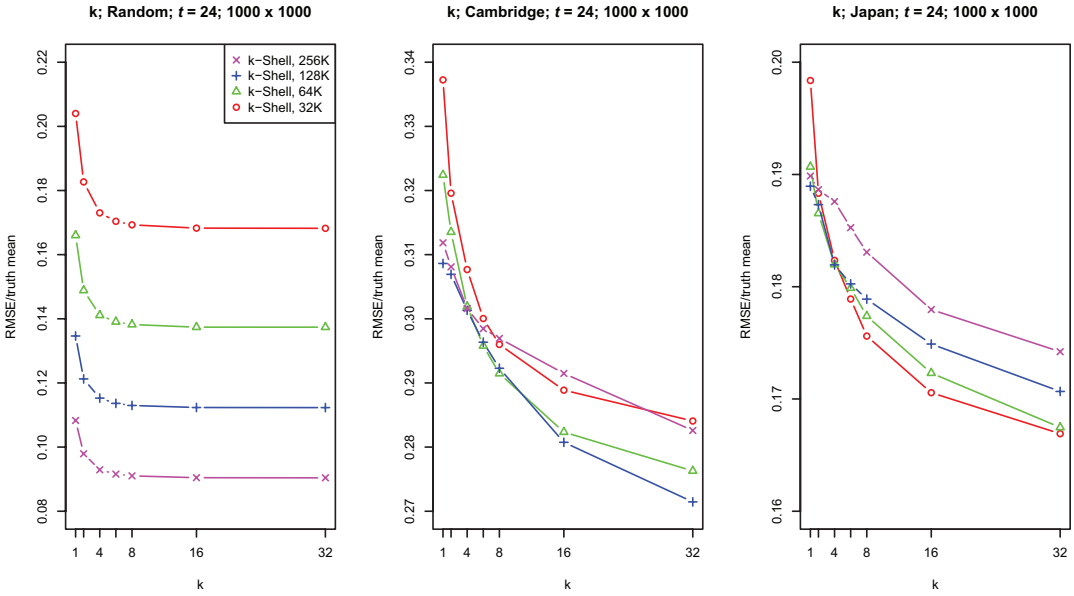


FIGURE 10 Impact of parameter k for the k-Shell algorithm (t = 24)

6.4.3 | Summary of Run-time Evaluations

Summarizing the run-time tests of the proposed algorithms, we assess that *k*-Shell and *ak*-Shell perform significantly faster that Shell for random sample distribution and dense metropolitan networks (see Figure 16 and Figure 17). The tests show that *k*-Shell and *ak*-Shell interpolate 250k observations in around 0.3 s, which fulfills the real-time performance criteria. For sparser, rural street networks, *ak*-Shell outperforms the other two algorithms, taking around 1 s for 250k samples, while *k*-Shell takes 5 s for the same configuration. Thus, for areas with sparse, irregular sampling capping the search radius is essential to keep run-time within real-time requirements.

6.5 | PERFORMANCE TEST SUMMARY AND DISCUSSION

In the last test, we compared run-time versus RMSE for the Shell, *k*-Shell and *ak*-Shell approaches (see Figure 18). The population sizes tested are n = 32K, 64K, 128K, 256K and the grid is 500 × 500.

We tested the three sampling distributions as mentioned before. In the randomly sampled data set, the Shell algorithm with r = 16 reduced the RMSE from 0.15 to 0.08 while the run-time increased from 1.5 to 10 s from the smallest to the largest data set. This algorithm depends more on the population size with regard to run-time performance, and the run-time increases disproportionately to the reduction in prediction error. For the *k*-Shell approach, the RMSE is in the same range as for the Shell algorithm, while the run-time for *k*-Shell is consistently around 0.25 s, even for the largest data sets. Thus, the *k*-Shell approach does deliver real-time performance, even for massive data sets, with an almost stable real-time run-time performance and comparable RMSE as methods that use more sample points for interpolation for random sampling cases. For comparison, the kriging baseline using all samples and a local neighborhood of 32 elements has a RMSE of 0.1 (with a run-time of 5 h).

Further, we tested the two sampling scenarios that are based on *dense* and *sparse* road networks. In the Cambridge data set, the *k*-Shell algorithm produces a RMSE between 0.295 and 0.28 with increasingly larger data sets, while the run-time is constant around 0.25 s. On the other hand, the Shell approach reduces the RMSE from

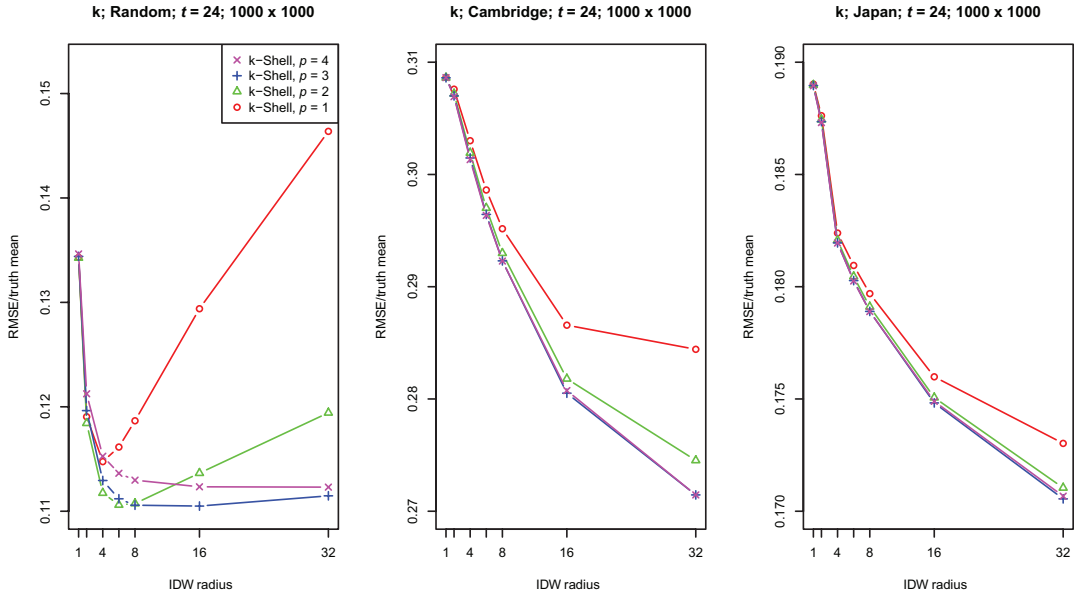


FIGURE 11 *k* Parameter vs. power parameters for the *k*-Shell algorithm (t = 24)

0.29 to 0.27 for r = 16 with increasingly larger data sets, and from 0.27 to 0.25 for r = 32, while the run-time increases to 25 s for the larger data sets. Thus, the run-time for the Shell approach does increase with data set size. Note that for this data set the RMSE is not further improved with n = 128K or n = 256K, that is, sufficient samples are available with n = 128K. Still, the k-Shell algorithm still provides real-time run-time performance and produces a RMSE of 0.28 compared to a value of 0.25 delivered by the Shell approach. For comparison, the kriging baseline RMSE is 0.27.

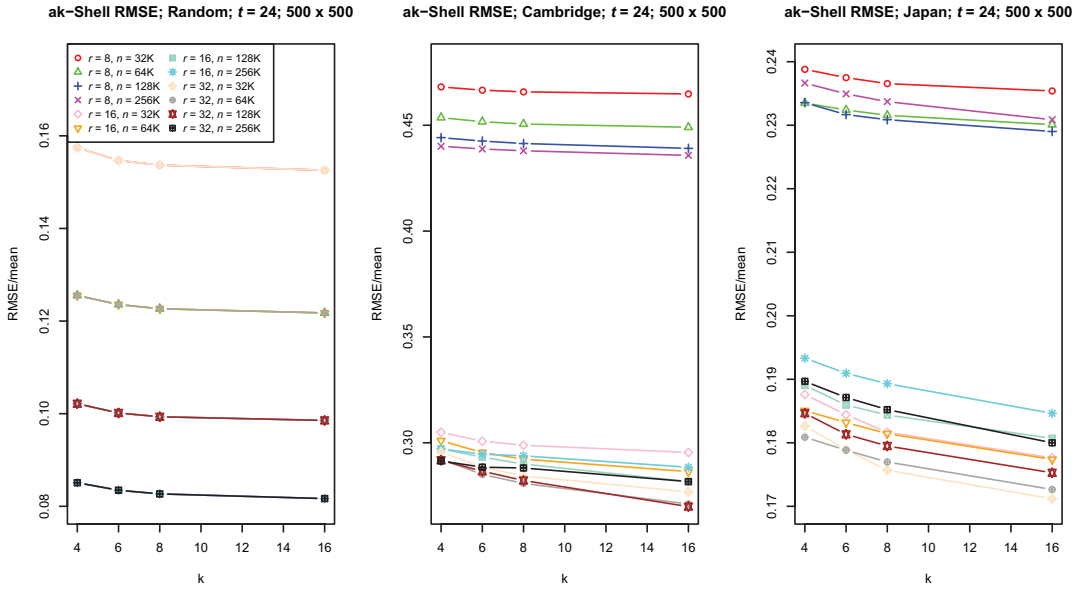


FIGURE 12 Impact of r_{max} on the ak-Shell algorithm (t = 24)

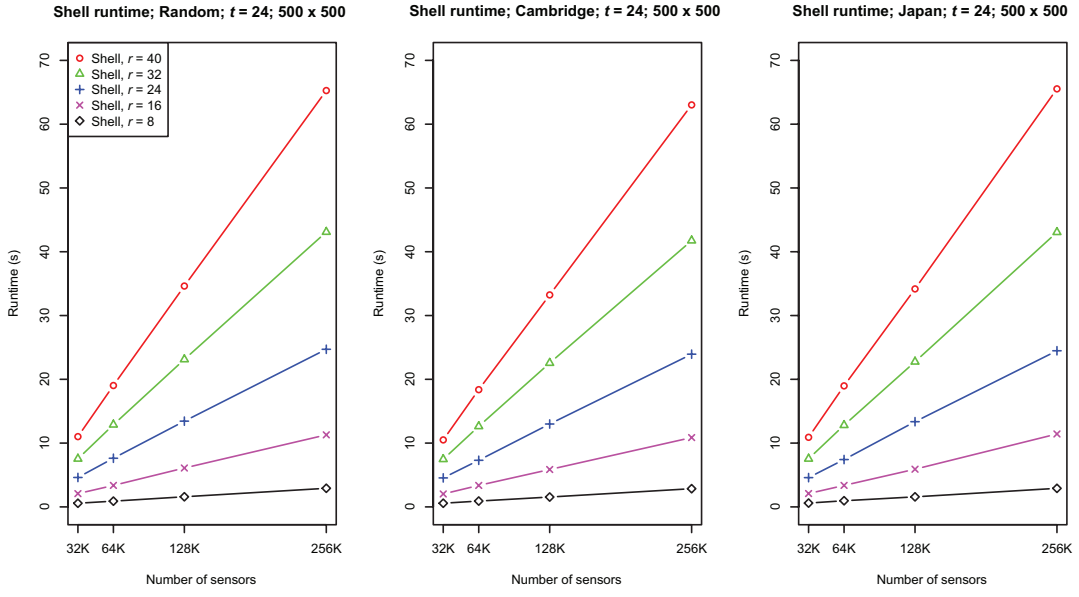


FIGURE 13 Run-time for the Shell algorithm (t = 24)

In the Japan data set, the *ak*-Shell approach has the shortest run-time (around 1 s) and the RMSE is between 0.175 and 0.19 for increasingly larger data sets. The *k*-Shell algorithm produces a RMSE between 0.185 and 0.175, while the run-time is constant around 2.5 s. This test case shows the necessity of a stop criterion for the search radius (i.e., r_{max}) to provide for real-time performance (the kriging baseline RMSE is 0.15). The Shell algorithm has the smallest RMSE of the tested algorithms (1.68 for r = 16), while the run-time increases to 10 s.

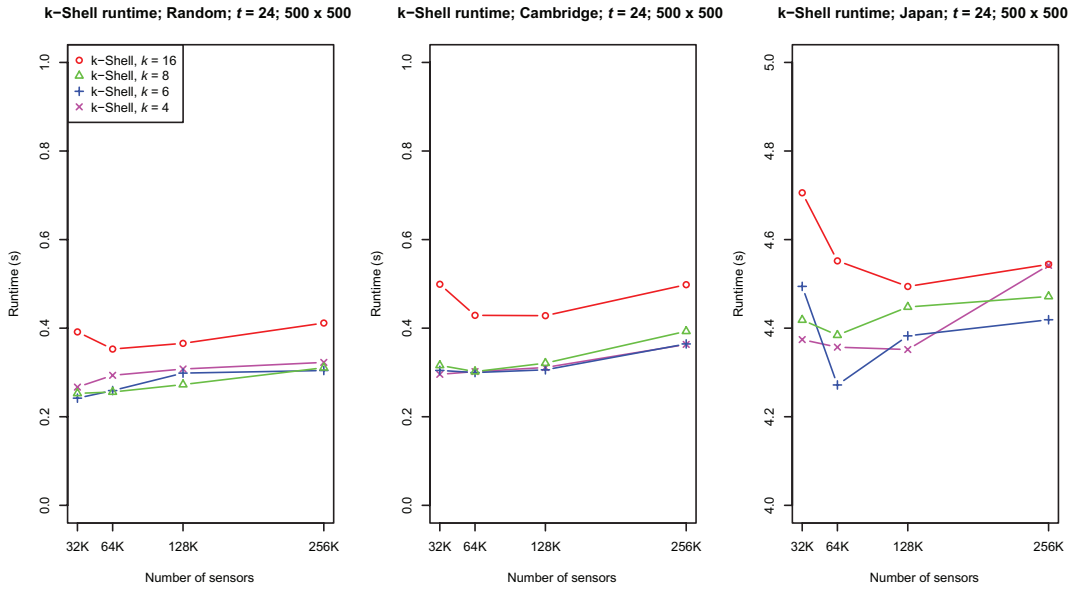


FIGURE 14 Run-time for the *k*-Shell algorithm (t = 24)

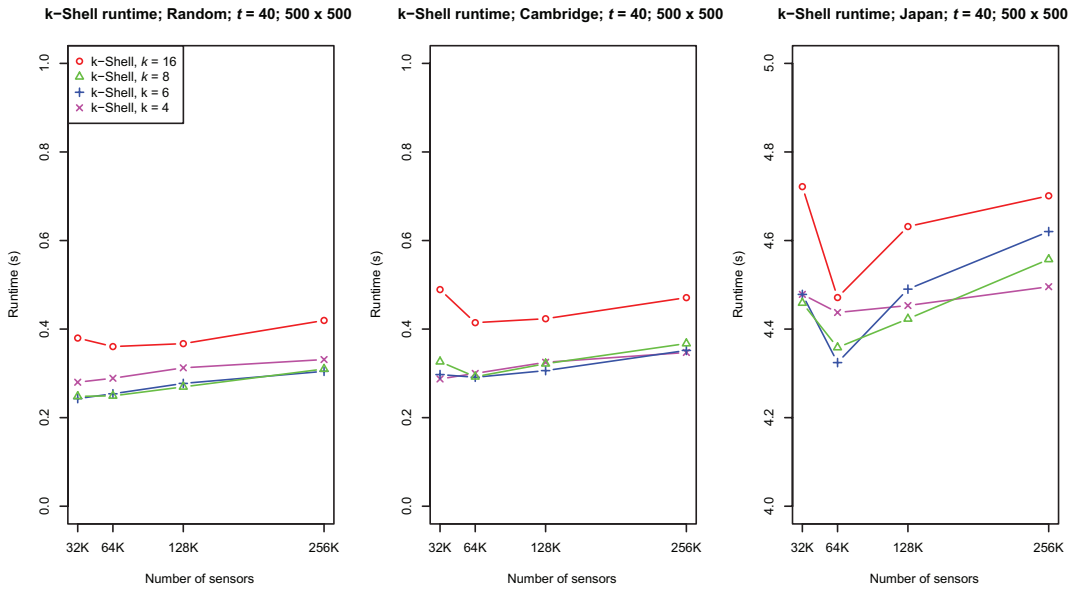


FIGURE 15 Run-time for the *k*-Shell algorithm (t = 40)

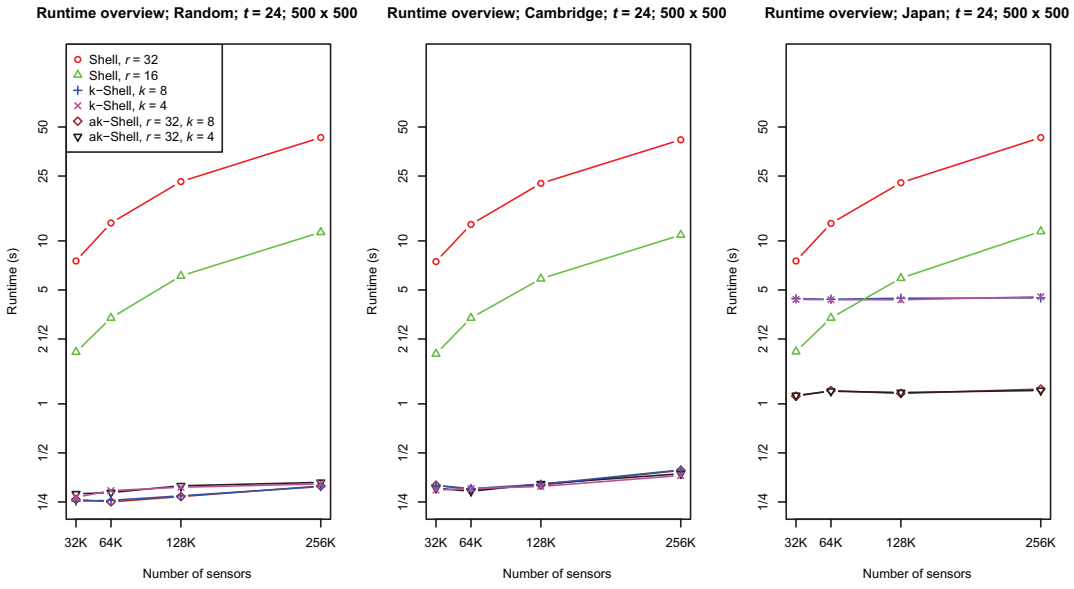


FIGURE 16 Run-time comparison for Shell, *k*-Shell, and *ak*-Shell (*t* = 24)

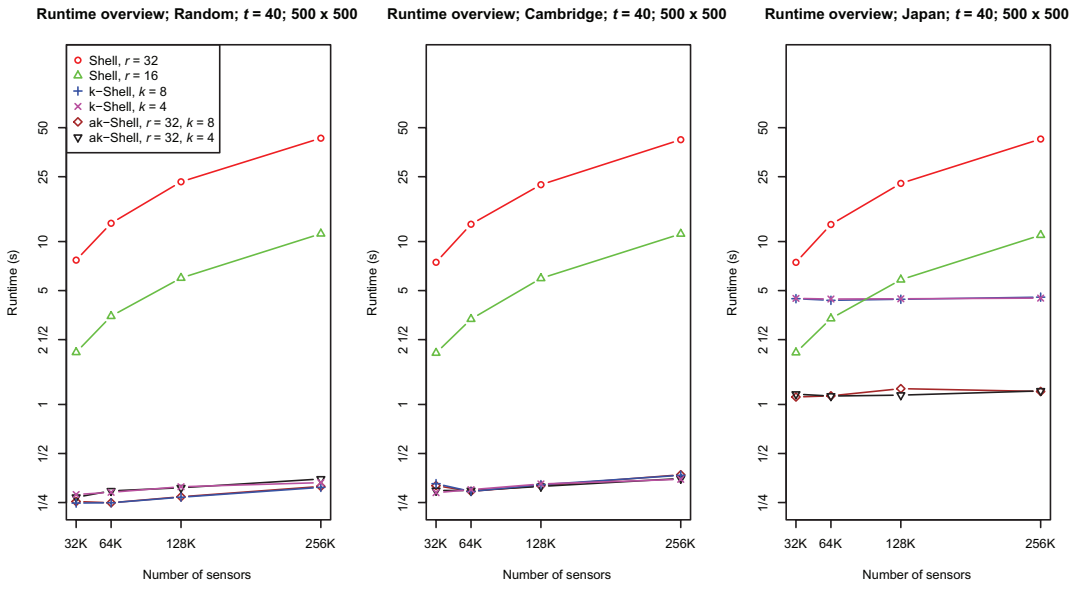


FIGURE 17 Run-time comparison for Shell, *k*-Shell, and *ak*-Shell (*t* = 40)

7 | CONCLUSIONS AND FUTURE WORK

With advances in technology, environmental monitoring has become increasingly sensor dense and real time. Sensor data streams enable real-time applications such as environmental hazard detection, or earthquake, wildfire, or radiation monitoring, but in-depth analysis of such spatial fields is often based on *continuous* representations.

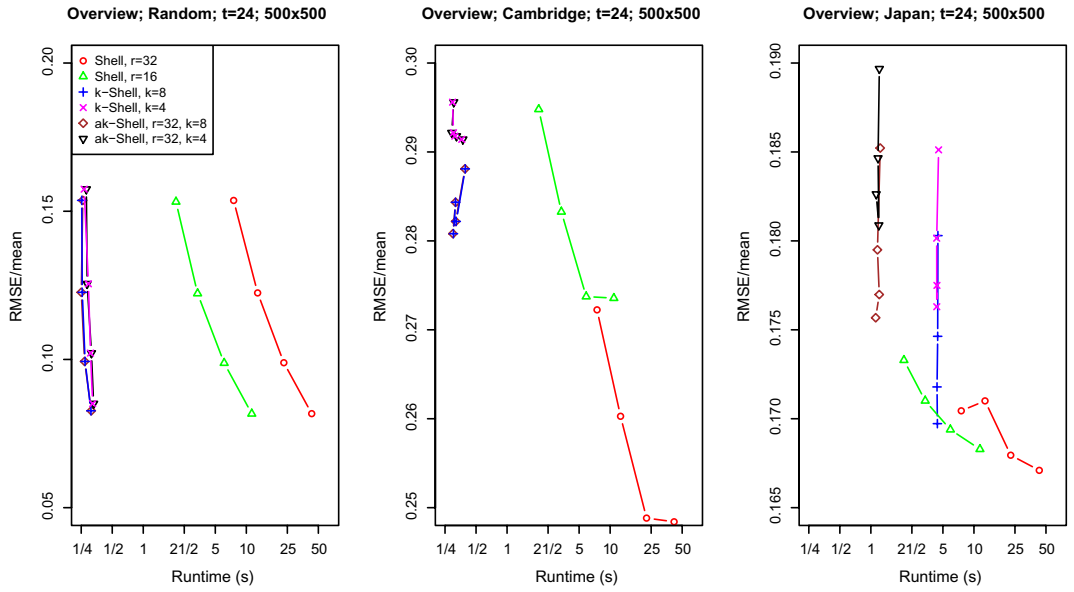


FIGURE 18 RMSE vs. run-time for Shell, *k*-Shell, and *ak*-Shell (t = 24)

In this article, we presented an approach leveraging data stream engines to achieve scalable, high-throughput inverse distance weighting interpolation. This stream operator framework continuously generates representations of spatial continuous phenomena in near real time. In detail, we introduced a novel stream query operator framework that extends general-purpose data stream engines. The proposed framework includes a two-panel, spatio-temporal grid-based index and the Shell, *k*-Shell and *ak*-Shell algorithms to estimate individual grid cells efficiently and adaptively to sampling skew. For our performance tests, we generated several different spatio-temporal stream data sets based on the radiation deposits in the Fukushima region after the nuclear accident of March, 2011 in Japan. We took different sampling scenarios into account (such as dense metropolitan road networks, sparse rural networks, and random sampling). For our performance experiments, we first investigated parameters such as search radius, power, and k parameters that achieved the lowest RMSE for the estimated fields. Following on, we tested the throughput and run-time performance of the proposed algorithms using these parameters. Our results showed that the *k*-Shell algorithm of the proposed framework continuously produces rasters based on 250k update/s in under 0.5 s using a state-of-the-art workstation.

While this article focused on the questions of throughput and run-time performance, another element of continuous stream processing is the automatically adaptive nature of a stream query framework. For instance, in this article we determined parameter settings for IDW such as p, r, and k experimentally. An automated system with queries that run for several weeks might experience significant changes in sampling patterns of moving sensors and also phenomena changes, so that parameters might need to change over the run-time of the query. Future research is required to explore concepts of how to determine and adapt such parameters during stream query execution on-the-fly and automatically.

8 | ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback. The authors also thank Balaji Venkatesan and Anthony Peterson for support with performance testing. The research in this article was supported by the National Science Foundation through Grant No. 1527504.

NOTES

[1]Sensors might also sample asynchronously, but this case is not considered in this article.

[2]The great circle distance between a cell center $ic_{x,y}$ and the relevant tuples x_j is calculated using the Haversine function.

REFERENCES

Agrible, Inc. (2016). *Agrible.* Retrieved from http://www.agrible.com

Ali, M., Chandramouli, B., Raman, B. S., & Katibah, E. (2010). Spatio-temporal stream processing in Microsoft StreamInsight. In *Proceedings of the 26th IEEE International Conference on Data Engineering* (pp. 542–543). Long Beach, CA: IEEE.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink™: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, *36*(4), 28–38.

Chino, M., Ishikawa, H., & Yamazawa, H. (1993). SPEEDI and WSPEEDI: Japanese emergency response systems to predict radiological impacts in local and workplace areas due to a nuclear accident. *Mathematics & Physical Sciences Radiation Protection Dosimetry*, *50*(2), 145–152.

Faulkner, M., Olson, M., Chandy, R., Krause, J., Chandy, K., & Krause, A. (2011). The next big one: Detecting earthquakes and other rare events from community-based sensors. In *Proceedings of the 10th International Conference on Information Processing in Sensor Networks* (pp. 13–24). Chicago, IL: IEEE.

Galic, Z., Baranovič, M., Krízanovič, K., & Méskovič, E. (2014). Geospatial data streams: Formal framework and implementation. *Data & Knowledge Engineering*, *91*, 1–16.

Guan, Q., Kyriakidis, P. C., & Goodchild, M. F. (2011). A parallel computing approach to fast geostatistical areal interpolation. *International Journal of Geographical Information Science*, *25*(8), 1241–1267.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 47–57). Boston, MA: ACM.

Hennebohl, K., Appel, M., & Pebesma, E. (2011). Spatial interpolation in massively parallel computing environments. In *Proceedings of the 14th AGILE International Conference on Geographic Information Science*. Utrecht, The Netherlands: AGILE.

Heuvelink, G. B. M., Jiang, Z., De Bruin, S., & Twenhofel, C. J. W. (2010). Optimization of mobile radioactivity monitoring networks. *International Journal of Geographical Information Science*, *24*(3), 365–382.

Huang, F., Bu, S., Tao, J., & Tan, X. (2016). OpenCL implementation of a parallel universal kriging algorithm for massive spatial data interpolation on heterogeneous systems. *ISPRS International Journal of Geo-Information*, *5*(6), 96.

Hudnut, K. W., Bock, Y., Galetzka, J. E., Webb, F. H., & Young, W. H. (2002). The Southern California Integrated GPS Network (SCIGN). In Y. Fujinawa, & A. Yoshida (Eds.), *Seismotectonics in convergent plate boundary* (pp. 167–189). Tokyo, Japan: TerraPub.

Kazemitabar, S., Demiryurek, U., Ali, M., Akdogan, A., & Shahabi, C. (2010). Geospatial stream query processing using Microsoft SQL Server StreamInsight. *Proceedings of the VLDB Endowment*, *3*(1&2), 1537–1540.

Kong, Q., Kwony, Y. W., Schreierz, L., Allen, S., Allen, R., & Strauss, J. (2015). Smartphone-based networks for earthquake detection. In *Proceedings of the 15th International Conference on Innovations for Community Services* (pp. 1–8). Nuremberg, Germany: IEEE.

Krige, D. G. (1951). *A statistical approach to some mine valuation and allied problems on the Witwatersrand* (Unpublished Ph.D. Dissertation). University of the Witwatersrand, Johannesburg, South Africa.

Lam, N. S. N. (1983). Spatial interpolation methods: A review. *Cartography and Geographic Information Science*, *10*(2), 129–150.

Liang, Q., Nittel, S., & Hahmann, T. (2016). From data streams to fields: Extending stream data models with field data types. In J. A. Miller, D. O'Sullivan, & N. Wiegand (Eds.), *Geographic Information Science: 9th International Conference, GIScience 2016, Montreal, QC, Canada, September 27–30, 2016, Proceedings* (Lecture Notes in Computer Science, Vol. 9927, pp. 178–194). Cham, Switzerland: Springer.

Lorkowski, P., & Brinkhoff, T. (2015a). Environmental monitoring of continuous phenomena by sensor data streams: A system approach based on kriging. In *Proceedings of EnviroInfo and Third International Conference on ICT for Sustainability*. Copenhagen, Denmark.

Lorkowski, P., & Brinkhoff, T. (2015b). Towards real-time processing of massive spatio-temporally distributed sensor data: A sequential strategy based on kriging. In F. Bacao, Y. M. Santos, & M. Painho (Eds.), *AGILE 2015: Geographic information science as an enabler of smarter cities and communities (Lecture Notes in Geoinformation and Cartography* (pp. 145–163). Cham, Switzerland: Springer.

Mead, M., Popoola, O., Stewart, G., Landshoff, P., Calleja, M., Hayes, M., …, &Jones, R. (2013). The use of electrochemical sensors for monitoring urban air quality in low-cost, high-density networks. *Atmospheric Environment*, *70*, 186–203.

Mei, G., Xu, N., & Xu, L. (2016). Improving GPU-accelerated adaptive IDW interpolation algorithm using fast kNN search. *SpringerPlus*, *5*(1), 1389.

Miller, J., Raymond, M., Archer, J., Adem, S., Hansel, L., Luti, M., ..., & Ali, M. (2011). An extensibility approach for spatio-temporal stream processing using Microsoft StreamInsight. In D. Pfoser, Y. Tao, K. Mouratidis, M. Nascimento, M. F. Mokbel, S. Shekhar, & Y. Huang (Eds.), *Advances in Spatial and Temporal Databases: 12th International Symposium, SSTD 2011, Minneapolis, MN, USA, August 24–26, 2011, Proceedings* (pp. 496–501). Berlin, Germany: Springer.

Mitas, L., & Mitasova, H. (1999). Spatial interpolation. In P. Longley, M. F. Goodchild, D. J. Maguire, & D. W. Rhind (Eds.), *Geographical information systems: Principles, techniques, management and applications*, 2nd ed. (pp. 481–492). New York, NY: Wiley.

Murty, R. N., Mainland, G., Rose, I., Chowdhury, A. R., Gosain, A., Bers, J., & Welsh, M. (2008). CitySense: An urban-scale wireless sensor network and testbed. In *Proceedings of the 2008 IEEE Conference on Technologies for Homeland Security* (pp. 583–588). Waltham, MA: IEEE.

Nittel, S. (2009). A survey of geosensor networks: Advances in dynamic environmental monitoring. *Sensors*, *9*(7), 5664–5678.

Nittel, S. (2015). *Real-time sensor data streams. SIGSPATIAL Newsletter*, *7*(2), 22–28.

Nittel, S., Whittier, J. C., & Liang, Q. (2012). Real-time spatial interpolation of continuous phenomena using mobile sensor data streams. In *Proceedings of the 20th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (pp. 530–533). Redondo Beach, CA: ACM.

Pebesma, E. J. (2004). Multivariable geostatistics in S: The gstat package. *Computers & Geosciences*, *30*(7), 683–691.

Resch, B., Mittlboeck, M., Girardin, F., Britter, R., & Ratti, C. (2009). Real-time geo-awareness sensor data integration for environmental monitoring in the city. In *Proceedings of the 2009 International Conference on Advanced Geographic Information Systems and Web Services* (pp. 92–97). Cancun, Mexico: IEEE.

Safecast (2016). *Safecast*. Retrieved from https://blog.safecast.org

Sanchez, L., Galache, J., Gutierrez, V., Hernandez, J. M., Bernat, J., Gluhak, A., & Garcia, T. (2011). SmartSantander: The meeting point between future Internet research and experimentation and the smart cities. In *Proceedings of the 2011 Future Network and Mobile Summit*. Warsaw, Poland.

Shepard, D. (1968). A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 23rd ACM National Conference* (pp. 517–524). Las Vegas, NV: ACM.

Srinivasan, B. V., Duraiswami, R., & Murtugudde, R. (2010). Efficient kriging for real-time spatio-temporal interpolation. In *Proceedings of the 20th Conference on Probability and Statistics in Atmospheric Sciences* (pp. 228–235). Atlanta, GA: AMS.

Stohl, A., Forster, C., Frank, A., Seibert, P., & Wotawa, G. (2005). The Lagrangian particle dispersion model FLEXPART version 6.2. *Atmospheric Chemistry & Physics*, *5*, 2461–2474.

Stonebraker, M., Çetintemel, U., & Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, *34*(4), 42–47.

The R Project for Statistical Computing (2017). *The R project for statistical computing*. Retrieved from https://www.r-project.org

Venkatesan, B. (2013). *Feasibility study of continuous real-time spatial interpolation of phenomena using built-in functionality of a commercial data stream management system* (Technical Report). Orono, ME: University of Maine.

Whittier, J., Nittel, S., Liang, Q., & Plummer, M. (2013). Towards window stream queries over continuous phenomena. In *Proceedings of the Fourth International ACM SIGSPATIAL Workshop on GeoStreaming* (pp. 1–10). Orlando, FL: ACM.

Whittier, J., Nittel, S., & Subasinghe, I. (2017). Real-time earthquake monitoring with spatio- temporal fields. In *Proceedings of the International Symposium on Spatiotemporal Computing*. Cambridge, MA.

Wilensky, U. (1999). *NetLogo: Center for Connected Learning and Computer-Based Modeling*. Retrieved from http://ccl.northwestern.edu/netlogo/

Wotawa, G., & Skomorowski, P. (2012). Long-range transport of particulate radionuclides from the Fukushima NPP accident: Sensitivity analysis for wet deposition. In *Proceedings of the EGU General Assembly* (p. 10494). Vienna, Austria.

Xiao, C., Chen, N., Gong, J., Wang, W., Hu, C., & Chen, Z. (2017). Event-driven distributed information resource-focusing service for emergency response in smart city with cyber-physical infrastructures. *ISPRS International Journal of Geo-Information*, *6*(8), 251.

Zhong, X., Kealy, A., & Duckham, M. (2016). Stream kriging: Incremental and recursive ordinary kriging over spatiotemporal data streams. *Computers & Geosciences*, *90*, 134–143.

Zhong, X., Kealy, A., Sharon, G., & Duckham, M. (2015). Spatial interpolation of streaming geosensor network data in the RISER system. In J. Gensel, & M. Tomko (Eds.), *Web and wireless geographical information systems: 14th International Symposium, W2GIS 2015, Grenoble, France, May 21–22, 2015, Proceedings* (Lecture Notes in Computer Science, Vol. 9080, pp. 161–177). Berlin, Germany: Springer.

Zhou, L., Chen, N., & Chen, Z. (2017). Efficient streaming mass spatio-temporal vehicle data access in urban sensor networks based on Apache Storm. *Sensors*, *17*(4), e815.

How to cite this article: Liang Q, Nittel S, Whittier JC, de Bruin S. Real-time inverse distance weighting interpolation for streaming sensor data. *Transactions in GIS*. 2018;00:1–26. https://doi.org/10.1111/tgis.12458