FPCS: Feature Preserving Compensated Sampling of Streaming Time Series Data

Hongyan Li, Bo Yang, and Yansong Chua

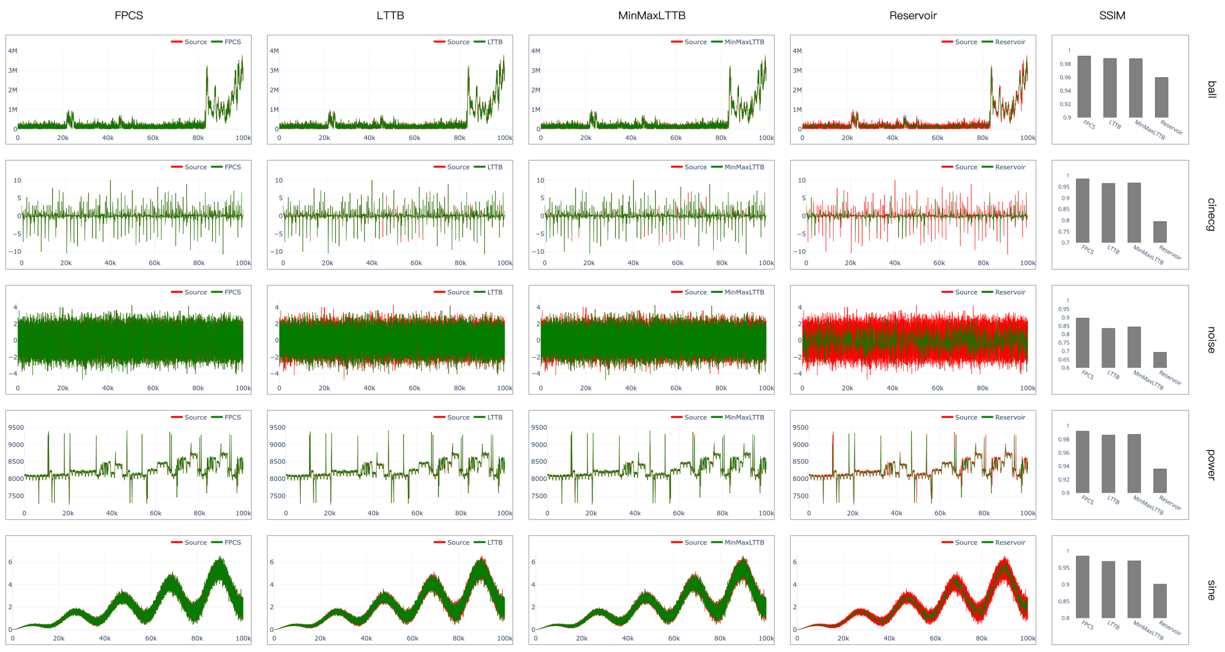

Fig. 1: Each row corresponds to one of the five typical datasets: ball, cinecg, noise, power, and sine. Columns 1 through 4 represent the visualization fitting effect of the first $100,000$ data points in these datasets using the newly proposed FPCS algorithm (column 1), the LTTB algorithm (column 2), the MinMaxLTTB algorithm (column 3), and the Reservoir Sampling algorithm (column 4), based on a $100:1$ sampling ratio. The red line represents the visualization result of the original data points, and the green line represents the visualization result of the sampled data points. Column 5 uses SSIM to compare the visual differences in the line charts of the sampled data points and the original data points.

Abstract—Data visualization aids in making data analysis more intuitive and in-depth, with widespread applications in fields such as biology, finance, and medicine. For massive and continuously growing streaming time series data, these data are typically visualized in the form of line charts, but the data transmission puts significant pressure on the network, leading to visualization lag or even failure to render completely. This paper proposes a universal sampling algorithm FPCS, which retains feature points from continuously received streaming time series data, compensates for the frequent fluctuating feature points, and aims to achieve efficient visualization. This algorithm bridges the gap in sampling for streaming time series data. The algorithm has several advantages: (1) It optimizes the sampling results by compensating for fewer feature points, retaining the visualization features of the original data very well, ensuring high-quality sampled data; (2) The execution time is the shortest compared to similar existing algorithms; (3) It has an almost negligible space overhead; (4) The data sampling process does not depend on the overall data; (5) This algorithm can be applied to infinite streaming data and finite static data.

Index Terms—Data visualization, Massive, Streaming, Time series, Line charts, Sampling, Feature, Compensating

1 Introduction

Data visualization assists in exploring and understanding data through intuitive charts and graphical representations, uncovering hidden trends, and providing a more comprehensive understanding of the overall situation, serving as an efficient method of information dissemination [15, 20, 38]. With the development of the internet, the domain of streaming data, which arrives in a sequential, large volume, quickly, continuously, and expands indefinitely over time, has become increasingly widespread [17, 28, 42]. The analysis of time series data that changes over time holds significant importance in fields such as economics and business [10, 11, 18]. However, visualizing massive data is extremely resource-intensive and can even cause program crashes [2, 6, 45].

To address such issues, Vitter proposed the Reservoir Sampling algorithm [40] in 1985, which enables sampling of large volumes of streaming data when the total data volume is unknown. This algorithm has applications in many data processing and machine learning tools,

• *Hongyan Li, Bo Yang, and Yansong Chua are with China Nanhu Academy of Electronics and Information Technology. E-mail: {3271961659@qq.com | ustcboy@outlook.com | caiyansong@cnaeit.com}.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

such as Apache Spark [43] and TensorFlow [1], but it retains each data point with the same probability, failing to achieve the goal of retaining feature points. Steinarsson introduced the LTTB (Largest-Triangle-Three-Buckets) [33] sampling algorithm in 2013, which significantly improves the fitting effect of visualization line charts for large volumes of time series data. It is applied in Apache ECharts [26] for sampling when rendering millions of data points, but LTTB requires calculating the triangular surface for each data point and depends on the overall data for bucketing, with the retention of data points being related to adjacent buckets. To enhance the efficiency of the LTTB algorithm, Van Der Donckt and others proposed the MinMaxLTTB [14] sampling algorithm in 2023, which is applied to the time series data visualization tool plotly-resampler [39]. However, issues such as the complexity of calculations and the existence of relationships between data points still exist, and both the LTTB and MinMaxLTTB algorithms are only suitable for processing finite static data.

To address the challenge of sampling streaming time series data, unlike all existing solutions, this paper introduces a brand-new algorithm called FPCS: For data points received in sequence, it continuously updates the maximum and minimum value points based on the value of the data points. According to the user-input sampling ratio, during each sampling operation, it retains the earlier received maximum or minimum value point. The other extreme value point not retained will be considered for compensation in the next sampling operation based on whether it conforms to the trend of data growth or decline between two sampling operations. Compared to the Reservoir Sampling, LTTB, and MinMaxLTTB algorithms, the FPCS algorithm achieves the best feature retention results for line charts, the best visualization effects, the shortest algorithm execution time, and requires almost negligible space, the sampling process also does not depend on the overall data, marking the first algorithm to achieve sampling of streaming time series data.

2 RELATED WORK

With the arrival of the era of big data, visualizing massive data is full of challenges [9, 23]. Storage requires sufficient resources, network transmission consumes a large amount of bandwidth, and visualization rendering requires strong hardware support. However, data visualization only needs to retain feature data that can be perceived by the human eye, and for a large amount of repetitive or unperceivable data, it can be discarded. Reasonable sampling [25, 27, 36] can significantly reduce the total amount of visualization data [24], solving the problem of data visualization in the background of big data. This paper mainly studies the sampling of streaming time series data, but there is currently no similar algorithm. Below, we will discuss the related work on streaming data sampling and time series data sampling separately.

2.1 Stream Sampling

Streaming data is an infinite collection of dynamic data that grows over time [34]. Streaming data has the following characteristics: data arrives in real-time; the scale of data is vast and unpredictable; once processed, unless specifically saved, the data cannot be reused. Sampling of streaming data is popular, with algorithms such as Reservoir Sampling [40], algorithms using sliding windows [7, 8, 35], and algorithms for processing graph streams [3, 19, 44].

The implementation of the Reservoir Sampling algorithm is as follows: set the total sampling quantity as k, when the nth data point arrives, if $n <= k$, then directly retain the received data point; otherwise, generate a random integer r in the range of 1 to n, if $r <= k$, then modify the rth item of the retained data with the newly received data. The space complexity of the Reservoir Sampling algorithm is $O(k)$, which is very efficient in handling large volumes of streaming data, but this algorithm is a type of random sampling, where each data point is retained with the same probability, of failing to achieve the goal of retaining feature points.

Many algorithms are using sliding windows, such as the algorithm proposed by Arasu et al. [7] in 2004 for maintaining approximate counts and quantiles of time-varying streaming data, which uses limited memory space but lacks generality and cannot be applied in other scenarios. Babcock et al. [8] proposed two algorithms for sampling data streams using sliding windows in 2002, one of which extends the Reservoir Sampling algorithm but requires maintaining a linked list for each data, which may occupy more space; the other algorithm's space consumption cannot be guaranteed, and both sampling algorithms are random sampling, resulting in poor preservation of feature points.

In the research on related works on streaming data sampling algorithms, this paper aims to propose an algorithm that only caches a limited amount of data and as few as possible, even if the total amount of data cannot be predicted, still controlling the space complexity within a relatively low range [30], and an algorithm that does not allow any relationship between the data as a whole.

2.2 Time Series Data Sampling

Time series data is a collection of data recorded in chronological order. It is typically visualized as simple two-dimensional line charts [4, 5], which are among the most widely used chart types [32, 37]. There has been considerable research on sampling time series line charts, such as MinMax, M4 [22], LTTB [33], and MinMaxLTTB [14].

The MinMax algorithm buckets all data according to a sampling ratio, retaining the maximum and minimum value points in each bucket. The maximum and minimum value points from all buckets form the result of the sampling. Although the MinMax algorithm has a very small time complexity, it requires caching all data points, resulting in a large space complexity, and the effectiveness of data sampling varies.

The M4 algorithm's bucketing method is similar to the MinMax algorithm, but in addition to retaining the maximum and minimum value points, it also retains the first and last points in each bucket. The points retained in all buckets form the result of the sampling. The M4 algorithm's performance is similar to the MinMax algorithm, but in the same dataset and sampling ratio, the sampling effect of the M4 algorithm is generally not as good as the MinMax algorithm.

The LTTB algorithm determines the size of the buckets based on the sampling ratio, divides the data points evenly into buckets, with the first and last points each occupying a bucket; retains the point in the first bucket; starting from the second bucket, iterates through all points in the bucket, calculates the area of the triangle formed by each point, the selected point of the previous bucket, and the average point of the next bucket, and selects the point with the largest area as the selected point of the current bucket; retains the point in the last bucket. The points retained in all buckets form the result of the sampling. The LTTB algorithm has a very significant sampling effect, maintaining the visual characteristics of the original graph; however, it almost requires complex calculations for all data points, the algorithm execution time is long, and during the selection of data points, points from adjacent buckets need to participate in the operation, indicating a correlation among the data.

To improve the performance of the LTTB algorithm, the MinMaxLTTB algorithm first uses the MinMax algorithm to preselect data, then applies the LTTB algorithm to the preselected points. This reduces the total amount of data that needs to be subjected to complex calculations, but the time complexity is still not as small as the MinMax algorithm or the M4 algorithm, the initial loading of all data points results in a large space complexity, and there is still a correlation between the data as a whole, requiring adjacent buckets' points to participate in the selection of data points.

In researching related works on time series data sampling algorithms, this paper aims to propose a novel algorithm that can maintain the excellent time complexity of the MinMax and M4 algorithms while achieving the visualization effect after sampling as the LTTB and MinMaxLTTB algorithms.

In summary, this paper is dedicated to proposing a completely new algorithm for sampling streaming time series data. In terms of visualization effect, it achieves the feature preservation results of the LTTB and MinMaxLTTB algorithms; in terms of time complexity, similar to the MinMax and M4 algorithms, it reduces a large number of complex calculations; in terms of space complexity, it leverages the advantages of streaming data, only caching a limited amount, controllable, and finite data in memory, reducing memory usage; especially, because it is processing streaming data, whether a data point is retained cannot depend on the overall data.

3 DESIGN PHILOSOPHY: WHICH, WHEN, AND HOW

Based on the analysis of Sec. 2, to achieve efficient sampling of streaming time series data, this section will describe the algorithm's design philosophy from three perspectives: Which, When, and How. This includes Retention Criteria for Data Points, Schedule of Sampling Operation, and Design of the Sampling Algorithm.

3.1 Which: Retention Criteria for Data Points

During the visualization process of time series data, extreme points form the key features of line charts, and over a continuous period, time series data typically exhibit fluctuating growth or decline trends. Therefore, in most cases, only the current maximum value point and minimum value point need to be considered during each sampling operation. This paper uses *MaxPoint* to represent the maximum value point and *MinPoint* to represent the minimum value point.

As shown in Fig. 2[b], during the previous sampling interval, the purple point is the *MinPoint*, and the green point is the *MaxPoint*. During the current sampling interval, the orange point is the *MaxPoint*, and the blue point is the *MinPoint*. Observably, the green point aligns the growth trend from the purple point to the orange point and can be not retained without causing significant differences in visualization. Similarly, the green point in Fig. 2[d] can also be not retained. So, sometimes it is sufficient to retain only one point between *MaxPoint* and *MinPoint*.

As shown in Fig. 2[a], during the previous sampling interval, the purple point is the *MinPoint*, and the green point is the *MaxPoint*. During the current sampling interval, the blue point is the *MinPoint*, and the orange point is the *MaxPoint*. Observably, the green point does not align with the growth trend from the purple point to the blue point, it is a feature point that should not be overlooked during visualization and needs to be retained. Similarly, the green point in Fig. 2[c] also needs to be retained. Therefore, sometimes it is necessary to retain both the *MaxPoint* and *MinPoint*.

In summary, to better fit the trend of the original data starting to change, each sampling operation retains only the earlier received *MaxPoint* or *MinPoint* within the current interval. For the *MinPoint* or *MaxPoint* received later, the decision to retain or discard is made in the next sampling operation. If this point aligns with the data change trend of the points retained in the adjacent two sampling operations, it does not need to be retained; otherwise, it must be retained. This paper uses *PotentialPoint* to represent the *MinPoint* or *MaxPoint* not retained in the previous sampling.

3.2 When: Schedule of Sampling Operation

For streaming data whose total quantity cannot be predicted and continues to arrive endlessly, based on the user-input sampling ratio $R:1$, a sampling operation is performed every time R data points are received.

Because there are situations where *PotentialPoint* needs to be compensated, sometimes a single sampling operation requires retaining two data points. Ultimately, the total number of sampled data points will be greater than or equal to the expected amount, but the compensation operation significantly improves the fitting effect of the visualized sampled data points to the original data points.

3.3 How: Design of the Sampling Algorithm

In Fig. 2[f], the current sampling interval corresponds to the previous sampling interval in Fig. 2[g]. As shown in Fig. 2[f], all points within the current sampling interval align with the growth trend from the blue point to the orange point in Fig. 2[g], and this interval lacks any prominent feature points that need to be retained for visualization purposes. Therefore, there exists a situation where no data points are retained within a single sampling interval. To maximize the retention of significant feature points during each sampling operation, considering the characteristics of streaming time series data, *MaxPoint* and *MinPoint* are continuously updated based on the values of received data points, rather than finding the maximum and minimum value points within each interval. This design allows intervals lacking obvious feature points during visualization to delegate the opportunity to retain data points to nearby intervals, and it meets the near real-time requirement for data point retention operations.

Whenever R data points are received and a sampling operation is needed, the earlier received *MaxPoint* or *MinPoint* is retained. If the retained point is *MaxPoint*, then *MaxPoint* and *PotentialPoint* are updated to *MinPoint*; otherwise, *MinPoint* and *PotentialPoint* are updated to *MaxPoint*. The design of updating data points ensures that the *MinPoint* or *MaxPoint* not retained in each sampling operation participates in the next sampling operation, preventing the loss of prominent feature points during visualization.

For most cases, only one data point needs to be retained within a single sampling operation. As shown in Fig. 2[b], *PotentialPoint* aligns with the growth trend from the purple point to the orange point, so the sampling operation only needs to retain the *MaxPoint* received earlier within the current sampling interval. As seen in Fig. 2[e], Fig. 2[f], and Fig. 2[h], if *PotentialPoint* and *MaxPoint*, or *PotentialPoint* and *MinPoint* are the same point, then that point is the earlier received point among *MaxPoint* and *MinPoint*, which is the point to be retained in the current sampling operation. Therefore, in such cases, there is no need to consider compensating for *PotentialPoint*, as the sampling operation only needs to retain the earlier received *MaxPoint* or *MinPoint*. There are only two special situations where compensation for *PotentialPoint* is needed: when both of the adjacent sampling operations retain *MinPoint*, and *PotentialPoint* and *MinPoint* are not the same point (Fig. 2[a]), or when both of the adjacent sampling operations retain *MaxPoint*, and *PotentialPoint* and *MaxPoint* are not the same point (Fig. 2[c]). In these two situations, *PotentialPoint* is likely to be a feature point that cannot be ignored during visualization, as it does not conform to the growth or decline trend from the retained data point to the current point to be retained. If not retained, significant oscillation changes would be lost during visualization.

In summary, the design of the sampling process of this algorithm is shown in Fig. 2[a] through Fig. 2[h]. Between adjacent images, "the current sampling" of the preceding image is "the previous sampling" of the following image. Dashed arrows show the updating process of variables. "retained data point" refers to data points that have already been retained. In the sampling operation of Fig. 2[a], *PotentialPoint* meets the compensation condition and needs to be retained. Retain the earlier *MinPoint* received. Update *MinPoint* and *PotentialPoint* to *MaxPoint*. In the sampling operation of Fig. 2[b], update *MaxPoint* and *MinPoint*. *PotentialPoint* does not meet the compensation condition and does not need to be retained. Retain the earlier *MaxPoint* received. Update *MaxPoint* and *PotentialPoint* to *MinPoint*. In the sampling operation of Fig. 2[c], update *MaxPoint* and *MinPoint*. *PotentialPoint* meets the compensation condition and needs to be retained. Retain the earlier *MaxPoint* received. Update *MaxPoint* and *PotentialPoint* to *MinPoint*. In the sampling operation of Fig. 2[d], update *MaxPoint* and *MinPoint*. *PotentialPoint* does not meet the compensation condition and does not need to be retained. Retain the earlier *MinPoint* received. Update *MinPoint* and *PotentialPoint* to *MaxPoint*. In the sampling operation of Fig. 2[e], update *MinPoint*. Retain the earlier *MaxPoint* received. Update *MaxPoint* and *PotentialPoint* to *MinPoint*. In the sampling operation of Fig. 2[f], update *MaxPoint*. Retain the earlier *MinPoint* received. Update *MinPoint* and *PotentialPoint* to *MaxPoint*. In the sampling operation of Fig. 2[g], update *MaxPoint* and *MinPoint*. *PotentialPoint* does not meet the compensation condition and does not need to be retained. Retain the earlier *MaxPoint* received. Update *MaxPoint* and *PotentialPoint* to *MinPoint*. In the sampling operation of Fig. 2[h], update *MaxPoint*. Retain the earlier *MinPoint* received. Update *MinPoint* and *PotentialPoint* to *MaxPoint*. Connecting Fig. 2[a] through Fig. 2[h] as shown in Fig. 2[i], sample 270 data points and retain a total of 11 data points highlighted in blue, successfully preserving the features for data visualization.

Therefore, the design of this algorithm is as follows: User input a sampling ratio $R:1$. Continuously update *MaxPoint* and *MinPoint* based on the values of received data points. Perform a sampling operation once every R data points received. Compensate for the

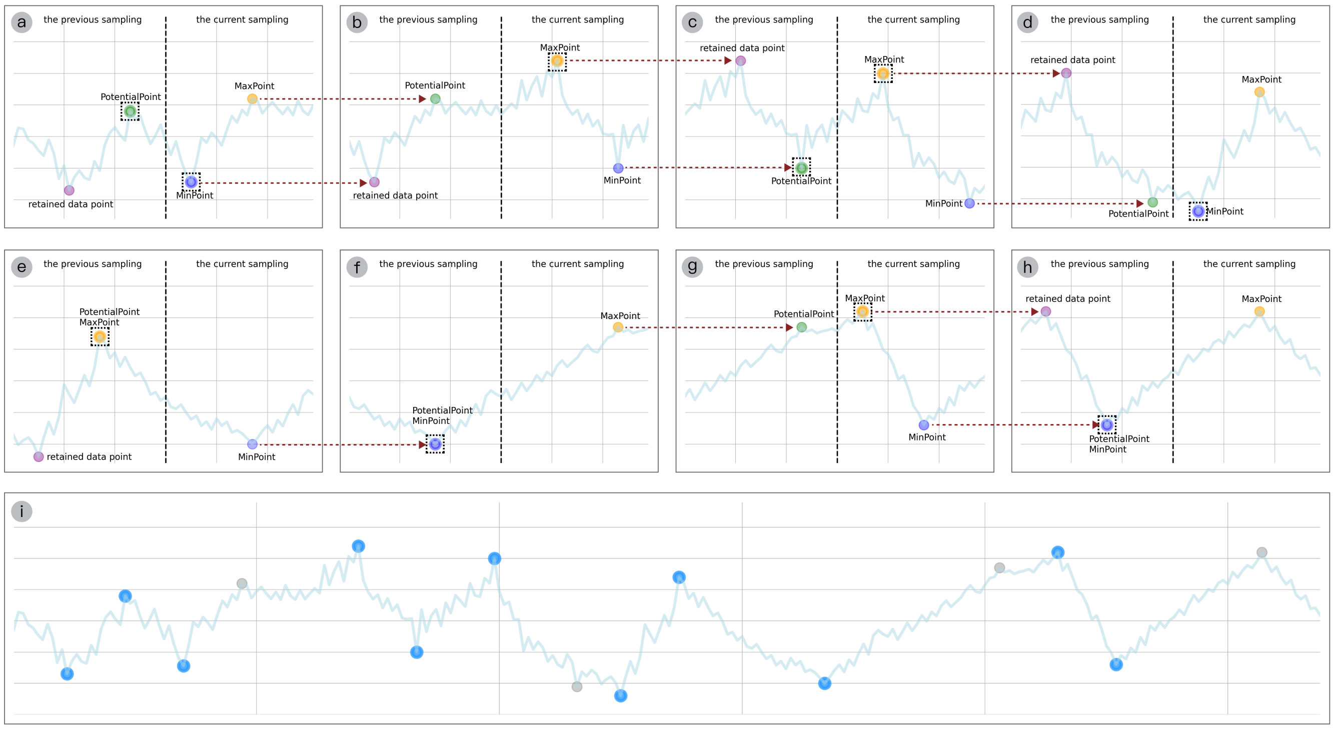

Fig. 2: Figures a through h illustrate the sampling process of 270 data points at a sampling ratio of $30:1$. In each image, the data points to be retained in the current sampling are framed with a black dashed rectangle. Between adjacent images, "the current sampling" of the preceding image is "the previous sampling" of the following image. Dashed arrows show the updating process of variables. Figure i is the image formed by connecting Figures a through h, showing the final sampling effect of 270 data points, with a total of 11 data points highlighted in blue that were retained.

$PotentialPoints$ under two special conditions, retaining the earlier received $MaxPoint$ or $MinPoint$. This algorithm's variable update method meets the requirements for near real-time sampling of streaming time series data; it achieves the effect of preserving as many original data visualization features as possible under a limited sampling ratio.

4 FPCS

Based on the analysis of Sec. 3, to achieve efficient sampling of continuously arriving streaming time series data while maintaining high visualization fidelity and low time and space complexity, unlike all existing algorithms, this paper proposes the innovative FPCS algorithm for the first time. The algorithm is primarily divided into the following steps: (i) Determine the sampling ratio; (ii) Initialize variables; (iii) Decide to update the maximum or minimum value point based on the newly received data points; (iv) When sampling is required according to the sampling ratio, determine whether to compensate for the minimum or maximum value point not retained in the previous sampling and retain the maximum or minimum value point received earlier. Supplemental Materials include a digital video dynamically showing the FPCS algorithm sampling process and the code implementation of the FPCS algorithm.

The first step involves determining the sampling ratio $R:1$ based on user input, meaning that sampling occurs once after receiving R data points. Therefore, R must be a positive integer greater than 1; Otherwise, the input data will be incorrect, and sampling will not be needed.

The second step initializes the maximum value point $MaxPoint$ and the minimum value point $MinPoint$ to the first received data point. It also initializes the maximum or minimum value point not retained in the previous sampling $PotentialPoint$ to Null, the flag indicating whether the minimum value point was retained in the previous sampling $PreviousMinFlag$ to -1, and the $Counter$ to 0.

The third step increments $Counter$ by 1 each time a data point $P(x,y)$ is received. If the value $P.y$ of the newly received data point P is greater than or equal to the value $MaxPoint.y$ of $MaxPoint$, $MaxPoint$ is updated to P. If the value $P.y$ of the newly received data point P is less than the value $MinPoint.y$ of $MinPoint$, $MinPoint$ is updated to P. As shown in Fig. 2[a], when the previous sampling ends, $MaxPoint$, $MinPoint$, and $PotentialPoint$ are all updated to the green point in the diagram. After the current sampling begins, because values larger and smaller data points have been received, $MaxPoint$ is updated to the orange point in the diagram, and $MinPoint$ is updated to the blue point in the diagram.

The fourth step checks if $Counter$ is greater than or equal to R, signifying that data points need to be retained at this time. If both the previous sampling and the current sampling retain either the maximum value point or the minimum value point, and $PotentialPoint$ and the currently to-be-retained data point are not the same point, such situations require compensation for $PotentialPoint$ since it often represents obvious feature points. Retain the earlier received point between $MaxPoint$ and $MinPoint$. Update $MaxPoint$, $MinPoint$, and $PotentialPoint$ to the currently unretained $MinPoint$ or $MaxPoint$. If the current sampling retains $MinPoint$, update $PreviousMinFlag$ to 1; otherwise, set it to 0. Finally, reset $Counter$ to 0 and return to step three. As shown in Fig. 2[a], in both the previous sampling and the current sampling, the earlier received minimum value point is retained, and $PotentialPoint$ and $MinPoint$ are not the same point, so the $PotentialPoint$ in the figure meets the criteria for retaining feature points and requires compensation. If this point were not retained, it would lose the obvious upward trend from the purple point to the green point, as well as the clear downward trend from the green point to the blue point. However, the $PotentialPoint$ in Fig. 2[d] does not meet the criteria for retaining feature points and does not require compensation. This point falls within the range of decreasing fluctuations from the purple point to the blue point and can be ignored during visualization. Since $MinPoint$ is received before $MaxPoint$ ($MinPoint.x < MaxPoint.x$) in both Fig. 2[a] and Fig. 2[d], $MinPoint$ needs to be retained. In the current sampling, having retained $MinPoint$, update $MinPoint$ and $PotentialPoint$ to the

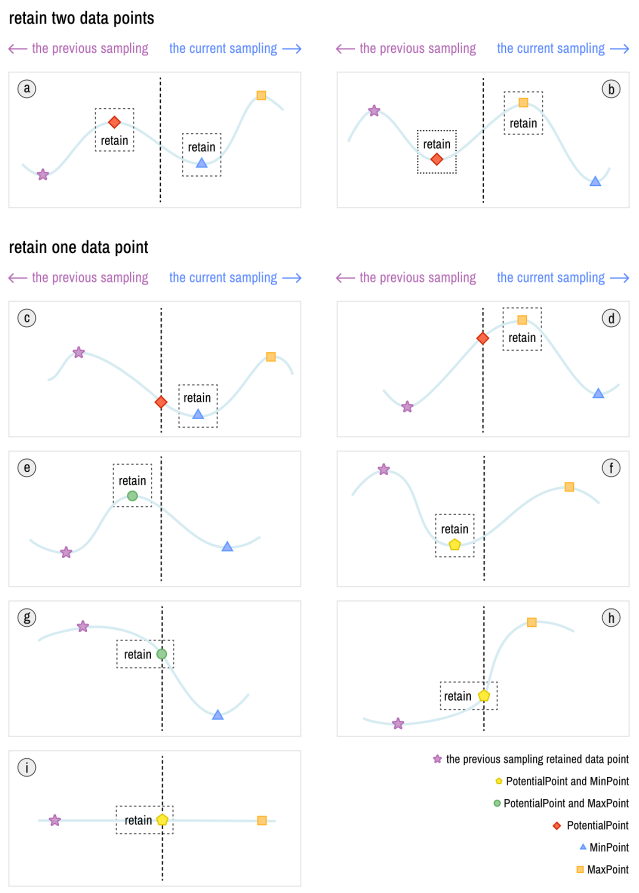

Fig. 3: Using the FPCS algorithm, sampling is performed on 9 discrete theoretical cases. In each image, the data points to be retained in the current sampling are framed with a black dashed rectangle.

unretained *MaxPoint*, and update *PreviousMinFlag* to 1. Finally, reset the *Counter* to 0.

The FPCS algorithm repeats steps three and four for each received data point until the data stream ends. Algorithm 1 shows the sampling process of the FPCS algorithm. Fig. 3 displays nine discrete theoretical cases and the sampling outcomes under different distributions of data points.

The FPCS algorithm only maintains the maximum value point *MaxPoint*, the minimum value point *MinPoint*, the maximum or minimum value point *PotentialPoint* not retained in the previous sampling, a *Counter*, and a flag *PreviousMinFlag*, so the space complexity of the algorithm is $O(1)$.

The FPCS algorithm performs very few comparison calculations and assignment operations each time it receives a new data point, so the algorithm execution time is very short. Assuming a total of n data points are received, the time complexity of the algorithm is $O(n)$.

5 Experimental Results

In this section, through numerous different experiments, we analyze the FPCS algorithm, including Visual Effects, Analysis of Visual Differences, Performance, etc. Supplemental Materials provide datasets, source codes, and experimental results for all experiments.

5.1 Experimental Setup

The following will describe the experimental environment and dataset used for testing the FPCS algorithm's results and performance.

Algorithm 1 Workflow of FPCS

```
Require: Sampling ratio $R : 1$
Ensure: Use $MaxPoint$ to represent the point of the maximum value. Use $MinPoint$ to represent the point of the minimum value. Use $PotentialPoint$ to represent the point of the maximum or minimum value not retained from the previous sampling. Use $PreviousMinFlag$ to represent whether the minimum value point was retained from the previous sampling. Use $Counter$ to count
 1: Initialize $PotentialPoint = NULL$, $PreviousMinFlag = -1$, $Counter = 0$   ▷ $PreviousMinFlag$ : $-1$(no data points have been retained), 0(the previous sampling retained $MaxPoint$), 1(the previous sampling retained $MinPoint$)
 2: $FP$ = the first data point received
 3: $MaxPoint = FP$
 4: $MinPoint = FP$
 5: $Counter = Counter + 1$
 6: while receive a data point $P \leftarrow (x, y)$ do
 7:     $Counter = Counter + 1$
 8:     if $P.y \geq MaxPoint.y$ then
 9:         $MaxPoint = P$   ▷ Update $MaxPoint$
10:     else
11:         if $P.y < MinPoint.y$ then
12:             $MinPoint = P$   ▷ Update $MinPoint$
13:         end if
14:     end if
15:     if $Counter \geq R$ then
16:         if $MinPoint.x < MaxPoint.x$ then
17:             if $(PreviousMinFlag == 1)$ and $(MinPoint \neq PotentialPoint)$ then
18:                 Retain $PotentialPoint$   ▷ Both adjacent samplings retain $MinPoint$, and $PotentialPoint$ and $MinPoint$ are not the same point, so $PotentialPoint$ needs compensation, as shown in Fig. 2[a]
19:             end if
20:             Retain $MinPoint$   ▷ Receiving $MinPoint$ before $MaxPoint$, retain $MinPoint$
21:             $PotentialPoint = MaxPoint$   ▷ Update $PotentialPoint$ to the unretained $MaxPoint$
22:             $MinPoint = MaxPoint$   ▷ Update $MinPoint$ to the unretained $MaxPoint$
23:             $PreviousMinFlag = 1$   ▷ In the current sampling, $MinPoint$ is retained, so $PreviousMinFlag$ is updated to 1
24:         else
25:             if $(PreviousMinFlag == 0)$ and $(MaxPoint \neq PotentialPoint)$ then
26:                 Retain $PotentialPoint$   ▷ Both adjacent samplings retain $MaxPoint$, and $PotentialPoint$ and $MaxPoint$ are not the same point, so $PotentialPoint$ needs compensation, as shown in Fig. 2[c]
27:             end if
28:             Retain $MaxPoint$   ▷ Receiving $MaxPoint$ before $MinPoint$, retain $MaxPoint$
29:             $PotentialPoint = MinPoint$   ▷ Update $PotentialPoint$ to the unretained $MinPoint$
30:             $MaxPoint = MinPoint$   ▷ Update $MaxPoint$ to the unretained $MinPoint$
31:             $PreviousMinFlag = 0$   ▷ In the current sampling, $MaxPoint$ is retained, so $PreviousMinFlag$ is updated to 0
32:         end if
33:         $Counter = 0$
34:     end if
35: end while
```

5.1.1 Experimental Environment

The experiment was conducted on a hardware configuration with an Intel(R) Core(TM) i9-12900K processor (4.2GHZ), 64GB DDR4 memory, and an NVIDIA GeForce RTX 3090 graphics card. The operating system used was Ubuntu 20.04.6 LTS 64-bit version. The main software packages used in the experiment included Python 3.11.8, NEST 3.6.0, rustic 1.78.0-nightly, argminmax 0.6.1, tsdownsample 0.1.2, and plotly_resampler 0.9.2.

5.1.2 Dataset

To demonstrate the effectiveness of the sampling algorithm in retaining feature points, the dataset must have a large number of fluctuating data points. To test the performance of the algorithm, it is important to avoid the impact of special cases, so the total amount of data in the dataset should be sufficient. The main focus of this paper is on sampling streaming time series data, and the data in the dataset must be time series data.

Given these characteristics, the experiment selected the NEST simulator [13, 16] from the Neural Simulation Technology Initiative to generate the experimental dataset. NEST is a simulator for spiking neural network models, providing over 50 neuron models. The experiment chose the Cortical microcircuit model [31], and by recording the Membrane Potential data generated by the model, a dataset that meets the conditions can be obtained. The simulation total time was set to $1,000,000ms$, with a simulation step of $0.01ms$, meaning that a data point is collected every $0.01ms$. After the simulation ends, a Membrane Potential dataset of size $100,000,000(100M)$ and meeting the conditions can be obtained.

To prove the universality of the FPCS algorithm, the experiment will also use four algorithms to sample datasets from various sources in the DEBS Grand Challenge [21, 29] and UCR Time Series Classification Archive [12], comparing the sampling effects of the four algorithms under different datasets and sampling ratios.

5.2 Results

The following will discuss the sampling results of the FPCS, Reservoir Sampling, LTTB, and MinMaxLTTB algorithms from the perspectives of visual effects and visual differences, and will analyze the sampling effects of the four algorithms based on multiple typical datasets.

5.2.1 Visual Effect

To ensure the objectivity of the experimental results, a data point was randomly selected from the Membrane Potential dataset as the starting point. Given the limited visualization graph display range, the experiment sampled the continuous $50,000(50K)$ data points starting from the starting point using the FPCS, Reservoir Sampling, LTTB, and MinMaxLTTB algorithms, comparing the results of retaining feature points.

To facilitate observation, the experiment set the sampling ratio to $100:1$, meaning 100 data points are retained for 1. Because the FPCS algorithm compensates for some data points in special cases, the FPCS algorithm was first used to sample the data points, and then the remaining three algorithms were used for sampling based on the number of data points sampled by the FPCS algorithm.

Fig. 4 is a comparison chart of the fitting effect of visualizing the sampled data points with the original data points using the FPCS algorithm, sampling $50K$ data points with an initial sampling ratio of $100:1$, retaining 576 data points. Fig. 5 uses the Reservoir Sampling algorithm, Fig. 6 uses the LTTB algorithm, and Fig. 7 uses the MinMaxLTTB algorithm, to sample $50K$ data points, retaining 576 data points, and comparing the fitting effect of visualizing the sampled data points with the original data points.

Comparing the experimental results of the four algorithms on the same dataset and the same sampling ratio. The Reservoir Sampling algorithm performs very poorly in retaining feature points. The LTTB and MinMaxLTTB algorithms are not ideal in handling a large number of extrema. The FPCS algorithm's visual fitting line chart after sampling is almost identical to the original data points, with the highest fitting degree among the images, the complete retention of effective feature points, and the best visualization effect.

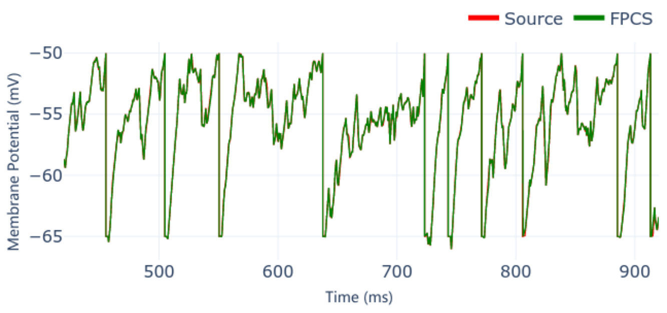

Fig. 4: Using the FPCS algorithm, 576 data points are retained from $50K$ data points, with a comparison chart of visual fitting effects with the original data points.

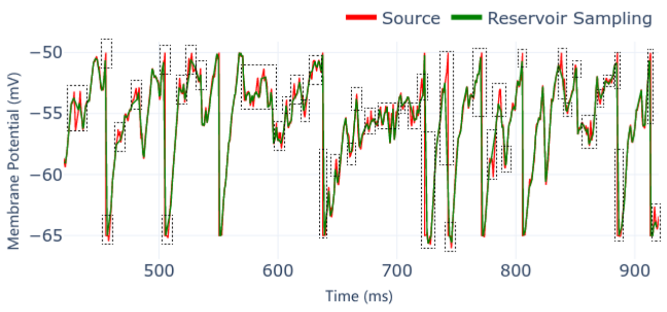

Fig. 5: Using the Reservoir Sampling algorithm, 576 data points are retained from $50K$ data points, with a comparison chart of visual fitting effects with the original data points. A black dashed rectangle is used in the figure to highlight differences.

5.2.2 Analysis of Visual Differences

Considering the quantitative analysis of the visual effect after sampling, the experiment was conducted on a Membrane Potential dataset with a total capacity of $1,000,000(1M)$, using sampling ratios of $100:1$, $200:1$, $500:1$, and $1000:1$. Four algorithms, FPCS, Reservoir Sampling, LTTB, and MinMaxLTTB, were used for sampling, and the sampled data points were drawn into corresponding visualization line charts. To use data to reflect the difference between the visualization line charts of the sampled data points and the original data points, the experiment chose SSIM [41] to measure the similarity between pictures. The closer the SSIM value is to 1, the smaller the visual difference between the two pictures. To ensure that the SSIM measurement value reflects the

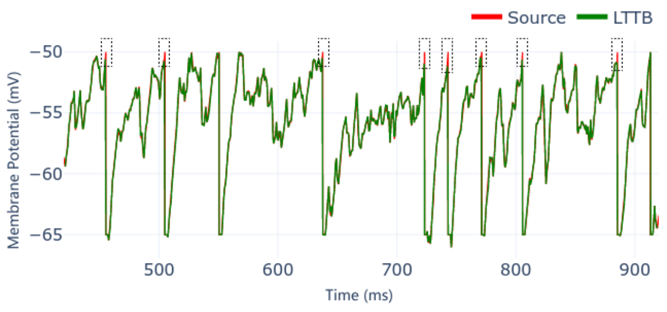

Fig. 6: Using the LTTB algorithm, 576 data points are retained from $50K$ data points, with a comparison chart of visual fitting effects with the original data points. A black dashed rectangle is used in the figure to highlight differences.

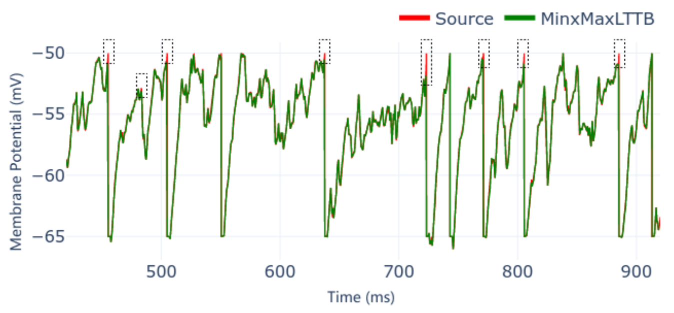

Fig. 7: Using the MinMaxLTTB algorithm, 576 data points are retained from $50K$ data points, with a comparison chart of visual fitting effects with the original data points. A black dashed rectangle is used in the figure to highlight differences.

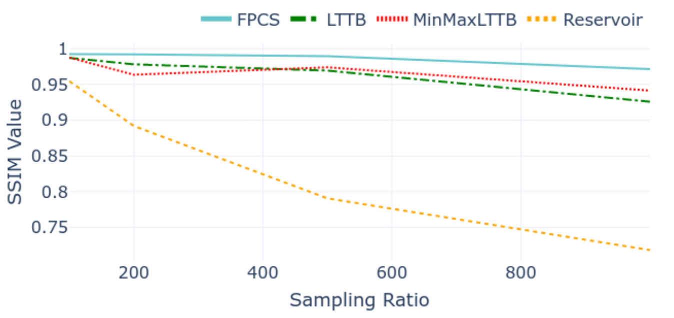

Fig. 8: Using the FPCS, Reservoir Sampling, LTTB, and MinMaxLTTB algorithms, sampling is performed on a dataset with a total capacity of $1M$ based on sampling ratios of $100:1$, $200:1$, $500:1$, and $1000:1$. A comparison is made between the visualization of the sampled data points and the original data points in line charts.

superiority or inferiority of the sampling effect as much as possible, all generated visualization line charts in the experiment maintained consistent image sizes, image brightness, etc. Fig. 8 shows the trend of SSIM values under the same dataset, four different sampling ratios, and four different algorithms.

Analyzing Fig. 8, comparing the other three algorithms, the FPCS algorithm has the highest and most stable SSIM value at all sampling ratios, supporting from a data perspective that the visualization effect of this algorithm is the best, with the smallest visual difference from the original data points visualization line chart.

5.2.3 Typical Dataset Sampling Results

FPCS is a universal algorithm for sampling streaming time series data. Using the FPCS, Reservoir Sampling, LTTB, and MinMaxLTTB algorithms, sampling was performed on five typical datasets from the DEBS 2012 Grand Challenge [21], DEBS 2013 Grand Challenge [29], and UCR Time Series Classification Archive [12]. For ease of observation, experiments were conducted on the first $100,000(100K)$ data points of the ball, cinecg, noise, power, and sine datasets. Fig. 1 shows a comparison of the fitting effect and SSIM values of the visualized line chart of the sampled data points against the original data points for the five typical datasets, using the four algorithms, based on a $100:1$ sampling ratio. Fig. 9 displays the visual differences in the visualized line charts of the sampled data points against the original data points for the five typical datasets, using the four algorithms, under different sampling ratios of $100:1$, $200:1$, $500:1$, and $1000:1$.

Observing Fig. 1, among the five typical datasets, the FPCS algorithm shows the best-fitting effect of the visualized line chart between sampled data points and original data points, with the highest SSIM values, and the best preservation results for feature points, when compared to the other three algorithms. Fig. 9 supports this from a data perspective, showing that the FPCS algorithm's sampling effect is the best across different sampling ratios and various source datasets.

5.3 Performance

To test the performance of the FPCS algorithm, experiments were conducted on the Membrane Potential dataset with a total capacity of $100M$, incrementing by $10M$ each time. The time and space complexity of FPCS, Reservoir Sampling, LTTB, MinMaxLTTB, and MinMaxLTTB (parallel) were calculated based on the same dataset, with a sampling ratio of $100:1$. To ensure the accuracy of the experimental results, each experiment was repeated hundreds of times, and the average of all results was taken as the final value.

5.3.1 Time Complexity

The FPCS algorithm performs only simple comparison calculations and assignment operations on all received data points. Theoretically, the time complexity of the algorithm is $O(n)$.

To avoid interference from the network on performance testing, the experiments aimed to reflect only the time required for the algorithm execution, with time complexity experiments conducted on local static datasets. The Reservoir Sampling algorithm is specifically designed for stream data, making it irrelevant to compare with Reservoir Sampling in this context.

Fig. 10 shows a comparison of execution time for the FPCS, LTTB, MinMaxLTTB, and MinMaxLTTB (parallel) algorithms across different dataset sizes ($10M$, $20M$, $30M$, $40M$, $50M$, $60M$, $70M$, $80M$, $90M$, $100M$). To ensure objectivity in the experiment, all four algorithms were implemented in Rust, with LTTB, MinMaxLTTB, and MinMaxLTTB (parallel) using the most efficient implementations available.

Observing Fig. 10, it is intuitively clear that under the Membrane Potential dataset and a sampling ratio of $100:1$ when sampling across 10 datasets of different sizes, the FPCS algorithm has the shortest execution time; the MinMaxLTTB algorithm has the longest execution time; the execution times of the LTTB and MinMaxLTTB (parallel) algorithms are relatively good.

Theoretically, the time complexity of the FPCS algorithm is $O(n)$, and the computers perform only a small amount of binary operations on each data point, so the execution time of the FPCS algorithm will increase slowly as the number of data points increases, with a lower slope on the curve, resulting in shorter algorithm execution times and better time complexity performance.

5.3.2 Space Complexity

The FPCS algorithm processes streaming data, regardless of the size of the dataset, only requiring 3 data points, 1 counter, and 1 flag bit to be cached during the execution process. Each data point has 2 coordinates, all of which are of the $float64$ type; the counter and flag bit are both of the $int64$ type. Therefore, theoretically, it only occupies 64 bytes of space, the space complexity of the algorithm is $O(1)$.

During the execution of the LTTB, MinMaxLTTB, and MinMaxLTTB (parallel) algorithms, the space occupied will increase as the size of the sampled dataset increases. Testing showed that the space occupied by sampling a dataset of $10M$ using these three algorithms was greater than $130MB$, making a comparison of these algorithms in this context meaningless.

Tab. 1 records the comparison results of the maximum space occupied during the execution process of the FPCS and Reservoir Sampling algorithms on different scale datasets ($10M$, $20M$, $30M$, $40M$, $50M$, $60M$, $70M$, $80M$, $90M$, $100M$). To ensure the fairness of the experimental results, both the FPCS and Reservoir Sampling algorithms were implemented in Python, with Reservoir Sampling utilizing the official library provided.

Observing Tab. 1, it is intuitively clear that under the Membrane Potential dataset and a sampling ratio of $100:1$ when sampling across 10 datasets of different sizes, the FPCS algorithm has the best space complexity performance, with the maximum space occupied is less than $0.1MB$ regardless of the size of the dataset. The Reservoir Sampling algorithm caches all the data points it intends to retain, so its space

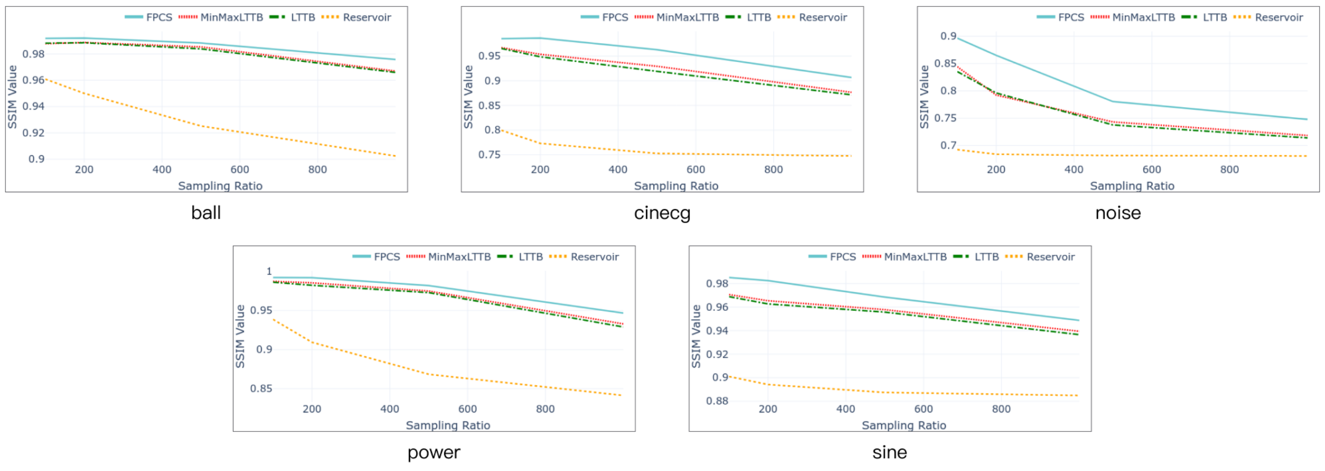

Fig. 9: Based on sampling ratios of $100:1$, $200:1$, $500:1$, and $1000:1$, using the FPCS, Reservoir Sampling, LTTB, and MinMaxLTTB algorithms, sampling is performed on the first $100K$ data points of the five typical datasets. Under different sampling ratios and from various source datasets, the visual differences in the visualized line charts of the sampled data points against the original data points are displayed for the four algorithms.

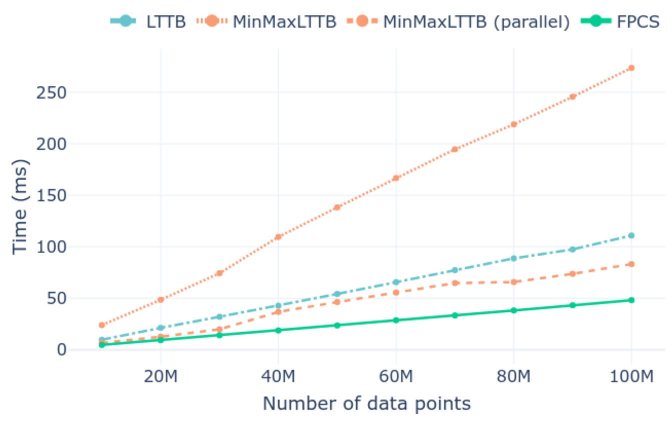

Fig. 10: Time complexity performance comparison of FPCS, LTTB, MinMaxLTTB, and MinMaxLTTB (parallel). The FPCS algorithm is implemented in Rust, the LTTB algorithm is implemented in Rust using plotly_resampler v0.9.2, and both the MinMaxLTTB algorithm and the MinMaxLTTB (parallel) algorithm are implemented in Rust using tsdownsample v0.1.2.

Table 1: A comparison table of the maximum space occupied during the execution process of the FPCS and Reservoir Sampling algorithms.

Dataset Size	FPCS (MB)	Reservoir Sampling (MB)
10M	0.086	7.926
20M	0.071	15.352
30M	0.079	26.844
40M	0.092	30.211
50M	0.089	33.637
60M	0.086	53.063
70M	0.087	56.551
80M	0.086	59.902
90M	0.093	63.344
100M	0.068	66.824

usage will continue to increase as the number of data points to retain increases.

Theoretically, the space complexity of the FPCS algorithm is $O(1)$, The space complexity of the algorithm is not affected by the original dataset size or the number of data points to be retained, and the space occupied by the algorithm during execution is very small and limited, resulting in excellent space complexity performance.

6 Conclusion

This paper proposes a universal FPCS algorithm, which is the first to implement sampling for streaming time series data. Compared to the most widely used algorithms such as the Reservoir Sampling algorithm, LTTB algorithm, and MinMaxLTTB algorithm, the FPCS algorithm achieves the best visualization effect for retaining feature points, fitting the visualized line chart of original data points best. In terms of time complexity, the FPCS algorithm does not involve complex calculations, resulting in the shortest algorithm execution time. Regarding space complexity, regardless of the size of the dataset, it only occupies less than $0.1MB$ of memory, making the algorithm execution space the smallest. Based on typical datasets from various sources, extensive testing has been conducted at different sampling ratios, and the visual difference between the sampled data points and the original data points visualized in line charts is the smallest. Future work will focus on optimizing the FPCS algorithm to dynamically adjust the sampling ratio during the sampling process, adapting to the characteristics of different datasets, and further improving the efficiency of sampling. The FPCS algorithm is expected to be widely applied in fields such as medicine, biology, and finance, where data needs to be retained for features and efficient visualization.

Supplemental Materials

All supplemental materials have been uploaded to the submission system. In particular, they include (1) A digital video titled "Animation demonstration of the FPCS algorithm.mp4". This video dynamically demonstrates the sampling process of the FPCS algorithm and the distribution of the top ten data points with the highest occurrence probability; (2) Another digital video titled "Sampling process and effect of the FPCS algorithm.mp4". This video dynamically displays a real sampling process and the final sampling effect of the FPCS algorithm on the Membrane Potential dataset; (3) An engineering file named "fpcs.zip". This includes all source code and datasets. It contains implementations of the FPCS algorithm based on both static and streaming data; and all experiments and results mentioned in this paper.

Acknowledgments

We would like to thank the anonymous reviewers for their precious comments and suggestions.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. {TensorFlow}: A system for {Large-Scale} machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283. USENIX Association, Savannah, 2016. 2

[2] R. Agrawal, A. Kadadi, X. Dai, and F. Andres. Challenges and opportunities with big data visualization. In *Proceedings of the 7th International Conference on Management of Computational and Collective IntElligence in Digital EcoSystems*, pp. 169–173. Association for Computing Machinery, New York, 2015. doi: 10.1145/2857218.2857256 1

[3] N. K. Ahmed, J. Neville, and R. Kompella. Network sampling: From static to streaming graphs. *ACM Trans. Knowl. Discov. Data*, 8(2):1–56, 2013. doi: 10.1145/2601438 2

[4] W. Aigner, A. Bertone, S. Miksch, C. Tominski, and H. Schumann. Towards a conceptual framework for visual analytics of time and time-oriented data. In *2007 Winter Simulation Conference*, pp. 721–729. IEEE, New York, 2007. doi: 10.1109/WSC.2007.4419666 2

[5] W. Aigner, S. Miksch, W. MÃijller, H. Schumann, and C. Tominski. Visualizing time-oriented dataâĂŤa systematic view. *Comput. Graphics*, 31(3):401–409, 2007. doi: 10.1016/j.cag.2007.01.030 2

[6] S. M. Ali, N. Gupta, G. K. Nayak, and R. K. Lenka. Big data visualization: Tools and challenges. In *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, pp. 656–660. IEEE, New York, 2016. doi: 10.1109/IC3I.2016.7918044 1

[7] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 286–296. Association for Computing Machinery, New York, 2004. doi: 10.1145/1055558.1055598 2

[8] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 633–634. Society for Industrial and Applied Mathematics, USA, 2002. 2

[9] N. Bikakis. Big data visualization tools. *arXiv preprint arXiv:1801.08336*, 2018. doi: 10.48550/arXiv.1801.08336 2

[10] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung. *Time Series Analysis: Forecasting and Control*. John Wiley & Sons, Hoboken, 5nd ed., 2015. 1

[11] K. Choi, J. Yi, C. Park, and S. Yoon. Deep learning for anomaly detection in time-series data: Review, analysis, and guidelines. *IEEE Access*, 9:120043–120065, 2021. doi: 10.1109/ACCESS.2021.3107975 1

[12] H. A. Dau, A. Bagnall, K. Kamgar, C.-C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, and E. Keogh. The ucr time series archive. *IEEE/CAA J. Autom. Sin.*, 6(6):1293–1305, 2019. doi: 10.1109/JAS.2019.1911747 6, 7

[13] M. Diesmann and M.-O. Gewaltig. Nest: An environment for neural systems simulations. *Forschung und wisschenschaftliches Rechnen, BeitrÃď'ge zum Heinz-Billing-Preis*, 58:43–70, 2001. 6

[14] J. V. D. Donckt, J. V. D. Donckt, M. Rademaker, and S. V. Hoecke. Minmaxlttb: Leveraging minmax-preselection to scale lttb. In *2023 IEEE Visualization and Visual Analytics (VIS)*, pp. 21–25. IEEE, New York, 2023. doi: 10.1109/VIS54172.2023.00013 2

[15] O. Embarak. *The Importance of Data Visualization in Business Intelligence*, pp. 85–124. Apress, Berkeley, 1nd ed., 2018. doi: 10.1007/978-1-4842-4109-7_2 1

[16] M.-O. Gewaltig and M. Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007. doi: 10.4249/scholarpedia.1430 6

[17] H. M. Gomes, J. Read, A. Bifet, J. P. Barddal, and J. a. Gama. Machine learning for streaming data: state of the art, challenges, and opportunities. *SIGKDD Explor. Newsl.*, 21(2):6–22, 2019. doi: 10.1145/3373464.3373470 1

[18] J. D. Hamilton. *Time Series Analysis*. Princeton University Press, Princeton, 2nd ed., 1994. doi: 10.1515/9780691218632 1

[19] C. Huan, S. L. Song, Y. Liu, H. Zhang, H. Liu, C. He, K. Chen, J. Jiang, and Y. Wu. T-gcn: A sampling based streaming graph neural network system with hybrid architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 69–82. Association for Computing Machinery, New York, 2023. doi: 10.1145/3559009.3569648 2

[20] M. Islam and S. Jin. An overview of data visualization. In *2019 International Conference on Information Science and Communications Technologies (ICISCT)*, pp. 1–7. IEEE, New York, 2019. doi: 10.1109/ICISCT47635.2019.9012031 1

[21] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic. The debs 2012 grand challenge. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pp. 393–398. Association for Computing Machinery, New York, 2012. doi: 10.1145/2335484.2335536 6, 7

[22] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4: a visualization-oriented time series data aggregation. *Proc. VLDB Endow.*, 7(10):797–808, 2014. doi: 10.14778/2732951.2732953 2

[23] M. S. Kahil, A. Bouramoul, and M. Derdour. Big data and interactive visualization: Overview on challenges, techniques and tools. In *Advanced Intelligent Systems for Sustainable Development (AI2SD'2019)*, pp. 157–167. Springer International Publishing, Cham, 2020. doi: 10.1007/978-3-030-36674-2_17 2

[24] B. C. Kwon, J. Verma, P. J. Haas, and Ã. Demiralp. Sampling for scalable visual analytics. *IEEE Comput. Graphics Appl.*, 37(1):100–108, 2017. doi: 10.1109/MCG.2017.6 2

[25] P. S. Levy and S. Lemeshow. *Sampling of Populations: Methods and Applications*. John Wiley & Sons, Hoboken, 4nd ed., 2013. 2

[26] D. Li, H. Mei, Y. Shen, S. Su, W. Zhang, J. Wang, M. Zu, and W. Chen. Echarts: A declarative framework for rapid construction of web-based visualization. *Visual Informatics*, 2(2):136–146, 2018. doi: 10.1016/j.visinf.2018.04.011 2

[27] S. L. Lohr. *Sampling: Design and Analysis*. Chapman and Hall/CRC, New York, 3nd ed., 2021. doi: 10.1201/9780429298899 2

[28] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani. Deep learning for iot big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):2923–2960, 2018. doi: 10.1109/COMST.2018.2844341 1

[29] C. Mutschler, H. Ziekow, and Z. Jerzak. The debs 2013 grand challenge. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, pp. 289–294. Association for Computing Machinery, New York, 2013. doi: 10.1145/2488222.2488283 6, 7

[30] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. Streaming-data algorithms for high-quality clustering. In *Proceedings 18th International Conference on Data Engineering*, pp. 685–694. IEEE, New York, 2002. doi: 10.1109/ICDE.2002.994785 2

[31] T. C. Potjans and M. Diesmann. The cell-type specific cortical microcircuit: Relating structure and activity in a full-scale spiking network model. *Cereb. Cortex*, 24(3):785–806, 2012. doi: 10.1093/cercor/bhs358 6

[32] A. Spreafico and G. Carenini. Neural data-driven captioning of time-series line charts. In *Proceedings of the 2020 International Conference on Advanced Visual Interfaces*, pp. 1–5. Association for Computing Machinery, New York, 2020. doi: 10.1145/3399715.3399829 2

[33] S. Steinarsson. *Downsampling time series for visual representation*. PhD thesis, University of Iceland, Iceland, 2013. 2

[34] W. Szewczyk. Streaming data. *Wiley Interdiscip. Rev. Comput. Stat.*, 3(1):22–29, 2011. doi: 10.1002/wics.130 2

[35] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *Proc. VLDB Endow.*, 8(7):702–713, 2015. doi: 10.14778/2752939.2752940 2

[36] S. K. Thompson. *Sampling*. John Wiley & Sons, Hoboken, 3nd ed., 2012. 2

[37] T. Trautner and S. Bruckner. Line weaver: Importance-driven order enhanced rendering of dense line charts. *Comput. Graphics Forum*, 40(3):399–410, 2021. doi: 10.1111/cgf.14316 2

[38] A. Unwin. Why is data visualization important? what is important in data visualization? *Harvard Data Sci. Rev.*, 2(1):1–7, 2020. doi: 10.1162/99608f92.8ae4d525 1

[39] J. Van Der Donckt, J. Van der Donckt, E. Deprost, and S. Van Hoecke. Plotly-resampler: Effective visual analytics for large time series. In *2022 IEEE Visualization and Visual Analytics (VIS)*, pp. 21–25. IEEE, New York, 2022. doi: 10.1109/VIS54862.2022.00013 2

[40] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985. doi: 10.1145/3147.3165 1, 2

[41] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Trans. Image Process.*, 13(4):600–612, 2004. doi: 10.1109/TIP.2003.819861 6

[42] B. Yadranjiaghdam, S. Yasrobi, and N. Tabrizi. Developing a real-time data analytics framework for twitter streaming data. In *2017 IEEE International*

Congress on Big Data (BigData Congress), pp. 329–336. IEEE, New York, 2017. doi: 10.1109/BigDataCongress.2017.49 1

[43] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016. doi: 10.1145/2934664 2

[44] A. Zakrzewska and D. A. Bader. Streaming graph sampling with size restrictions. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, pp. 282–290. Association for Computing Machinery, New York, 2017. doi: 10.1145/3110025.3110058 2

[45] G. D. Zion and B. K. Tripathy. *Comparative Analysis of Tools for Big Data Visualization and Challenges*, pp. 33–52. Springer Singapore, Singapore, 1$^{\text{nd}}$ ed., 2020. doi: 10.1007/978-981-15-2282-6_3 1