Large-scale sensor network analysis
Applications in structural health monitoring

Joaquin Vanschoren, Ugo Vespier, Shengfa Miao, Marvin Meeng, Ricardo Cachucho, and Arno Knobbe

Abstract Sensors are increasingly being used to monitor the world around us. They measure movements of structures such as bridges, windmills and plane wings, human’s vital signs, atmospheric conditions, and fluctuations in power and water networks. In many cases, this results in large networks with different types of sensors, generating impressive amounts of data. As the volume and complexity of data increases, their effective use becomes more challenging, and novel solutions are needed both on a technical as well as a scientific level. Founded on several real-world applications, this chapter discusses the challenges involved in large-scale sensor data analysis, and describes practical solutions to address them. Due to the sheer size of the data, and the large amount of computation involved, these are clearly ‘Big Data’ applications.

Joaquin Vanschoren e-mail: joaquin@liacs.nl
Ugo Vespier e-mail: uvespier@liacs.nl
Shengfa Miao e-mail: miao@liacs.nl
Marvin Meeng e-mail: meeng@liacs.nl
Ricardo Cachucho e-mail: cachucho@liacs.nl
Arno Knobbe e-mail: knobbe@liacs.nl

LIACS, University of Leiden, Leiden, The Netherlands
Large-scale sensor network analysis

Joaquin Vanschoren, Ugo Vespier, Shengfa Miao, Marvin Meeng, Ricardo Cachucho, and Arno Knobbe

1 Introduction

Sensors are increasingly being used to monitor the world around us. They measure movements of structures such as bridges, windmills and plane wings, vital signals of humans, atmospheric conditions, and fluctuations in power and water networks. In many cases, this results in large networks with different types of sensors, generating impressive amounts of data. In this chapter, we look at a specific case in considerable detail - a sensor network attached to a large highway bridge - and demonstrate what generic techniques may be required to analyse this data [11, 19].

1.1 Bridge monitoring

Structural health monitoring, i.e. the monitoring of the behaviour of man-made structures such as bridges, tunnels and buildings, is an important case for large-scale sensor data analysis [11]. Many existing structures are now past their design lifetime, and complex degradation processes such as corrosion and excessive load cycles can sometimes cause unpredictable behaviour. Monitoring their dynamic behaviour is key in understanding how fast they are degrading and predicting when critical maintenance, or even replacement, is needed. This requires the infrastructure to be equipped with a network of sensors, continuously measuring and collecting various structural and climate features such as vibration, strain and weather conditions. This continuous measuring process generates a massive amount of streaming data which must be analyzed over large time intervals.

One such structure is the ‘Hollandse Brug’ (Holland Bridge), one of the busiest highway bridges in the Netherlands. The bridge was opened on June 1969, but on April 2007 inspection measurements showed that the bridge did not meet quality and security requirements. Subsequently, it was closed in
both directions to freight traffic, and most of the bridge deck has since been replaced. To monitor the bridge’s behaviour, and avoid such abrupt bridge closures in the future, a sensor network was installed to learn how the bridge, and many other bridges like it, degrades over time and to use this data to plan future maintenance more accurately.

The monitoring system comprises 145 sensors that measure how the bridge responds to external forces, at several locations along its span (see Figure 1 for an illustration). The following types of sensors are employed:

- 34 ‘geo-phones’ (vibration sensors) that measure the vertical movement of the bottom of the road-deck as well as the supporting columns.
- 16 strain-gauges embedded in the concrete, measuring horizontal longitudinal stress, and an additional 34 gauges attached to the outside.
- 28 strain-gauges embedded in the concrete, measuring horizontal stress perpendicular to the first 16 strain-gauges, and an additional 13 gauges attached to the outside.
- 10 thermometers embedded in the concrete, and 10 attached on the outside.

These sensors measure at 100 Hz, yielding about 4 GB of data per day, or well over a terabyte of data per year. In addition, a weather station measures wind speeds, rainfall and solar radiation, and a video camera provides a live stream of the traffic on the bridge. This system creates a constant influx of high-throughput data. While manageable over short intervals, this data needs to be collected over many years for detailed analysis, quickly turning this fast data into big data.

A small 1-minute sample of the strain and vibration data is shown in Figure 2. One can easily distinguish the ‘bumps’ caused by passing traffic and the resulting oscillation of the bridge deck.

Analyzing such large, real-world data is wrought with difficulty. For instance, Figure 3a shows the output of one single strain sensor over the course of an entire day. One can easily distinguish events occurring on different time scales. First, narrow, sharp peaks are caused by passing vehicles of different...
Fig. 2: Short sample of strain (top) and vibration (bottom) sensor data.

weights: a detailed peak is shown in Figure 3b). Second, prolonged increases in strain are caused by traffic congestion during rush hours (9:00 and 14:00). Mini-traffic jams, lasting only a few minutes, also occur. Third, there is a subtle, gradual change in strain over the course of the day caused by changes in the outside temperature, which influences the stiffness of the concrete and hence the strain measurements. On even longer time scales (weeks and months), one will also see the effects of seasonal changes and atmospheric conditions. In Section 4, we will use scale-space filtering [12, 22] to decompose this signal into \( k \) sub-signals corresponding to events occurring at different time scales, as shown in Figure 4a. We can also build models by first labelling each interval of the strain data by the type of traffic that is passing over the sensor location at that time. In Section 3, we will use clustering to identify the different types of traffic on the bridge (e.g., car, truck, or heavy truck) and then label each part of the signal accordingly.

Moreover, these signals are measured per support girder and deck element, and they influence each other indirectly. For instance, in Figure 3a one can see peaks during traffic congestion. This is caused by traffic passing in one direction, whilst there is traffic congestion in the other direction, thus on the other side of the bridge. As such, we need to model the dependencies between signals measured at different locations. Moreover, we also need to model the dependencies between signals generated by different types of sensors. For instance, Figure 4b shows the effect of temperature on the strain data. This shows that the effect of temperature on the strain in the bridge is not linear, otherwise it would vary along a straight line. Instead, there exists a delay...
Fig. 3: (a) One day of strain measurements. The multiple external factors affecting the bridge are visible at different time scales. (b) A detail of plot (a) showing one of the peaks caused by a passing vehicle.

Fig. 4: (a) Signal (top) and top-ranked scale decomposition, respectively showing traffic, congestion, and temperature-related baseline shifts. (b) Strain readings plotted against temperature readings over the course of two days (different colors).

between a temperature change and a change in (average) strain. This can be modelled accurately using an exponential decay function, as will be discussed in Section 5.

1.2 Large-scale sensor data analysis

It will be clear that these different types of analysis require substantial computation. While some analyses build on fairly standard techniques from Digital Signal Processing, such as signal convolution or Fourier transforms, they become extremely challenging on data of the size we are dealing with in this context. To be able to support the kinds of computations described here, we reimplemented many time series analysis operations in the MapReduce framework using Hadoop [21]. This allows us to parallelize the necessary disk reads over many different computing nodes, map the required data points together, and run computations over them as if all data were read by the same node. These implementations scale linearly with the amount of data, and can be speeded up by simply adding more nodes to the cluster.
In addition, to perform real-time analysis, and issue quick alarms when something is out of the ordinary, we built a workflow system that processes the data from its raw form to the key indicators we wish to observe. It is specifically designed for streaming data, allowing individual readings from many sensors to flow through the system, be processed and combined.

In the remainder of this chapter, we will first introduce the basic concepts needed to formalise our methods in Section 2. Afterwards, we discuss different types of analysis performed on this data, including Clustering (Section 3), Multi-Scale Analysis (Section 4) and modelling the interaction between various types of sensor data (Section 5). Section 6 concludes.

2 Preliminaries

In this section we introduce notation and basic definitions for operations needed throughout this chapter. Since we are analysing sensor data, we deal with finite sequences of numerical measurements (samples), collected by observing some property of a system with a sensor, and represented in the form of time series as defined below.

Definition 1. A time series of length \( n \) is a finite sequence of values \( x = (x[1], \ldots, x[n]) \) of finite precision.\(^1\) A subsequence \( x[a : b] \) of \( x \) is defined as follows:

\[
x[a : b] = (x[a], x[a + 1], \ldots, x[b]), \quad a < b
\]

We also assume that all the considered time series have no missing values and that their sampling rate is constant.

2.1 Convolution

A fundamental operation for analysing time series data is the convolution:

Definition 2. Given a signal \( x \) of length \( N \) and a response function (kernel) \( h \) of length \( M \), the result of the convolution \( x * h \) is the signal \( y \) of length \( N \), defined as:

\[
y[t] = \sum_{j=-M/2+1}^{M/2} x[t - j] h[j]
\]

For most of our analysis, \( h \) is a Gaussian kernel with mean \( \mu = 0 \), standard deviation \( \sigma \), area under the curve equal to 1, discretized into \( M \) values.\(^2\) Also,\(^3\)

\(^1\) 32-bit floating point values in our experiments.

\(^2\) To capture almost all non-zero values, we define \( M = [6\sigma] \).
since \( x \) is finite, \( x[t - j] \) may be undefined. To account for these boundary effects, \( x \) is padded with \( M/2 \) zeros before and after its defined range. A complete overview on how to compute the Gaussian convolutions for discrete signals can be found in Lindeberg (1990) \[12\].

The convolution acts as a smoothing filter which smooths each value \( x[t] \) based on its surrounding values. The amount of removed detail is directly proportional to the standard deviation \( \sigma \) (and thus \( M \)), from now on referred to as the scale parameter. This is illustrated in Figure 5. It shows the calibrated, but still noisy sensor data, and the result of two convolutions with Gaussian kernels of different widths. The wider kernel (k1) removes a lot of detail, only preserving large bumps, while the narrow kernel (k2) removes noise but still preserves small oscillations in the signal. In the limit, when \( \sigma \rightarrow \infty \), the result of the Gaussian convolution converges to the mean of the signal \( x \).

### 2.2 Discrete Fourier Transform

**Definition 3.** Given a signal \( x \) of length \( N \), the **discrete Fourier Transform (DFT)** is a series of \( N \) complex numbers \( X[k] \), each describing a sine function of given frequency:

\[
X[k] = \sum_{n=0}^{N-1} x[n] e^{-2\pi i kn/N}
\]

Conversely, the inverse Fourier transform (DFT\(^{-1}\)) is defined by

\[
x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{2\pi i kn/N}
\]

The Fourier transform converts a signal from the time domain into the frequency domain, representing the data as a sum of sine functions with different frequencies and amplitudes, as shown in Figure 6. The Fourier transform has
several useful applications. For instance, we can filter out noise by removing high- and/or low frequency components (i.e., dropping some of the sine functions), and taking the inverse Fourier transform to get us back to the time domain.

![Figure 6: Discrete Fourier transform (right) of the vibration and strain data (left). The large peak in (b) corresponds to the eigenfrequency of the bridge.](image)

The Fourier transform proves very useful to detect and remove baseline shifts in sensor data. In the frequency domain, the baseline is usually treated as a low frequency signal, and filtering out low frequency components in the signal spectrum helps to remove the baseline. The question remains which frequencies must be filtered out, i.e. do we only want to remove very gradual changes, or also spurious events such as traffic jams? There also exist very useful methods to remove baselines in the time domain [15], some of which will be discussed below.

### 2.3 MapReduce

When working with terabytes of data, loading all data in memory, e.g. to compute a Fourier transform, is out of the question, and reading the data from disk becomes a major bottleneck. For instance, reading 4TB of data at 50MB/s takes almost an entire day. However, using the MapReduce framework [1], and its open-source implementation Hadoop [21], we can parallelize
For a good understanding of our methods, we include a short, high-level introduction here.

In the MapReduce framework, data is processed in two stages, as depicted in Figure 7. The first phase, the *Map phase*, reads the input data from a distributed file system, into a set of *M splits* (input partitions) of typically 16 to 64MB. Each split is assigned to a worker node, a *Mapper*. Each mapper reads the contents of the input split, parses key/value pairs out of the input data (e.g. line numbers and a line of text, or a timestamp and sensor readings), and passes each pair to a user-defined *Map* function. The latter processes this data and emits new, *intermediate* key-value pairs, which are then partitioned into *R* partitions using a *partition function* (e.g., \(\text{hash(key)} \mod R\)) and stored on local disk. Optionally, a *Combiner* function can be written that combines intermediate key-value pairs with the same key into a new key-value pair in which the value is a combination of the initial values. This reduces traffic between the mappers and the reducers.

In the *Reduce phase*, a set of worker nodes called *Reducers* collects all partitions (from all mappers) that are assigned to them, sorts the key-value pairs based on the intermediate keys, and passes each key and the corresponding

---

\(^3\) For instance, in our setup, we use a cluster with 66 DataNode, each with 4 disks.
set of intermediate data to a user-defined Reduce function. Finally, each Reduce function emits output key-value pairs which are appended to output files.

Programs written in this framework, by implementing the Map and Reduce functions, are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of scheduling the program’s execution across a set of machines, handling machine failures, and managing the required inter-machine communication [1].

3 Subsequence Clustering

In this section, we focus on one specific type of sensor: the strain measurements such as shown in Figure 3. This data can be used to identify the type of traffic that is passing over the bridge at each point in time. While we could impose a number of classes manually, such as ‘car’, and ‘truck’, it is usually better to infer such categories from the data itself using subsequence clustering (SSC) [5, 9, 2, 7]; we first employ a sliding window technique to split the data stream in individual subsequences (observations), which can then be clustered, and labeled with the cluster they belong to.

Definition 4 (Subsequence). A subsequence $S_{p,w}$ of a time series $x = x_1, ..., x_m$ is the sequence of values $x_p, ..., x_{p+w-1}$ such that $1 \leq p \leq m - w + 1$ and the window length $w < m$.

Definition 5 (Subsequences Set). The subsequences set $D(X, w) = \{S_{i,w} \mid 1 \leq i \leq m - w + 1\}$ is the set of all the subsequences extracted by sliding a window of length $w$ over the time series $X$.

The subsequences set $D(X, w)$ contains all possible subsequences of length $w$ of a time series $X$. The aim of subsequence clustering is discovering groups of similar subsequences in $D(X, w)$.

3.1 Subsequence Clustering with k-Means

The clustering method typically used for SSC is k-Means, a well known method that, given a set of vectors $D = \{x_1, ..., x_n\}$ aims to find a partition $P = \{C_1, ..., C_k\}$ and a set of centroids $C = \{c_1, ..., c_k\}$ such that the sum of the squared distances between each $x_i$ and its associated centroid $c_j$ is minimized. In this case, the set $D$ is the set of subsequences $D(X, w)$.

The classic $k$-Means heuristic implementation looks for a local minimum by iteratively refining an initial random partition. The algorithm involves four steps:
1. (initialization) Randomly choose $k$ initial cluster prototypes $c_1, ..., c_k$ in $D$.

2. (assignment) Assign every vector $x_i \in D$ to its nearest prototype $c_j$ according to a distance measure. The classic $k$-Means uses the Euclidean distance.

3. (recalculation) Recalculate the new prototypes $c_1, ..., c_k$ by computing the means of all the assigned vectors.

4. Stop if the prototypes did not change more than a predefined threshold or when a maximum number of iterations has been reached, otherwise go back to step 2.

The intuition in SSC is that, if there are repeated similar subsequences in $X$, e.g. bumps of a specific shape, they will be grouped in a cluster and eventually become associated to an actual event of the application domain, such as a passing truck.

### 3.2 Subsequence Clustering versus Event Detection

However, this intuition does not always translate into good results. It has been shown that SSC with $k$-Means is prone to a number of undesirable behaviors that make it unsuitable for the task at hand [9, 7, 2]. Most notably, the cluster prototypes, constructed by averaging the subsequences in each cluster, do not always resemble the subsequences themselves: smooth, sinusoid-like prototypes emerge even when the original data was extremely noisy and angular. Also, different random initializations were shown to lead to completely different results.

This unintuitive behavior of SSC can be understood by considering the nature of the subsequence set $D(X, w)$ that is the outcome of the initial sliding window step. Each member of $D(X, w)$ forms a point in a Euclidean $w$-dimensional space, which we will refer to as $w$-space, illustrated in Figure 8. As each subsequence is fairly similar to its successor, the associated points in $w$-space will be quite close, and the members of $D(X, w)$ form a trajectory in $w$-space. Figure 8 shows an example of a (smoothed) fragment of strain data, and its associated trajectory in $w$-space (only two dimensions shown). Individual prototypes correspond to points in $w$-space, and the task of SSC is to find $k$ representative points in $w$-space to succinctly describe the set of subsequences, in other words, the trajectory. Figure 8 (right) also shows an example of a run of $k$-Means on this data. As the example demonstrates, the prototypes do not necessarily lie along the trajectory, as they often represent an (averaged) curved segment of it.

While most of these issues do not occur in our sensor data (see Vespier et al. [19] for a detailed analysis), the clustering does indeed result in multiple representations of what is intuitively one single event, as shown in Figure 9. Indeed, each of the two bump-shaped prototypes resembles only a fraction
Fig. 8: Two plots of the same data, showing the original data as a function of time (left), and a projection on two selected dimensions in $w$-space and the four prototypes generated by $k$-Means (red circles). Clearly, the sliding window technique creates a trajectory in $w$-space, where each loop corresponds to a bump in the original signal.

Fig. 9: Multiple representation of events. The left plot shows the prototypes computed by the classic $k$-Means. The right plot shows, in black, the portion of the data assigned to the two bump-shaped prototypes: the top sequence corresponds to the left prototype, the bottom sequence to the right prototype.

of the visible bump in the strain data. In other words, our notion of traffic event does not coincide with the Euclidean distance used in $k$-Means, which assigns a large distance to essentially quite similar subsequences.

3.3 The Snapping Distance

To resolve this issue, we introduced a novel distance function that explicitly takes the local context of a subsequence into account. Given a time series $X$ and two subsequences $S_{p,w} \in X$ and $S_{fixed}$ of length $w$, we consider not only the Euclidean distance between $S_{fixed}$ and $S_{p,w}$, but also between $S_{fixed}$ and the neighboring subsequences, to the left and to the right, of $S_{p,w}$. The
minimum Euclidean distance encountered is taken as the final distance value between \( S_{p,w} \) and \( S_{\text{fixed}} \).

Formally, given a shift factor \( f \) and a number of shift steps \( s \), we define the neighbor subsequences indexes of \( S_{p,w} \) as:

\[
NS = \{ p + \frac{f w}{s} \cdot i \mid -s \leq i \leq s \}
\]

The extent of data analyzed to the left and to the right of \( S_{p,w} \) is determined by the shift factor while the number of subsequences considered in the interval is limited by the shift steps parameter. The \textit{Snapping} distance is defined as:

\[
\text{Snapping}(S_{p,w}, S_{\text{fixed}}) = \min \{ \text{Euclidean}(S_{i,w}, S_{\text{fixed}}) \mid i \in NS \}
\]

Figure 10 illustrates the intuition behind the \textit{Snapping} distance measure in the context of \( k \)-Means clustering. When measuring the distance between each cluster centroid \( C_k \) and the subsequence \( S_{p,w} \), in the \textit{assignment} step of the \( k \)-Means algorithm, the snapping distance looks beyond the edges of the subsequence window to match the bump in the sensor data to the centroid, thus avoiding boundary effects. Moreover, we force the initialization step to choose the random subsequences such that they do not overlap in the original time series.
3.4 Scalable implementation

Given the amount of data generated by the sensor network, it is important to have a very scalable implementation of our clustering method. As in most clustering applications, the main computational bottleneck lies in calculating the (snapping) distances between every subsequence and the cluster centroids. Indeed, the snapping distance further increases the computational complexity since it also needs to take the neighbourhood of each subsequence into account, depending on the shift factor and shift steps parameters. Furthermore, since the data stream is too large to load into memory, repeated disk reads will be necessary.

This type of I/O-intensive operation calls for a MapReduce approach, so we can distribute the data reads over a cluster of machines. The resulting system, implemented in the Hadoop [21] framework, is a chain of two MapReduce phases/jobs.

In the first phase, we prepare the data for the clustering phase, as shown in Figure 11. The data is stored on disk in flat files with a timestamp and the sensor readings of the 145 sensors on each line. For faster reading, data can be reformatted to SequenceFiles, consisting of binary key/value pairs. Each mapper reads a data split, selects the sensor(s) of interest\(^4\), and links each sensor reading to \((1 + f)w\) subsequences, marked by the initial timestamp \(t_s\). Remember that \(f\) is the shift factor used for the snapping distance and \(w\) is the window length. The reducers collect the data points belonging to a given timestamp key and emit the whole subsequence as an array, including the correct lead-in and lead-out necessary to compute the snapping distances. Optionally, additional processing can be done in the reducers, such as data segmentation to reduce the number of individual readings in the subsamples, and thus speed up future computations.

One important implementation detail here is the choice of key. Using the time in milliseconds results in a poor partitioning since the 100Hz data only has a reading every 10 milliseconds. This means that the hash function will not distribute keys evenly over the reducers, and at least half of them will not receive any key-value pairs, thus sitting idle. A much better distribution can be obtained using a string key or, in this case, by dividing the time in milliseconds by 10.

As soon as the subsequences are emitted, the clustering phase can start, as shown in Figure 12. First, \(k\) non-overlapping subsequences are randomly chosen as initial cluster centroids. Each mapper then reads in part of the subsequences, measures the snapping distance to each of the cluster centroids, and emits a key-value pair consisting of the nearest centroid and the subsequence. Optionally, a combiner can be used that counts and computes the partial vector sum of all subsequences emitted by the same mapper and

\(^4\) When processing multiple sensors in parallel, extra care must be taken to distinguish the subsequences of different sensors, e.g., by adding the sensor id to the keys.
Fig. 11: The data preparation phase: sensor readings are read from file and emitted as subsequences with lead-ins and lead-outs according to the shift factor $f$ of the used snapping distance.

Fig. 12: The $k$-Means clustering phase: after choosing $k$ non-overlapping subsequences as initial cluster centroids, all subsequences are clustered and new cluster centroids are computed until convergence.

mapped to the same centroid. Next, each reducer receives all the subsequences mapped to a specific cluster, and emits the new cluster centroid, i.e., the average of all received subsequences. This map-reduce cycle is repeated $n$ times or until the clusters converge.

We were able to build upon the $k$-Means implementation present in the Mahout library\(^5\), though extended to support the more complex snapping distance.

We evaluated this implementation on two hadoop clusters: a small cluster with 6 nodes and larger one with 66 compute nodes. The runtimes are plotted in Figure 13. This shows that the runtime scales linearly with the amount

of data, and that adding more compute nodes linearly reduces the required time to compute the clusters.

### 3.5 Experimental Evaluation

To evaluate the quality of the clustering, we first need to fix a number of parameters. First, the window length $w$ has been set at 4 seconds, given that the bridge spans are 50 meters long, and at a maximum speed of 100 km/h, a typical vehicle takes in the order of 2.5 seconds to cross the span. 4 seconds thus allows to capture a complete ‘bump’ in strain data. The shift factor $f$ and shift step $s$ were experimentally determined at values 0.5 and 10, respectively.

Figure 14 depicts the results obtained by applying the $k$-Means SSC based on the Snapping distance on the same data used previously to illustrate the effect of the classic (Euclidean) $k$-Means approach in Figure 9. Comparing
Fig. 15: Prototypes produced by applying \( k \)-Means respectively with Euclidean and Snapping distance on the FullWeekDay data, for both \( k = 4 \) (left) and \( k = 10 \) (right).

both figures, it is clear that this time, the clustering correctly converges to a single bump-shaped peak, caused by a heavy passing vehicle, as well as smaller prototypes for a lighter passing vehicles and the strain baseline. On the top right, the intervals are highlighted corresponding to the small peak (light vehicles), and on the bottom right, intervals are highlighted corresponding to the big peak (heavy truck).

Using more data results in even nicer cluster prototypes, as shown in Figure 15. On longer time series, more clusters are needed to match the higher variability caused by temperature variations, more varied traffic, and traffic jams. We found \( k = 10 \) to be large enough to account for most of the interesting variations in the time series, though we also show the result with \( k = 4 \) for comparison. The third image in Figure 15 shows that, again, the classic \( k \)-Means SSC introduces double representations of the same logical events. This is avoided by using the snapping distance (right image), which captures much better different ‘states’ of the bridge, corresponding to different events. From bottom to top: the key baselines, light vehicles, heavy vehicles, traffic jam baseline and light and heavy vehicles passing during traffic jams (in the opposite direction).

4 Multi-scale analysis

As briefly discussed in the introduction, and also apparent in the clusters generated in the previous section, different ‘events’ on the bridge occur at different time scales: passing vehicles rush by in seconds, while traffic jams can take minutes or hours, and temperature gradually changes over the course of
an entire day. In order to understand the various changes in the sensor signal, one would benefit substantially from separating out the events at various scales. The analysis of time scales in time series data is often approached from a scale-space perspective, which involves convolution of the original signal with Gaussian kernels of increasing size [22] to remove information at smaller scales. By subtracting carefully selected components of the scale-space, we can effectively cut up the scale space into $k$ ranges.

4.1 The Scale-Space Image

The scale-space image [22] is a scale parameterization technique for one-dimensional signals based on the operation of convolution, discussed in Section 2.1.

Given a signal $x$, the family of $\sigma$-smoothed signals $\Phi_x$ over scale parameter $\sigma$ is defined as follows:

$$\Phi_x(\sigma) = x \ast g_\sigma, \sigma > 0$$

where $g_\sigma$ is a Gaussian kernel having standard deviation $\sigma$, and $\Phi_x(0) = x$.

The signals in $\Phi_x$ define a surface in the time-scale plane $(t, \sigma)$ known in the literature as the scale-space image [12, 22]. This visualization gives a complete description of the scale properties of a signal in terms of Gaussian smoothing. Moreover, it has other properties useful for segmentation, as we will see later in the paper.

For practical purposes, the scale-space image is quantized across the scale dimension by computing the convolutions only for a finite number of scale parameters. More formally, for a given signal $x$, we fix a set of scale parameters

$$S = \{2^i \mid 0 \leq i \leq \sigma_{max} \land i \in \mathbb{N}\}$$

and we compute $\Phi_x(\sigma)$ only for $\sigma \in S$ where $\sigma_{max}$ is such that $\Phi_x(\sigma)$ is approximately equal to the mean signal of $x$.

As an example, Figure 16 shows the scale-space image of an artificially generated signal. The topmost plot represents the original signal, constructed by three components at different temporal scales: a slowly changing and slightly curved baseline, medium term events (bumps) and short term events (peaks). It is easy to visually verify that, by increasing the scale parameter, a larger amount of detail is removed. In particular, the peaks are smoothed out at scales greater than $\sigma = 2^4$, and the bumps are smoothed out at scales greater than $\sigma = 2^8$, after which only the baseline remains.

Figure 17 shows the scale-space created by convoluting the strain signal of a single sensor with Gaussian kernels of sizes ranging from $\sigma = 2^4$ to $\sigma = 2^9$. At the smallest scales, all detail is preserved, even the noise in the data. At slightly larger scales, noise is smoothed out, but the signal still follows the
Fig. 16: Scale-space image of an artificially generated signal totalling 259200 points.

Fig. 17: Scale-space convolutions overlapping with strain sensor data.

effect of a truck’s suspension on the bridge (resulting in oscillations in the large bump itself), as well as the oscillation of the bridge after the truck has passed. At larger scales, those oscillations are smoothed out so that only the bump remains, and at the very largest scales even the bump will diminish until it is also smoothed out and only the baseline remains.
In the next section, we show how to manipulate the scale-space image to filter out the effects of transient events in a specific range of scales. This will lead to the definition of a signal decomposition scheme.

### 4.2 Scale-Space Decomposition

Along the scale dimension of the scale-space image, short-time transient events in the signal will be smoothed away sooner than longer ones. In other words, we can associate to each event a maximum scale $\sigma_{\text{cut}}$ such that, for $\sigma > \sigma_{\text{cut}}$, the transient event is no longer present in $\Phi_x(\sigma_{\text{cut}})$. This fact leads to the following two observations:

- Given a signal scale-space image $\Phi_x$, the signal $\Phi_x(\sigma)$ is only affected by the transient events at scales greater than $\sigma$. This is conceptually equivalent to a low-pass filter in signal processing.
- Given a signal scale-space image $\Phi_x$ and two scales $\sigma_1 < \sigma_2$, the signal $\Phi_x(\sigma_1) - \Phi_x(\sigma_2)$ is mostly affected by those transient events present in the range of scales $(\sigma_1, \sigma_2)$. This is similar to a band-pass filter in signal processing.

As an example, reconsider the signal $x$ and its scale-space image $\Phi_x$ of Figure 16. Figure 18 shows (from top to bottom):

- the signal $\Phi_x(0) - \Phi_x(2^4)$, which is the result of a high-pass filtering; this feature represents the short-term events (peaks),
- the signal $\Phi_x(2^4) - \Phi_x(2^{10})$, which is the result of a band-pass filtering; this feature represents the medium-term events (bumps),
- the signal $\Phi_x(2^{10})$, which is the result of a low-pass filtering; this feature represents the long-term trend.

Generalizing the example in Figure 18, we can define a decomposition scheme of a signal $x$ by considering adjacent ranges of scales of the signal scale-space image. We formalize this idea below.

![Fig. 18: Examples of signal decomposition obtained from the scale-space image in Figure 16.](image-url)
Definition 6. Given a signal $x$ and a set of $k - 1$ scale parameters $C = \{\sigma_1, \ldots, \sigma_{k-1}\}$ (called the cut-points set) such that $\sigma_1 < \ldots < \sigma_{k-1}$, the scale decomposition of $x$ is given by the set of component signals $D_x(C) = \{x_1, \ldots, x_k\}$, defined as follows:

$$x_i = \begin{cases} 
& \Phi_x(0) - \Phi_x(\sigma_1) \quad \text{if } i = 1 \\
& \Phi_x(\sigma_{i-1}) - \Phi_x(\sigma_i) \quad \text{if } 1 < i < k \\
& \Phi_x(\sigma_{k-1}) \quad \text{if } i = k 
\end{cases}$$

Note that for $k$ components we require $k - 1$ cut-points. This decomposition has several elegant properties:

- $x_k$ can be seen as the baseline of the signal, as obtained by a low-pass filter;
- $x_i$ for $1 < i < k$ are signals as obtained by a band-pass filter, and can be used to identify transient events;
- $\sum_{i=1}^{k} x_i = x$, i.e., the original signal can be recovered from the decomposition.

4.3 Convolution in MapReduce

Computing the scale-space image and the scale-space decompositions are quite expensive operations when dealing with terabytes of sensor data. Moreover, our data streams are too large to fit in memory, so we will need to break them up into data splits and convolve each split separately. Again, the bottleneck will be the disk reads, so we again resort to the MapReduce framework for computing the convolutions and scale-space images. To simplify things, we start with convolution by itself.

In its simplest form, each mapper could just read a data split, do the convolution and output the convolved data (the reducer being simply an
identity reducer). However, this will not be correct, because at both ends of each data split, we don’t have the necessary data to calculate the convolution. This is also called the wrap-around problem, and is illustrated in Figure 19. Indeed, at the borders of the data split, the Gaussian kernel \( g_\sigma \) does not overlap the signal completely. While we could wrap the kernel around the edge, and take a portion of the right side of the signal while we are convoluting the very left side, this will result in a section of spoiled data.

To compute the convolution in practice, we will cut off the Gaussian kernel \( g_\sigma \) at some point, e.g. at \( 3\sigma \) on each side. That means that the first and the last \( \frac{3}{2}\sigma \) points of the convolution will be spoiled. If we use very wide Gaussians, e.g. to establish the baseline component, this will result in a lot of spoiled data.

A simple solution is to pad the signal with zeros on both ends, as shown in Figure 20. The resulting convolution will still include spoiled intervals, but now they correspond to the zero-padded intervals, so we can safely ignore them.

While this works for the very beginning and end of our sensor data, it is not correct for most data splits: indeed, the beginning and end of each split should be influenced by the previous and next split, respectively, not by zeros. Thus, these data points should somehow be appended to each data split. One solution would be to write a custom Hadoop InputFormat to generate data splits in such a way that they overlap. This is indeed a (nontrivial) solution, but what if the data is physically stored in separate files, or even on different machines?

Another solution is to activate our idle reducers. This is illustrated in Figure 21. The mappers each read a data split, and output it in as an array, again using the first timestamp of the split as the key. However, for the first and last \( \frac{3}{2}\sigma \) points, it will also emit them to the previous and next data split, respectively. An additional indicator will be sent with the data to indicate whether it is a head, body, or tail section of the enlarged split. The reducers thus receive the 3 components, can compute the convolution, and only emit
the unspoiled ‘body’ part of each split. At the very beginning and end of the signal, we still use zero padding.

However, zero-padding can also be used in a different way, illustrated in Figure 22. Here, the mappers read their data split, append zero's at the beginning and each of it, and emit the entire convolution. This will obviously result in ‘spoiled’ regions. However, if you have carefully zero-padded both ends, there is no ambiguity and you can reconstruct the convolution in that region by simply adding the convoluted values of both mappers in the reducer. Each mapper has effectively computed part of the convolution.

There is another, faster method to compute the convolution: the Fast Fourier Transform. This is a result of the discrete convolution theorem: if a signal $x$ is periodic with period $N$, so that it is completely determined by the $N$ values $x_0, x_{N-1}$, then its discrete convolution with a response function $h$ of finite duration $N$ can be obtained as the inverse DFT of the product of the individual transforms.

$$x * h = \sum_{k=-N/2+1}^{N/2} x_{t-k} h_k = DFT^{-1}\{X_n H_n\}$$
Here, $X_n(n = 0, N - 1)$ is the discrete Fourier transform of the signal $x_t(t = 0, N - 1)$, while $H_n(n = 0, N - 1)$ is the discrete Fourier transform of the response function $h_t(t = 0, N - 1)$. Thus, in simple terms, $x \ast h = DFT^{-1}\{DFT(x)DFT(h)\}$.

Because our data is not periodic, we face the same wrap-around problem, but we can work around this problem by using the zero-padding discussed above and shown in Figure 20. Second, our response function $g_\sigma$ is much shorter than $N$, but again this is solved by padding it with zeros.

All this is interesting because, while both the direct convolution and the DFT are $O(n^2)$, there exist several implementations for the parallel Fast Fourier Transform (FFT) that take $O(n \log n)$ time. However, their efficient implementation in MapReduce is non-trivial [18].

In our experiments, doing the convolution directly using MapReduce turned out to be sufficient. Indeed, our response function $g_\sigma$ is often much smaller than $N$, approaching $\log n$. Moreover, when computing the scale space image, we can compute multiple convolutions in parallel. Each mapper can compute $g_\sigma$ for several or all values of $\sigma$, as long as it employs the exact amount of zero-padding for each convolution, and emits the results using a key that includes the value of $\sigma$ used, so that the reduces can combine the correct partial convolutions.

Efficiently implemented, our MapReduce implementation of the direct convolution can smooth 3 months of data in only a few minutes. In fact, writing the results to disk takes longer than the actual computations. The time to compute large scale spaces also grows linearly in the length of the time series and the size of the longest considered time scale.

\subsection*{4.4 MDL Scale Decomposition Selection}

In order to construct a scale-space decomposition, such as shown in Figure 4(a), that can be used to adequately model the bridge’s behaviour, we still need to choose several key parameters: the appropriate scales at which to model the components, and the optimal number of components. More formally, we need to find a good subset of cut-points $C$ such that the resulting $k$ components of the decomposition $D_x(C)$ optimally capture the effect of transient events at different scales.

The rationale behind the scale decomposition is that it is easier to model the effect of a single class of transient events at a given scale than to model the superimposition of many, interacting transient events at multiple scales. We thus need to trade off the added complexity of having to represent multiple components for the complexity of the representations themselves. A good approach for establishing this trade-off is the Minimum Description Length (MDL) principle.
The Minimum Description Length [3] is an information-theoretic model selection framework that selects the best model according to its ability to compress the given data. In our context, the two-part MDL principle states that the best model \( M \) to describe the signal \( x \) is the one that minimizes the sum \( L(M) + L(x \mid M) \), where

- \( L(M) \) is the length, in bits, of the description of the model,
- \( L(x \mid M) \) is the length, in bits, of the description of the signal when encoded with the help of the model \( M \).

In order to apply the MDL principle, we need to define a model \( M_{D_x(C)} \) for a given scale decomposition \( D_x(C) \) and, consequently, how to compute both \( L(M_{D_x(C)}) \) and \( L(x \mid M_{D_x(C)}) \). The latter term is the length in bits of the information lost by the model, i.e., the residual signal \( x - M_{D_x(C)} \).

### 4.4.1 Time Series Values Discretization

In order to use the MDL principle we need to work with a quantized input signal and scale-space image. Because of this, we assume that the values \( v \) of both the input signal \( x \) and \( \Phi_x(\sigma) \), for each considered \( \sigma \), have been quantized to a finite number of symbols by employing the function defined below:

\[
Q(v) = \left\lfloor \frac{v - \min(x)}{\max(x) - \min(x)} l \right\rfloor - \frac{l}{2}
\]

where \( l \), assumed to be even, is the number of bins to use in the discretization while \( \min(x) \) and \( \max(x) \) are respectively the minimum and maximum value in \( x \). Throughout the rest of the paper, we assume \( l = 256 \). A similar approach is described in [6]. All the subsequent operations, from the computations of the scale decompositions to the encoding of the components, are kept in this quantized space.

### 4.4.2 Component Modelling

Next, we also need a model to describe each of the decomposition components and calculate the length \( L(M) \) of the model. In the next paragraphs we introduce two such methods, illustrated in Figure 23: the first is based on discretisation, the second on segmentation.

Discretization-based representation

A desirable encoding would giving short codes to long stretches of baseline and the commonly occurring amplitudes. Unfortunately, our original discretization is too fine-grained to capture regular occurrences of similar
amplitudes. Hence, we consider a more coarse-grained discretization by discretizing each value $v$ in a component to a value $\lfloor Q(v)/2^i \rfloor$, where several values for $i$ are considered for each component, typically $i \in \{2, 4, 6\}$. By doing so, similar values will be grouped together in the same bin. The resulting sequence of integers is compacted further by performing run-length encoding, resulting in a string of $(v, l)$ pairs, where $l$ represents the number of times value $v$ is repeated consecutively. This string is finally encoded using a Shannon-Fano or Huffman code (see Section 4.4.3).

Segmentation-based representation

The main assumption on which we base this method is that a clear transient event can be accurately represented by a simple function, such as a polynomial of a bounded degree. Hence, if a signal contains a number of clear transient events, it should be possible to accurately represent this signal with a number of segments, each of which represented by a simple function.

Given a component $x_i$ of length $n$, let

$$z(x_i) = \{t_1, t_2, ..., t_m\}, \quad 1 < t_i \leq n$$

be a set of indexes of the segment boundaries.

Let $\text{fit}(x_i[a:b], d_i)$ be the approximation of $x_i[a:b]$ obtained by fitting a polynomial of degree $d_i$. Then, we represent each component $x_i$ with the approximation $\hat{x}_i$, such that:

$$\hat{x}_i[0:z_1] = \text{fit}(x_i[0:z_1], d_i)$$
$$\hat{x}_i[z_i : z_{i+1}] = \text{fit}(x_i[z_i : z_{i+1}], d_i), \quad 1 \leq i < m$$
$$\hat{x}_i[z_m : n] = \text{fit}(x_i[z_m : n], d_i)$$

Note that approximation $\hat{x}_i$ is quantized again by reapplying the function $Q$ to each of its values.
For a given \(k\)-components scale decomposition \(D_\mathbf{x}(C)\) and a fixed polynomial degree for each of its components, we calculate the complexity in bits of the model \(M_{D_\mathbf{x}(C)}\), based on this representation scheme, as follows. Each approximated component \(\hat{\mathbf{x}}_i\) consists of \(|z(\mathbf{x}_i)| + 1\) segments. For each segment, we need to represent its length and the \(d_i + 1\) coefficients of the fitted polynomial. The length \(ls_i\) of the longest segment in \(\hat{\mathbf{x}}_i\) is given by

\[
ls_i = \max(z_1 \cup \{z_{i+1} - z_i \mid 0 < i \leq m\})
\]

We therefore use \(\log_2(ls_i)\) bits to represent the segment lengths, while for the coefficients of the polynomials we employ floating point numbers of fixed\(^6\) bit complexity \(c\). The MDL model cost is thus defined as:

\[
L(M_{D_\mathbf{x}(C)}) = \sum_{i=1}^{k} (|z(\mathbf{x}_i)| + 1) (\lceil\log_2(ls_i)\rceil + c(d_i + 1))
\]

So far we assumed to have a set of boundaries \(z(\mathbf{x}_i)\), but we did not specify how to compute them. A desirable property for our segmentation would be that a segmentation at a coarser scale does not contain more segments than a segmentation at a finer scale.

The scale space theory assures that there are fewer zero-crossing of the derivatives of a signal at coarser scales [22]. In our segmentation we use the zero-crossings of the first and second derivatives. These can be computed efficiently, because the derivative of the convolution of a signal \(\mathbf{x}\) is the same as the convolution of \(\text{textbf{f}}\) with the derivative of the response function \(g_\sigma\).

More formally, we define the segmentation boundaries of a component \(\mathbf{x}_i\) to be

\[
z(\mathbf{x}_i) = \left\{ t \in \mathbb{R} \mid \frac{d\mathbf{x}_i}{dt}(t) = 0 \right\} \cup \left\{ t \in \mathbb{R} \mid \frac{d^2\mathbf{x}_i}{dt^2}(t) = 0 \right\}.
\]

Figure 23b shows an example of segmentation obtained as above using fitted polynomials of degree 1. However, many other segmentation algorithms are known in the literature [8, 10] and all of them can be interchangeably employed in this context.

### 4.4.3 Residual Encoding

Given a model \(M_{D_\mathbf{x}(C)}\), its residual \(\mathbf{r} = \mathbf{x} - \sum_{i=1}^{k} \hat{\mathbf{x}}_i\), computed over the components approximations, represents the information of \(\mathbf{x}\) not captured by the model. Having already defined the model cost for the two proposed encoding schemes, we only still need to define \(L(\mathbf{x} \mid M_{D_\mathbf{x}(C)})\), i.e., a bit complexity \(L(\mathbf{r})\) for the residual \(\mathbf{r}\).

\(^6\) In our experiments \(c = 32\).
Here, we exploit the fact that we operate in a quantized space; we encode each bin in the quantized space with a code that uses approximately \(-\log(P(x))\) bits, where \(P(x)\) is the frequency of the \(x\)th bin in our data. The main justification for this encoding is that we expect that the errors are normally distributed around 0. Hence, the bins that reflect a low error will have the highest frequency of occurrences; we will give these the shortest codes. We use Huffman coding in our experiments, as in Hu et al. [6].

4.4.4 Model Selection

We can now define the MDL score that we are optimizing as follows:

**Definition 7.** Given a model \(M_{D_k(C)}\), its **MDL score** is defined as:

\[
L(M_{D_k(C)}) + L(r)
\]

In the case of discretization-based encoding, the MDL score is affected by the cardinality used to encode each component. In the case of segmentation-based encoding the MDL score depends on the boundaries of the segments and the degrees of the polynomials in the representation. In both cases, also the cut-points of the considered decomposition affect the final score.

The simplest way to find the model that minimizes this score is to enumerate, encode and compute the MDL score for every possible scale-space decomposition and all possible encoding parameters. As we shall now show, this brute-force approach is practically feasible.

The number of possible scale decompositions depends on the total number of cut-points sets we can build from the computed scale parameters in \(\Phi_x\). We fix the maximum number of cut-points in a candidate set to some value \(c_{\text{max}}\). This also means that we limit our search to those scale decompositions having \(c_{\text{max}} + 1\) components or less. Moreover, given our wish to consider only simple approximations of the signals, we can also assume a reasonably low limit \(d_{\text{max}}\) (in practice, \(d_{\text{max}} = 2\)) on the degree of the polynomials that approximate the segments of each given component.

Computing the MDL score for each encoded scale decomposition, obtained by ranging over all the possible configurations of cut-points \(C_1, \ldots, C_{k-1}\), and all the possible configurations of polynomial degrees \(d_1, \ldots, d_k\), hence requires calculating MDL scores for

\[
\sum_{k=2}^{c_{\text{max}}+1} \binom{|S|}{k-1} d_{\text{max}}^k
\]

scale decompositions. This turns out to be a reasonable number in most practical cases we consider, and hence we use an exhaustive approach in our experiments.
4.5 Experiments

In this section, we experimentally evaluate this method on real-world sensor data. For a more detailed analysis of the MDL-based approach on artificial data, see Vespier et al. [20].

Figure 24 shows strain data from a single sensor, and the resulting decomposition into three scale-space components. We evaluated all the possible decompositions up to three components (two cut-points) allowing both the discretisation and segmentation representation schemes. In the case of the discretization-based representations, we limit the possible cardinalities to 4, 16 and 64.

The top-ranked decomposition results in 3 components as shown in the last three plots in Figure 24. The selected cut-points appear at scales $2^6 = 64$ and $2^{11} = 2048$. All three components are represented with the discretization-based scheme, with a cardinality of respectively 4, 16, and 16 symbols. The decomposition has an MDL-score of 344,276, where $L(M) = 19,457$ and $L(D | M) = 324,818$. The found components accurately correspond to physical events on the bridge. The first component, covering scales lower than $2^6$, reflects the short-term influence caused by passing vehicles and represented as peaks in the signal. Note that the cardinality selected for this component is the lowest admissible in our setting (4). This is reasonable considering that the relatively simple dynamic behavior occurring at these scales, mostly the presence or not of a peak over a flat baseline, can be cheaply described with 4 or fewer states without incurring a too large error. The middle component, covering scales between $2^6$ and $2^{11}$, reflects the medium-term effects caused by traffic jams. As in the artificial data, the first component is slightly influenced by the second one, especially at the start and ending points of a traffic jam. Finally, the third component captures all the scales greater than $2^{11}$, here representing the effect of temperature during a whole day. To sum up, the top-ranked decomposition successfully reflects the real physical phe-
Fig. 25: Correlation matrices for Strain-Temperature (left), Strain-Vibration (middle) and Strain-Vibration after band-pass filtering (right, see Section 5.2). The numbers on the axes indicate the sensor number.

5 Modelling dependencies

When dealing with multiple sensors measuring a physical system, each individual sensor will be sensitive to some aspects of the system, based on the specific characteristics of the type of sensor and on which part of the system the sensor is placed. This is clearly the case for sensors of different types (such as vibration and temperature sensors), but also for identical sensors attached differently to the system. In this section, we focus on modelling dependencies between different types of sensors on high volumes of data.

5.1 Strain & Temperature

First, we study the relationship between strain and temperature sensors. Figure 25 (left) shows the absolute correlation coefficients between strain and temperature readings, varying from 0 to 0.97. For these sensor pairs with high correlation coefficients, we can simply employ a linear model that assumes the measured strain is directly influenced by the temperature of one of the temperature sensors:

\[ S = a \cdot T + b \]
In this model, the coefficients $a$ and $b$ translate between the temperature scale (in Celsius) and the micro-strain scale (in $\mu m/m$), and can be obtained through linear regression over a significant portion of the data streams of both sensors. Figure 26(left) shows the accuracy of this model for two sensor time series that are only moderately related, with $a = -3.288$ and $b = 27.547$. The correlation coefficient for this example is $r = 0.776$, which indicates that the selected pair of sensors are moderately correlated. This mismatch is caused by the fact that the bridge does not immediately react to a change in the outside temperature, but rather does so in a delayed fashion.

The amount of delay depends on the size and material of the structure, with larger structures (such as the bridge in question) being less sensitive to sudden changes of outside temperature. In other words, a large concrete bridge has a large capacity to store heat, which is mirrored in a slow response of the strain signal.

In the systems analysis field, systems with a capacity are often modelled as a Linear Time-Invariant system [4]. *Time-invariant* systems do not change over time, which is a reasonable assumption for a bridge, if subtle deterioration of the structure is ignored. LTI systems are *linear* because their ‘output’ is a linear combination of the ‘inputs’. In this case, the temperature of the bridge is modelled as a linear combination of the outside temperature over a certain period of time (typically the recent temperature history):

$$T_{bridge}(t) = \sum_{m=0}^{\infty} h(m)T(t-m)$$

where $T_{bridge}(t)$ is the internal temperature and $h$ is an impulse response (to be defined below). Note that this is a special case of *convolution*, which means we can reuse our methods for computing convolution over large amounts of data.

Of the many impulse response functions $h$, which include for example the well-known moving average operation, we decide to model the delayed effect of
the outside temperature using the exponential decay function $h_e(m) = e^{-\lambda m}$ (for $m \geq 0$). In this function, $\lambda$ is the decay factor, which determines how quickly the effect of past values reduces with time. Note that the resulting equation

$$S = a \cdot h_e \ast T + b, \text{ where } h_e(m) = e^{-\lambda m}$$

(2)

is the solution to a linear differential equation that is known as Newton’s law of cooling, which states that the change in temperature of the bridge is proportional to the difference between the temperature of the bridge and its environment:

$$\frac{dT_{\text{bridge}}}{dt} = -r \cdot (T_{\text{bridge}}(t) - T(t))$$

This is a somewhat simplified representation of reality, in that it assumes that the system consists of two ‘lumps’, the bridge and the environment, and that within each lump the distribution of heat is instantaneous. Although in reality this is clearly not the case, this model performs fairly well in practice.

For a given pair of sensors and the associated data, we will have to choose optimal values for $a$, $b$ and $\lambda$. It turns out that $\lambda$ behaves very decently, with only a single optimum for given $a$ and $b$, such that simple greedy optimisation will produce the desired result.

Figure 26 (right) shows the fitted model using Equation 2, which clearly demonstrates that the exponential decay model has removed the apparent delay in the data. The fitted coefficients were $a = -12.147$, $b = 30.463$, and $\lambda = 3 \cdot 10^{-5}$, yielding a correlation coefficient $r = 0.867$. Considering every possible pair of strain-temperature sensors, we find that the correlation coefficients of 47.4% of sensor pairs are improved by the exponential decay model.
5.2 Strain & Vibration

As illustrated in Figure 25 (middle), the correlations between strain and vibration sensor pairs are quite weak, the highest one for this data being 0.1557. A sensor pair with moderate correlation coefficient is shown in Figure 27a. The graphs show that the vibration sensor is a symmetric signal, while the strain sensor time series is not. However, the peaks in both occur consistently, which indicates that they are related. A simple correlation will however not be able to capture this similarity.

In order to extract the amplitude of the vibration signal, which should correspond to the magnitude of the strain on the bridge, we apply an envelope operation. We approximate the envelope by detecting peaks (using zero-crossings as discussed in Section 4.4.2) and interpolating between them. For negative peaks, we simply take the absolute value. The peaks are then interpolated with a piece-wise linear approximation. The result is demonstrated in Fig 28.

Figure 27b features the spectra obtained for the two signals by means of a Discrete Fourier Transform [17], showing that despite a lack of a direct relation in the time domain, the signals are actually fairly similar in parts of the spectrum, notably on frequencies above 1 Hz. Note the big peak around 2.8 Hz in both spectra. In fact, what is missing in the vibration spectrum are the lower frequencies, which correspond to slower bridge movements. In other words, the vibration sensors are not sensitive to gradual changes in the deflection of the bridge, as the sensors themselves simply move along with the bridge. The strain gauges, on the other hand, are sensitive even to the slowest changes in bridge deflection. However, both sensors measure shaking of the bridge (frequencies above 1 Hz) in a similar fashion.
Based on these observations, an obvious way to relate strain to vibration is to focus on a fairly specific range of frequencies. As we have done in the previous section, we can apply a band-pass filter (BPF) to remove all components of the signal outside the range 2.0 – 3.2 Hz. The linear model between the strain and vibration time series then becomes:

$$BPF_{2-3.2}(S) = a \cdot BPF_{2-3.2}(V) + b$$

After applying the band-pass filter operation to both strain and vibration data, the correlation coefficient improves from 0.10 to 0.94.

The model we achieved through the band-pass filter operation works well for a small selection of sensor pairs. In Figure 25 on the right, information is displayed on which sensor pairs specifically gain from this operation. Note that some strain gauges correspond well to most of the vibration sensors (dark columns in the matrix). These sensors are primarily located on the right side of the bridge. The few exceptions (St78, St79 and St83) are located on the girder entirely on the other side of the bridge. We look into such observations in more detail below.

### 5.3 Vibration & Temperature

As mentioned in the previous section, the vibration spectrum shows little activity in the range below 1 Hz, which happens to be where all of the temperature changes occur (for example due to the daily difference between day and night). For this reason, we do not expect significant dependencies between the sensors from Vi and Te. However, the vibration of the bridge does depend on the temperature. It is well known that bridges tend to oscillate at
specific frequencies, and that these frequencies are determined by the stiffness of the structure, which in turn is influenced by changes in the temperature of the material. In a simplified model of a span of the bridge, an Euler-Bernoulli beam [24], the natural frequency of the span is computed as follows:

\[
f_n = \frac{1}{2\pi} \sqrt{\frac{k}{m}}
\]

In this equation, \(m\) refers to the mass of the bridge (including the possible load on the bridge), and \(k\) is a stiffness coefficient that depends on several factors such as material, humidity, corrosion, etc., but also on temperature. Note that an increasing temperature leads to a decreasing stiffness \(k\), and hence a decrease in frequency, such that we expect a negative relationship between vibration and temperature sensors.

The effect of temperature on natural frequencies is widely studied [16, 23]. After external excitation, for example traffic or wind, a bridge can vibrate in different modes [14]. Each mode stands for one way of vibration, which can be vertical, horizontal, torsional or more complicated combinations thereof, and there is one natural frequency corresponding to each.

Figure 29a shows the modes corresponding to free vibration of the bridge (when no traffic is present). We can easily detect several interesting modes, summarised in Table 1. A mode ‘occurs’ in a given spectrum (corresponding to a traffic-free section of vibration data) if there is at least one peak with amplitude bigger than the average. Mode 2 (2.69 Hz) and mode 3 (2.88 Hz) are the principal modes of the bridge, which occur in every event. Mode 4 and mode 5 are also important modes, which have strong amplitude and happen in most events. Mode 1 and mode 8 have modest occurrence, but their amplitudes are relatively weak. Mode 6 and mode 7 are so weak that they can be ignored in most cases.

For a short period, we can assume the stiffness of the bridge as a constant. The only factor influencing the natural frequencies is the mass of the bridge.
When the truck is on the bridge, the mass of the bridge increases, and the natural frequencies should decrease. From Figure 29b, we can see that mode 2, mode 3, mode 4 and mode 5 indeed show a left shift of the peaks. Furthermore, the spectrum for passing trucks contains more peaks than that of free vibration, which can be explained by the complex interaction between the trucks’ suspension systems and the bridge. Further details can be found in [13].

6 Conclusions

In this chapter, we have introduced novel techniques for large-scale sensor network analysis. We have especially focussed on subsequence clustering, multi-scale analysis, scale-space decomposition, and modelling the interaction between various types of sensors in a network. For these methods we have offered scalable, MapReduce-based implementations, typically by focussing on the key underlying types of processing such as convolution, smoothing, computing derivatives and Fourier transforms. We also presented improvements on subsequence clustering techniques that are useful for all unlabeled time series applications, and introduced novel methods to automatically decompose complex sensor signals into subcomponents that correspond to events occurring on different temporal scales.

While we focussed on a specific application, a sensor network attached to a major highway bridge, these techniques will likely be useful in many other large-scale time series applications, as they often depend on the same basic building blocks such as convolution, segmentation, clustering and classification. On the other hand, some aspects are also of specific use to structural health monitoring, such as modelling physical interactions between temperature, vibration and strain, and monitoring key structural aspects such as vibration modes over long periods of time.

References