

IMPERIAL COLLEGE LONDON

DEPARTMENT OF MATHEMATICS

MSci RESEARCH PROJECT

---

# Signature Kernel Principal Components for Multivariate Time Series Compression

---

*Author:*

Daniel Zhang

*Supervisor:*

Dr. Cristopher Salvi

Submitted in partial fulfillment of the requirements for the MSci in Mathematics at  
Imperial College London

# Contents

<b>1</b>	<b>Background</b>	<b>5</b>
1.1	Principal Component Analysis . . . . .	5
1.1.1	Time Series as Vectors . . . . .	6
1.2	Kernel Principal Component Analysis . . . . .	7
<b>2</b>	<b>Literature Review</b>	<b>9</b>
2.1	Dictionaries . . . . .	9
2.2	Sequential Algorithms . . . . .	9
2.3	Functional Approximation . . . . .	9
2.4	Autoencoders . . . . .	9
<b>3</b>	<b>Proposed Method</b>	<b>10</b>
3.1	The Signature Transform . . . . .	10
3.1.1	Uniqueness . . . . .	10
3.1.2	Practical Considerations . . . . .	11
3.2	Signature Kernels . . . . .	12
3.3	Reproducing Kernel Hilbert Spaces . . . . .	14
3.3.1	Universality and Characteristicness . . . . .	14
3.3.2	Compression with Signature Kernel PCA . . . . .	15
3.4	Reconstruction . . . . .	17
3.5	Redundancies . . . . .	17
3.6	Sparse Signature . . . . .	18
<b>4</b>	<b>Experiments</b>	<b>19</b>
4.1	Eigenvalue Scree Plots . . . . .	20
4.1.1	Path Length . . . . .	20
4.1.2	Path Dimension . . . . .	22
4.1.3	Signature Depth . . . . .	23
4.2	Compression Ratio and Recovery Accuracy . . . . .	24
4.2.1	Signature Inversion with Gradient Descent . . . . .	24
4.2.2	Sparse Signature Inversion with Gradient Descent . . . . .	25
<b>5</b>	<b>Conclusion</b>	<b>26</b>
<b>6</b>	<b>Bibliography</b>	<b>27</b>
<b>7</b>	<b>Appendix</b>	<b>29</b>
7.1	Definitions . . . . .	29
7.2	Code . . . . .	30
7.2.1	Eigenvalue Scree Plots . . . . .	30
7.2.2	Signature Inversion with Standard Gradient Descent . . . . .	39
7.2.3	Sparse Signature Inversion with Gradient Descent . . . . .	42

## **Acknowledgements**

I would like to thank Dr. Cristopher Salvi for all his guidance and support throughout this project. I would also like to thank my friends and family for their continued support throughout my degree.

## Abstract

We present a novel compression technique for multivariate time series that produces a compressed representation whose size is independent of the sequence length and does not rely on regularly spaced sampling intervals. The work begins with a review of existing compression techniques, alongside the theoretical foundations of principal component analysis, signature transforms and reproducing kernel Hilbert spaces. We then develop the mathematical framework underpinning our method and empirically validate its effectiveness on synthetic and real world datasets.

## Introduction

In many real world applications, data arrives sequentially: over time, more data becomes available, and often the entire sequence of measurements is needed to extract meaningful insights. For example, the evolving patterns in a patient’s heart rate, temperature, and oxygen saturation can provide early indicators for sepsis onset.

Smart objects, like thermostats, Ring cameras and voice assistants are becoming increasingly common, forming a broader network known as the Internet of Things (IoT) [1]. These devices generate vast volumes of time-stamped data, creating a demand for efficient storage and transmission. For example, the Neuralink Compression Challenge 2024 involves 200Mbps of raw neural recordings from a non-human primate, captured from 1024 electrodes and must be compressed into a stream transmittable at under 1Mbps in real time, so the compression ratio must be at least 200. [2].

Although compression techniques already exist for both univariate and multivariate time series, for many of them, the compression size increases with the length of the time series. The compression method proposed in this thesis is independent of length, and there is less trade-off between compression size and recovery accuracy. Furthermore, it does not require the data to be sampled at regular time intervals. A key ingredient in our method is a collection of iterated path integrals known as the signature. Although the signature has already been applied to sound compression, a special case of time series data, [3], it does not initially reduce dimensionality using principal components.

## Background

### 1.1 Principal Component Analysis

Principal Component Analysis (PCA) is a dimensionality reduction technique that projects high-dimensional data onto a lower-dimensional subspace while preserving as much variance as possible. Given a dataset  $x_1, \dots, x_n \in \mathbb{R}^p$ , PCA finds orthonormal directions  $\{u_1, \dots, u_k\}$ , known as principal components. An example is shown below, with  $p = 3$ .

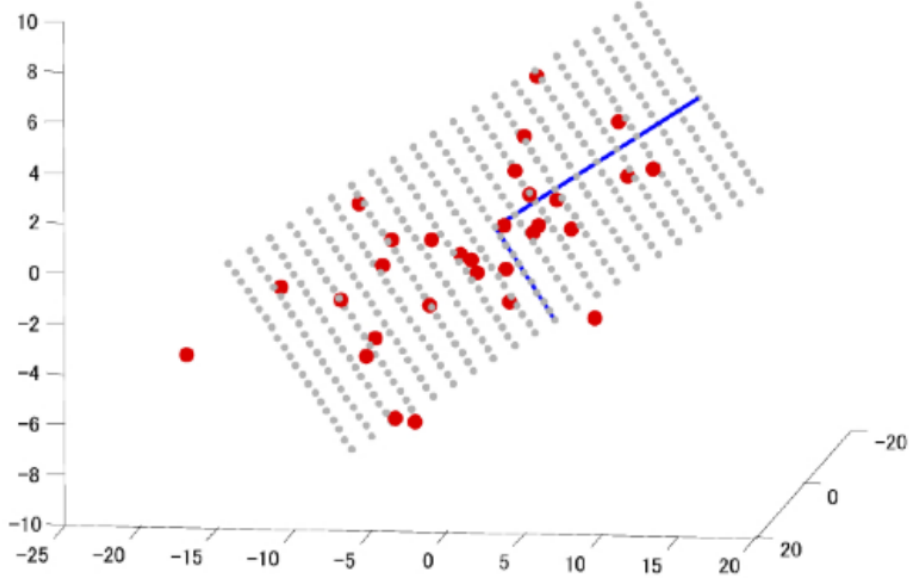


Figure 1: 3D data on a 2D subspace. Red: data points, Blue: principal components. [4]

Although the data is 3D, the data approximately lies on a 2D subspace spanned by the blue axes, so instead of storing all three coordinates of each individual data point, one can simply store the sample mean (to undo the centering), the two principal components and the projections onto the 2D subspace, known as the scores. In general, for a  $p$ -dimensional dataset  $x_1, \dots, x_n \in \mathbb{R}^p$  assumed to have zero mean (since we can always shift the data to  $x_1 - \bar{x}, \dots, x_n - \bar{x}$  which has zero mean), we construct the  $p \times n$  data matrix  $X = [x_1 | \dots | x_n]$ . The projection of the data  $x_1, \dots, x_n$  onto a unit vector  $a = (a_1, \dots, a_p)^\top$  is  $(x_1 | \dots | x_n)^\top a = X^\top a =: Z$ . Note that  $\bar{x} = 0$  means  $\bar{z} = 0$ .

$$\text{Var}(Z) = \frac{1}{n-1} \sum_{i=1}^n z_i^2 = \frac{1}{n-1} Z^\top Z = \frac{1}{n-1} a^\top X X^\top a = a^\top C a$$

where  $C = \frac{1}{n-1} X X^\top$  is the  $p \times p$  covariance matrix of the data. Recall that our aim is to find the axis  $a$  that maximises the variance of the projected data, and the requirement that  $a$  has unit length can be expressed as  $a^\top a = 1$ . Thus, the task of finding the largest (or first) principal component  $a^{(1)}$  can be formulated as an optimisation problem:

$$\begin{aligned} \max_a \quad & a^\top C a \\ \text{subject to} \quad & a^\top a = 1 \end{aligned}$$

For  $k > 1$ , the  $k^{\text{th}}$  largest (or simply  $k^{\text{th}}$ ) principal component  $a^{(k)}$  can be found iteratively, by solving

$$\begin{aligned} \max_{a^{(k)}} \quad & a^{(k)\top} C a^{(k)} \\ \text{subject to} \quad & a^{(k)\top} a^{(k)} = 1, \\ & a^{(k)\top} a^{(j)} = 0 \text{ for all } j = 1, \dots, k-1. \end{aligned}$$

where the second constraint ensures orthogonality:  $a^{(k)} \perp a^{(1)}, \dots, a^{(k-1)}$ . Using Lagrange multipliers, one can show that the solutions to the above optimisation problems coincide with the solutions to the following eigenproblem:

$$C a = \lambda a,$$

To reiterate, the eigenvectors corresponding to the largest  $k$  eigenvalues define the  $k$ -dimensional subspace that captures the most variance in the data. More precisely,

$$\text{Var}(X a^{(k)}) = a^{(k)\top} C a^{(k)} = a^{(k)\top} \lambda a^{(k)} = \lambda (a^{(k)\top} a^{(k)}) = \lambda$$

so the percentage variance explained by the  $k$ th principal component equals

$$\frac{\lambda_k}{\sum_{i=1}^p \lambda_i}.$$

Each data point can then be approximated by its projection:

$$x_i \approx P(x_i) = \sum_{k=1}^K (x_i^\top a^{(k)}) a^{(k)}.$$

This reduces the dimensionality from  $p$  to  $K \ll p$  while retaining the most important features of the data. When selecting the number of principal components, it is therefore useful to plot a graph of the eigenvalues in descending order, known as a scree plot, and look for an “elbow”. An example is shown on the following page.

### 1.1.1 Time Series as Vectors

Given a sequence of time indices  $t_1 < t_2 < \dots < t_T$ , a time series  $\{x_{t_1}, x_{t_2}, \dots, x_{t_T}\} \subset \mathbb{R}^d$  can be reshaped into a vector

$$\{x_{t_1}^{(1)}, \dots, x_{t_1}^{(d)}, x_{t_2}^{(1)}, \dots, x_{t_2}^{(d)}, \dots, x_{t_T}^{(1)}, \dots, x_{t_T}^{(d)}\} \in \mathbb{R}^{dT}$$

and standard PCA can be applied to the vector representations.

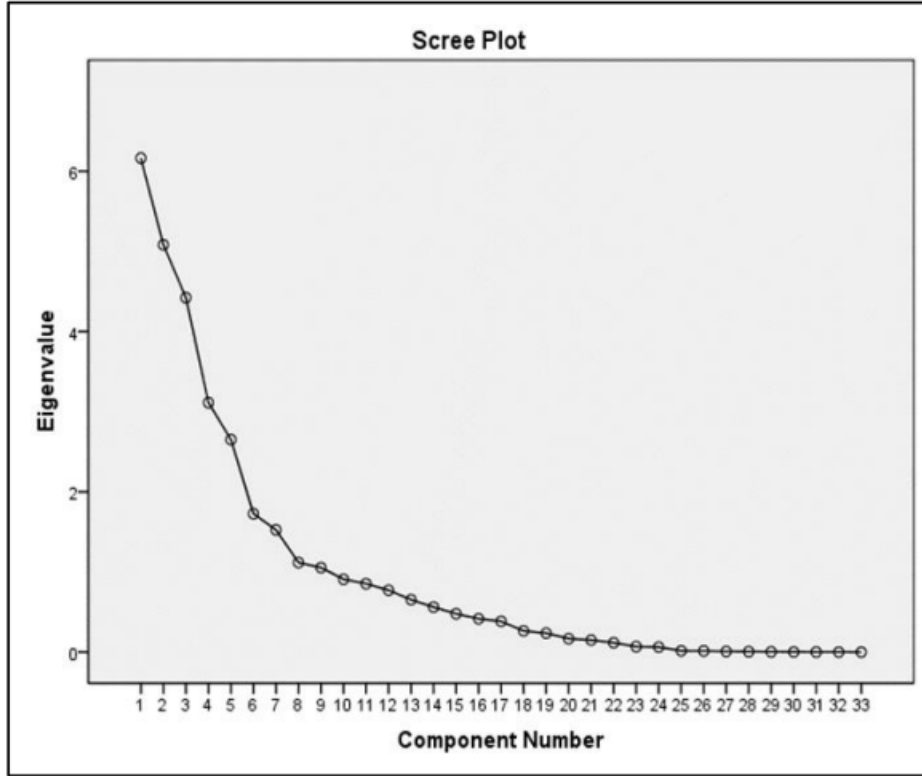


Figure 2: Scree plot of eigenvalues for an example covariance matrix. An elbow occurs around the 9th eigenvalue (1.054). The first 9 eigenvalues explain 81.39% of the total variance [5].

## 1.2 Kernel Principal Component Analysis

Standard PCA attempts to find a low-dimensional, linear subspace that the data lies on. A natural question one might ask is how to handle data that lies on a low-dimensional, nonlinear subspace? The picture below illustrates this [6]:

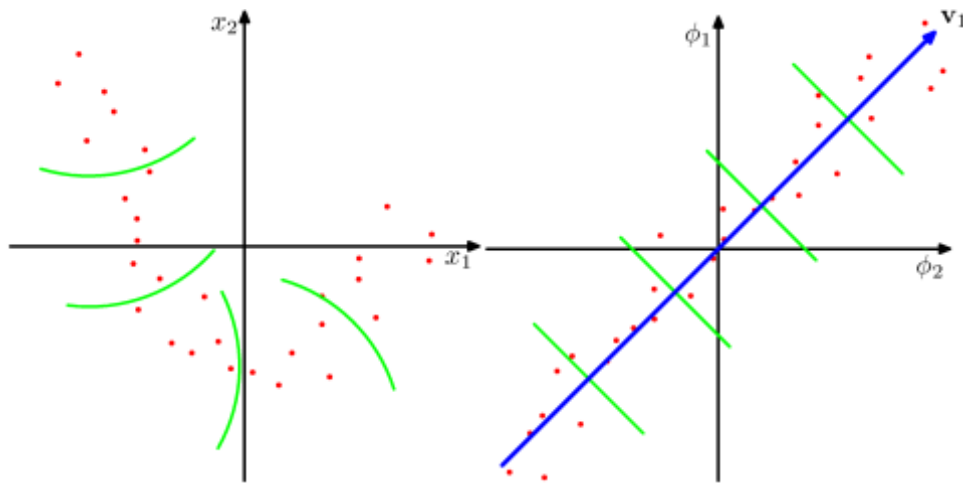


Figure 3: Data lying on a nonlinear subspace.

The data points on the left are primarily spread out along a curved trajectory, so there is no



straight line to project them onto. Instead, we transform the original data into another space which we will call the feature space, such that the transformed data lies along the straight line in the right plot. More generally, we seek a transformation  $\phi : \mathbb{R}^p \rightarrow \mathbb{R}^q$  that sends  $p$ -dimensional data onto a  $q$ -dimensional hyperplane, where  $q$  may be infinite.

The next step is to identify the principal components in the feature space and apply PCA. There is a caveat, however: applying standard PCA to the features can be computationally expensive if we work with high dimensional feature spaces, or impossible if the feature dimension is infinite. In practice, PCA is performed on another covariance matrix, as explained in the following procedure.

1. Let  $\{x_1, \dots, x_n\}$  be our data points in  $\mathbb{R}^p$ , and  $\phi(x_i)$  their images in feature space  $\mathbb{R}^q$ . We assume the transformed data is centered:  $\sum_{i=1}^n \phi(x_i) = 0$ .
2. The covariance matrix in feature space (which will not be calculated explicitly) is:

$$C = \frac{1}{n} \sum_{i=1}^n \phi(x_i) \phi(x_i)^\top.$$

3. The principal components are found by solving the eigenvalue problem:

$$Cv = \lambda v$$

where  $v = \sum_{i=1}^n \alpha_i \phi(x_i)$ .

4. Substituting  $C$  and  $v$  into the above equation leads to:

$$\frac{1}{n} \sum_{i=1}^n \phi(x_i) \phi(x_i)^\top \left( \sum_{j=1}^n \alpha_j \phi(x_j) \right) = \lambda \sum_{i=1}^n \alpha_i \phi(x_i)$$

5. This can be rewritten entirely in terms of inner products:

$$\frac{1}{n} \sum_{i=1}^n \phi(x_i) \left( \sum_{j=1}^n \alpha_j \langle \phi(x_i), \phi(x_j) \rangle \right) = \lambda \sum_{i=1}^n \alpha_i \phi(x_i)$$

6. Define the Gram matrix  $K$  where  $K_{ij} = k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$  is the kernel function of  $x_i$  and  $x_j$ . The eigenvalue problem then reduces to:

$$K\alpha = n\lambda\alpha$$

where  $\alpha = (\alpha_1, \dots, \alpha_n)^\top$ .

7. A kernel function  $k : \mathbb{R}^p \times \mathbb{R}^p \rightarrow \mathbb{R}$  computes these inner products implicitly:

$$k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$$

Common choices include the polynomial kernel  $k(x, y) = (x^\top y + c)^d$  and the Gaussian RBF kernel  $k(x, y) = \exp(-\gamma \|x - y\|^2)$ .

8. Finally, to project a new point  $x$  onto the  $k$ -th principal component:

$$\langle v_k, \phi(x) \rangle = \sum_{i=1}^n \alpha_i^k k(x_i, x)$$

where  $v_k$  is the  $k$ -th eigenvector and  $\alpha_i^k$  are the corresponding coefficients.

## **Literature Review**

### **2.1 Dictionaries**

The essence of dictionary-based compression algorithms is that time series data contains recurring segments, which can be represented by sequences of codes. [1]

### **2.2 Sequential Algorithms**

### **2.3 Functional Approximation**

Functional approximation methods are appropriate when the signal can be well-approximated by analytic forms or basis expansions, and when interpretability of the compressed representation (for instance, trend vs. detail) is beneficial.

### **2.4 Autoencoders**

## Proposed Method

### 3.1 The Signature Transform

The signature transform (or simply, the signature) arises from rough path theory and has, in recent years, received rapidly growing attention in machine learning communities thanks to its rich mathematical properties that make it a desirable feature set. It has contributed to several prize-winning applications such as Chinese handwriting recognition [7], sepsis prediction [8] and simulation of financial markets. The signature is particularly effective at encoding multivariate time series.

Consider a time series  $\{x(t) = (x_1(t), \dots, x_d(t)), t \in [a, b]\}$ , which can be viewed as a path  $x : [a, b] \rightarrow V$  where  $V = \mathbb{R}^d$ . Suppose that the path is sufficiently regular, by which we mean  $x$  has bounded  $p$ -variation 7.1 for  $1 \leq p < 2$ . We will write  $x \in C_p([a, b], V)$  to express these conditions more compactly. The signature of  $x$  is defined as the following collection of iterated integrals:

$$S(x) = (1, S^1(x), \dots, S^k(x), \dots) \in R \oplus V \oplus \dots \oplus (V)^{\otimes k} \oplus \dots =: T((V))$$

where  $S^k(x) = [S^{i_1, \dots, i_k}(x)]_{i_1, \dots, i_k=1}^d$  and for any multi-index  $\{i_1, \dots, i_k\} \in \{1, \dots, d\}^k$ ,

$$S^{i_1, \dots, i_k}(x) = \int_{a < t_1 < \dots < t_k < b} dx_{t_1}^{i_1} \dots dx_{t_k}^{i_k} \quad \text{for } 1 \leq k \leq N$$

which can be seen as a measure of interaction between variables  $i_1, \dots, i_k$  [9].

The set of  $k^{\text{th}}$ -order coefficients can be expressed more compactly as

$$S^k(x) = \int_{a < t_1 < \dots < t_k < b} dx_{t_1} \otimes \dots \otimes dx_{t_k} \in V^{\otimes k} \quad \text{for } k \geq 1$$

where  $\otimes$  denotes the tensor product. For example,  $S^1(x)$  is a vector with  $d$  entries,  $S^2(x)$  is a square matrix with  $d$  rows and  $d$  columns, and  $S^3(x)$  is a cube where each side has length  $d$ .

Unlike the Fourier transform, which captures frequency information, or the wavelet transform, which localises information in both time and frequency, the signature transform encodes the order of events happening across different channels as well as the geometric structure of the path, with higher-order coefficients encoding finer details.

#### 3.1.1 Uniqueness

A key property of signatures that justifies their application to time series compression is that signatures characterise paths up to tree-like equivalence 7.1. In fact, if a path has a strictly monotone increasing coordinate, such as time, and the starting point of the path is known, then the signature completely determines the path [10].

**Theorem: Uniqueness.** Let  $x, y \in C_p([a, b], V)$  with  $1 \leq p < 2$ . Write

$$x(t) = (x_1(t), \dots, x_d(t)), \quad y(t) = (y_1(t), \dots, y_d(t)),$$

and suppose

$$x_1(t) = y_1(t) = \rho(t),$$

where  $\rho: [a, b] \rightarrow \mathbb{R}$  is strictly monotone. If  $x(a) = y(a)$ , then

$$S(x) = S(y) \iff x(t) = y(t) \text{ for all } t \in [a, b].$$

### 3.1.2 Practical Considerations

The signature is an infinite sequence of coefficients which is clearly impossible to calculate and store on a computer. One might then ask if there is a systematic approach to select a finite subset of coefficients which encode the most important characteristics of the path. The answer lies in the following result:

**Theorem: Factorial Decay.**

Let  $x \in C_p([a, b], V)$  with  $1 \leq p < 2$ . Then the norm of the level- $k$  component  $S^k(x) \in V^{\otimes k}$  decays at least factorially fast in  $k$ . That is, there exists a constant  $C(x) > 0$  such that

$$\|S^k(x)\| \leq \frac{C(x)^k}{k!} \text{ for all } k \geq 0.$$

This ensures that the contribution from higher-order terms becomes increasingly negligible, hence the truncation of the signature at a finite level  $N$ . With this in mind, we now define the depth- $N$  signature:

$$S^{\leq N}(x) = (1, S^1(x), \dots, S^N(x))$$

### 3.2 Signature Kernels

Although it is possible to compress time series into PCA projections in truncated signature space, this is computationally inefficient, especially for rougher paths (having high 1-variation [11]) where higher-order signature coefficients are non-negligible, because the number of signature coefficients grows exponentially with truncation depth, but it also goes against the spirit of kernel methods, which is to avoid evaluating the feature map directly. The other issue is that truncating signatures will inevitably result in loss of information. It turns out that one can lift paths into the space of untruncated signatures, and compute inner products on that space efficiently and elegantly by solving a Goursat PDE [12].

Before introducing signature kernels, we will first define a family of inner products on the space of signatures,  $T((V))$ : For any  $k \in \mathbb{N}$ , an inner product  $\langle \cdot, \cdot \rangle_V$  on a vector space  $V$  induces an inner product on the tensor product space  $V^{\otimes k}$ , denoted by  $\langle \cdot, \cdot \rangle_{V^{\otimes k}}$  [11]. More precisely,

$$\langle v, w \rangle_{V^{\otimes k}} := \prod_{i=1}^k \langle v_i, w_i \rangle_V,$$

for any  $v = (v_1, \dots, v_k)$ ,  $w = (w_1, \dots, w_k)$  in  $V^{\otimes k}$ . This in turn defines a family of inner products: given a weight function  $\varphi : \mathbb{N} \cup \{0\} \rightarrow \mathbb{R}_+$ , we define the  $\varphi$ -inner product on the tensor algebra  $T(V)$  by

$$\langle v, w \rangle_\varphi := \sum_{k=0}^{\infty} \varphi(k) \langle v_k, w_k \rangle_{V^{\otimes k}},$$

for any  $v = (v_0, v_1, \dots)$ ,  $w = (w_0, w_1, \dots)$  in  $T(V)$ . The Hilbert space obtained by completing  $T(V)$  with respect to this inner product is denoted by  $T_\varphi((V))$ .

Due to time constraints, we will simply investigate  $\phi(k) \equiv 1$ , but in practice it is also common to consider weight functions that coincide with moments of random variables, i.e.  $\phi(k) = E[\pi^k]$  for a random variable  $\pi$ , as explained in [11]. Thus, for two paths  $x : [a, b] \rightarrow R^d$ ,  $y : [c, d] \rightarrow R^d$ , we work with the unweighted signature kernel:

$$k^{x,y}(s, t) = \langle S(x)_{[a,s]}, S(y)_{[c,t]} \rangle_{T((V))}$$

and take  $\langle \cdot, \cdot \rangle$  to be the dot product. We will now present the result that allows one to compute the kernel in practice.

#### Theorem: Signature Kernel PDE

Let  $I = [u, u_0]$  and  $J = [v, v_0]$  be two compact intervals, and let  $x \in C^1(I, V)$  and  $y \in C^1(J, V)$ . The signature kernel  $k_{x,y}$  is a solution to the following PDE [12]:

$$\frac{\partial^2 k_{x,y}}{\partial s \partial t} = \langle \dot{x}_s, \dot{y}_t \rangle_V k_{x,y}, \quad (1)$$

with boundary conditions

$$k_{x,y}(u, \cdot) = k_{x,y}(\cdot, v) = 1. \quad (2)$$

Here, the derivatives of  $x$  and  $y$  at time  $s$  and  $t$  are given by

$$\dot{x}_s = \left. \frac{dx_p}{dp} \right|_{p=s}, \quad \dot{y}_t = \left. \frac{dy_q}{dq} \right|_{q=t}. \quad (3)$$

Using the untruncated signature kernel offers perfect accuracy in theory, as the infinite depth signature encodes the path exactly. In practice, however, the accuracy depends on the chosen dyadic order parameter, with higher values corresponding to finer mesh resolution of the finite difference scheme and thus more accurate computations.

### 3.3 Reproducing Kernel Hilbert Spaces

#### Definition: Positive Semi-Definite Kernels

A kernel function  $k : X \times X \rightarrow \mathbb{R}$  is positive semidefinite if for any  $n \in \mathbb{N}$  and any points  $x_1, \dots, x_n \in X$ , the Gram matrix

$$K = (k(x_i, x_j))_{i,j=1}^n$$

is positive semidefinite. That is, for any real coefficients  $c_1, \dots, c_n$ , we have

$$\sum_{i=1}^n \sum_{j=1}^n c_i c_j k(x_i, x_j) \geq 0.$$

The signature kernel (and as a result, the centred signature kernel) is positive semi-definite (see [11] for a proof).

#### Theorem: Moore-Aronszajn Theorem [11]

If  $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$  is a positive semidefinite kernel, then there is a unique RKHS  $\mathcal{H}$  such that  $k$  has the reproducing property:

$$\forall x \in \mathcal{X}, k(x, \cdot) \in \mathcal{H} \quad \text{and} \quad \forall f \in \mathcal{H}, \langle f, k(x, \cdot) \rangle_{\mathcal{H}} = f(x)$$

In particular, taking  $f(\cdot) = k(y, \cdot)$ ,

$$\langle k(x, \cdot), k(y, \cdot) \rangle_{\mathcal{H}} = k(x, y)$$

#### 3.3.1 Universality and Characteristicness

### 3.3.2 Compression with Signature Kernel PCA

We are now ready to present our proposed method. The main idea is to associate a functional  $k(x_i, \cdot) \in \mathcal{H}$  to each path  $x_i$  and project the functional onto the principal axes (in this case, principal functions). The step-by-step procedure is explained below:

1. Sample or simulate  $n$  paths  $x_1, \dots, x_n$ .
2. Form the covariance (or Gram) matrix  $G$  where

$$G_{ij} = k(x_i, x_j) = \langle k(x_i, \cdot), k(x_j, \cdot) \rangle_{\mathcal{H}}$$

Then compute the centred Gram matrix  $\tilde{G}$ :

$$\tilde{G} = (I_n - \frac{1}{n}1_n)G(I_n - \frac{1}{n}1_n)$$

where  $I_n$  is the  $n \times n$  identity matrix and  $1_n$  is the  $n \times n$  matrix of 1s. We have

$$G_{ij} = \tilde{k}(x_i, x_j) = \langle k(x_i, \cdot) - \frac{1}{n} \sum_{l=1}^n k(x_l, \cdot), k(x_j, \cdot) - \frac{1}{n} \sum_{m=1}^n k(x_m, \cdot) \rangle_{\mathcal{H}}$$

3. Compute the eigenvalues and eigenvectors of  $G$ , i.e. all pairs of solutions  $(\lambda_k, \alpha_k)$  to  $G\alpha_k = \lambda_k\alpha_k$  and store them in descending order of eigenvalues  $\lambda_1 \geq \dots \geq \lambda_n$ . Note that the eigenvalues are all non-negative since  $G$  is a covariance matrix which is positive semi-definite.
4. Rescale the eigenvectors so that  $\|\alpha_k\| = 1/\sqrt{\lambda_k}$  for all  $k = 1, \dots, n$ . This ensures that the corresponding principal axes in feature space have unit norm, i.e.  $\left\| \sum_{i=1}^n \alpha_i^k k(x_i, \cdot) \right\|_{\mathcal{H}} = 1$ .
5. For  $k = 1, \dots, K$ , where  $K$  is the selected number of principal components, the  $k^{\text{th}}$  principal component score of path  $x_j$  is given by

$$p_k(x_j) = \left\langle \sum_{i=1}^n \alpha_i^k k(x_i, \cdot), k(x_j, \cdot) \right\rangle_{\mathcal{H}} = \sum_{i=1}^n \alpha_i^k k(x_i, x_j) \quad (\text{bilinearity} + \text{reproducing property})$$

where  $a^k = (a_1^k, \dots, a_n^k)$ . However, because the decompression part of this scheme currently involves truncated signatures, the scores are calculated as follows:

$$p_k^N(x_j) = \sum_{i=1}^n \alpha_i^k k^N(x_i, x_j) = \sum_{i=1}^n \alpha_i^k < S^{\leq N}(x_i), S^{\leq N}(x_j) >_{T^N(V)}$$

While RKHS pre-image problems have been studied, such as in [13], to the best of the author's knowledge, existing methods require knowledge of the original data matrix  $X = \{x_1, \dots, x_N\}$  and thus cannot be used for compression.

#### Equivalence between Signature Kernel RKHS and Signature Tensor Algebra

Crucially, although the above procedure considers the feature map  $\phi_1 : x \rightarrow k(x, \cdot)$ , we will instead consider the feature map  $\phi_2 : x \rightarrow S(x)$ , for the purposes of computation. The reason is that the Gram matrix is in fact the same:

$$G_{ij} = k(x_i, x_j) = \langle S(x_i) - \bar{S}, S(x_j) - \bar{S} \rangle_{T((V))}$$



And

$$G_{ij} = k(x_i, x_j) = \langle k(x_i, \cdot), k(x_j, \cdot) \rangle_{\mathcal{H}}$$

Consequently, the eigenvectors  $\{a_k\}_{k=1}^n$  are the same.

### 3.4 Reconstruction

Having stored the projections onto the principal components, we would like to recover the original path  $x$  from its projection. The following procedure yields a reconstruction (note that the paths are time-augmented first):

1. Compute projections in the standard basis of the feature space:

$$P(x) = \sum_{i=1}^K p_k(x) V_k^N$$

with

$$V_k^N = \sum_{i=1}^n \alpha_i^k S(x_i)$$

2. Reverse centering:

$$\hat{S}^{\leq N}(x) = P(x) + \frac{1}{n} \sum_{i=1}^n S(x_i)$$

3. Solve

$$\min_y \left\| \hat{S}^{\leq N}(x) - S^{\leq N}(y) \right\|$$

with gradient descent [14]:

$$y_{r+1} = y_r - \gamma \nabla_y \left\| \hat{S}^{\leq N}(x) - S^{\leq N}(y) \right\|_{y=y_r}$$

where  $\gamma > 0$  is a fixed step size. The optimisation was initially carried out using stochastic gradient descent (SGD), as the noisy loss functions (at each iteration, a subset of signature coefficients were randomly chosen) should help the trajectory escape local minima and saddle points. However, standard gradient descent produced more accurate reconstructions. Perhaps this is due to instability arising from SGD: because the signature coefficients can vary greatly in magnitude, even if the proportion of coefficients selected each iteration is very high, they might not accurately represent the true loss function.

Remark: The number of iterations  $R$  was chosen so that beyond iteration  $R$ , the loss no longer decreases significantly, in a similar fashion to choosing the number of principal components to keep in our kernel PCA pipeline. Otherwise, if we keep running gradient descent despite only marginal reductions in loss, the recovered path actually starts to deviate significantly from the original one. This may be because gradient descent increasingly focuses on minimising the mismatch in higher-order signature coefficients, which capture fine geometric details, while neglecting discrepancies in lower-order terms — especially the first-order signature, which corresponds to the net displacement in each coordinate and is essential for capturing the overall shape of the path.

### 3.5 Redundancies

Signature coefficients are not algebraically independent: products of lower-order signature coefficients can always be expressed as sums of higher-order coefficients. For example,  $S^{i,j}(x) +$

$S^{j,i}(x) = S^i(x)S^j(x)$ . Clearly, any three terms uniquely determines the fourth term, so knowing all four terms gives no additional information over knowing just three of them. However, if we work with the untruncated signature kernel, we solve a PDE rather than storing signature coefficients directly, so the argument only applies to truncated signature kernels.

The signature is not the only way to transform paths. A closely related transform is the log-signature, the logarithm of the signature in  $T((V))$ , and for the same truncation level, the log-signature has far fewer entries than the truncated-signature and has no algebraic dependencies. Crucially, the log-signature also characterises paths up to tree-like equivalence, and in the case of time-augmented paths, a path is completely determined by its log-signature. Thus, one might consider inverting the truncated log-signature to recover paths. However, when the signature is replaced by the log-signature in the loss function, it is difficult to compute the gradient vector due to differentiability issues. Instead, one may consider gradient-free optimisation algorithms such as Nelder-Mead [15]. The following subsection explores an alternative compact representation of paths based on standard signature coefficients.

### 3.6 Sparse Signature

With the algebraic dependencies in mind, let us consider multi-indices of the form  $(1, \dots, 1, k)$ ,  $k \in \{2, \dots, d\}$  where the first channel is time and there are  $d - 1$  spatial dimensions. This approach is motivated by the proof of uniqueness in [11], which shows that, for two time-augmented paths  $x, y \in C_p([a, b], V)$  with the same starting point  $x_a = y_a$ , it is already sufficient that  $S^{1, \dots, 1, k}(x) = S^{1, \dots, 1, k}(y)$  for all indices of this form to conclude that  $x = y$ . It can also be shown that there are no algebraic dependencies between these indices. If  $m$  is the number of 1s, define

$$\hat{S}_{\text{SPARSE}}^{\leq N}(x) = \left\{ \hat{S}^{(1, \dots, 1, k)}(x) \mid m \in \{0, \dots, N - 1\}, k \in \{2, \dots, d\} \right\}$$

then the loss function is

$$L_x(y) = \|\hat{S}_{\text{SPARSE}}^{\leq N}(y) - \hat{S}_{\text{SPARSE}}^{\leq N}(x)\|$$

and as before, the optimisation is done by running gradient descent. Although the loss function is defined for a single path, we can still use a batch size greater than 1 by interpreting the batch as a randomly selected subset of signature coefficients from  $\hat{S}_{\text{SPARSE}}^{\leq N}(x)$ .

## Experiments

Brownian motion is commonly used to model phenomena in biology, chemistry, physics and finance, making the simulation of such paths essential in many applications. In our experiments, we simulate 1000 3-dimensional Arithmetic Brownian motion paths with zero drift and unit volatility. The plots are shown on the following pages.

## 4.1 Eigenvalue Scree Plots

### 4.1.1 Path Length

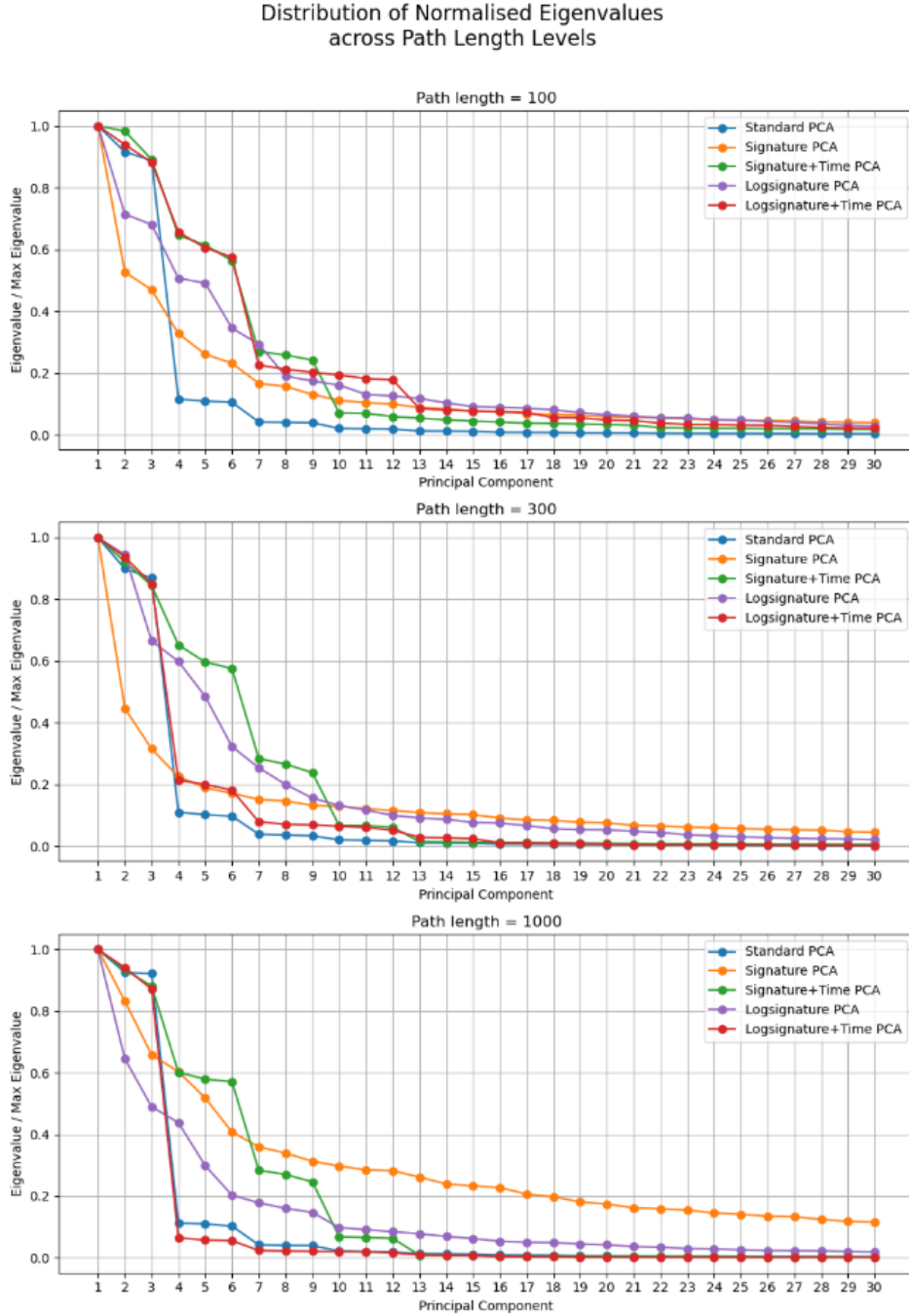


Figure 4: Scree plot showing the distribution of Gram matrix eigenvalues for different path lengths.

The sharp drop in normalised eigenvalue size from  $d = 3$  to  $d = 4$  is likely related to the simulated Brownian motion paths being 3-dimensional, because in general, as the path dimension  $d$  varies, the sharp drop occurs from between eigenvalues indexed  $d$  and  $d + 1$ . For our compression and reconstruction scheme, we focus on time-augmented paths. It can be seen from the eigenvalue

distributions that the "inherent dimensionality" of time-augmented paths is higher than those without time augmentation, because for time-augmented paths, the eigenvalues (and thus explained variance) do not decay to zero as quickly. One possible explanation is that the time channel causes the signature to become sensitive to speed, and thus captures more details of each path.

### 4.1.2 Path Dimension

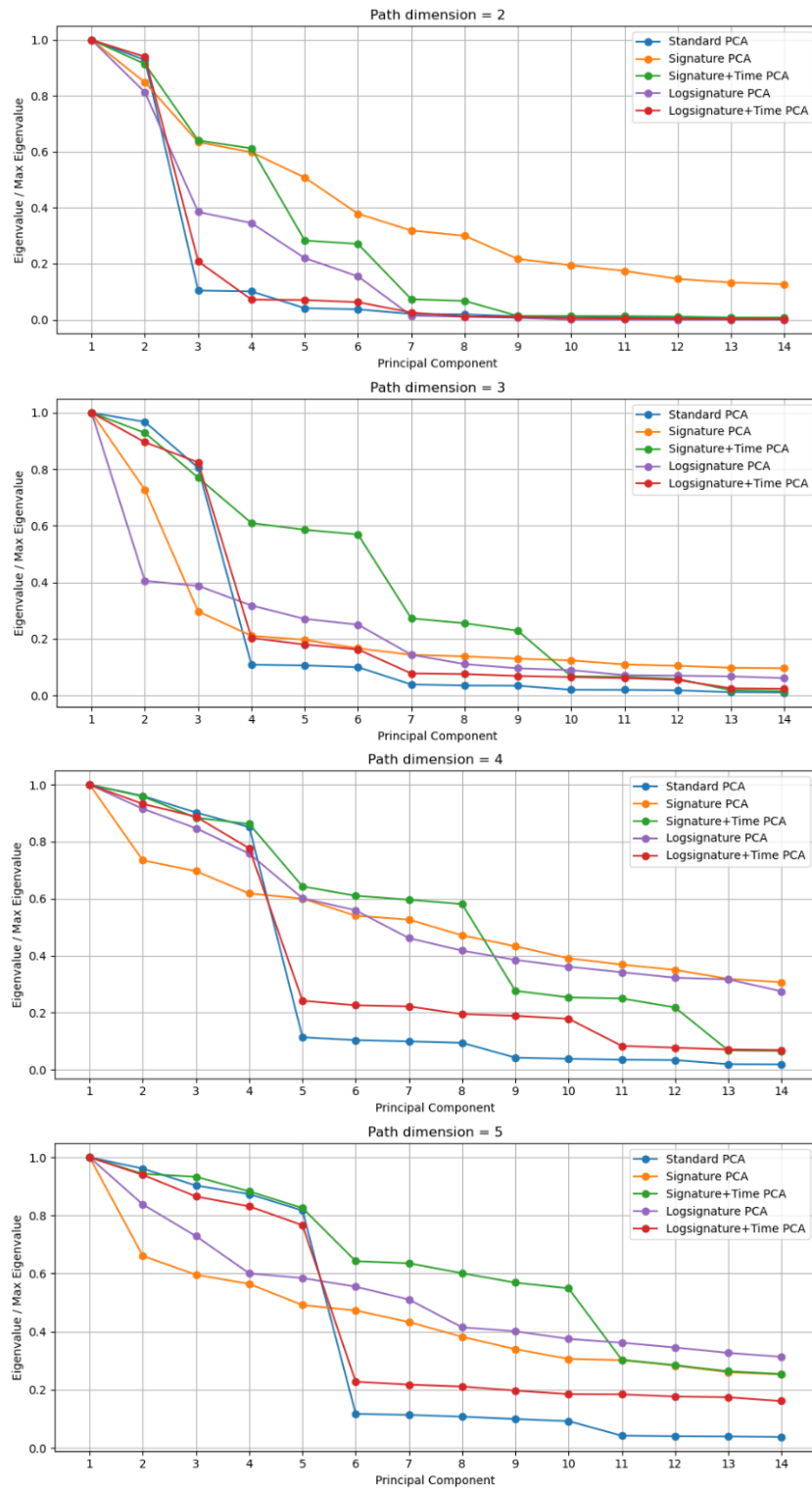


Figure 5: Scree plot showing the distribution of Gram matrix eigenvalues for different path dimensions.

### 4.1.3 Signature Depth

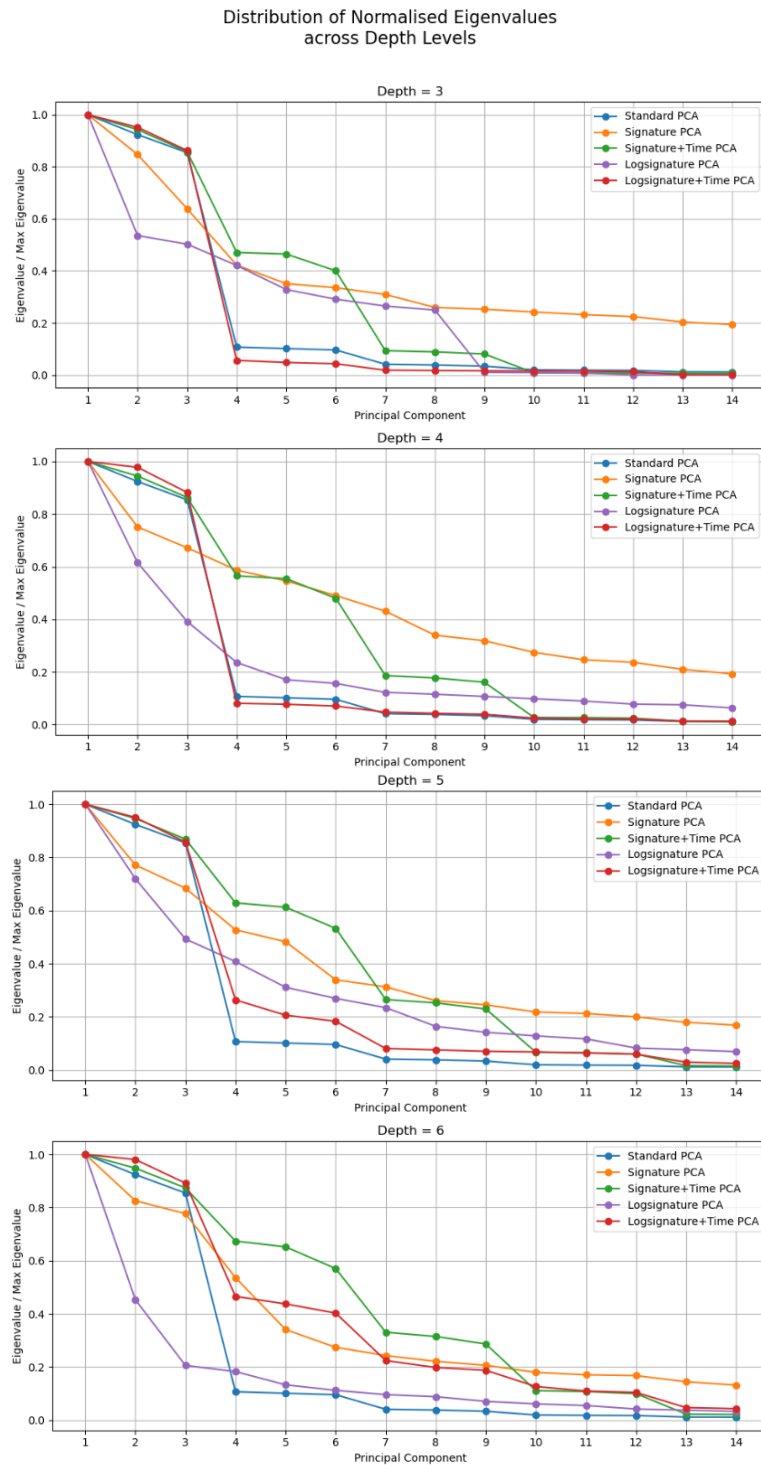


Figure 6: Scree plot showing the distribution of Gram matrix eigenvalues for different signature truncation depths.

Depth 3 and 5 appear to have similar distributions. This may be due to a property of standard Brownian motion: the expected signature coefficients at odd depths are zero.



## 4.2 Compression Ratio and Recovery Accuracy

### 4.2.1 Signature Inversion with Gradient Descent

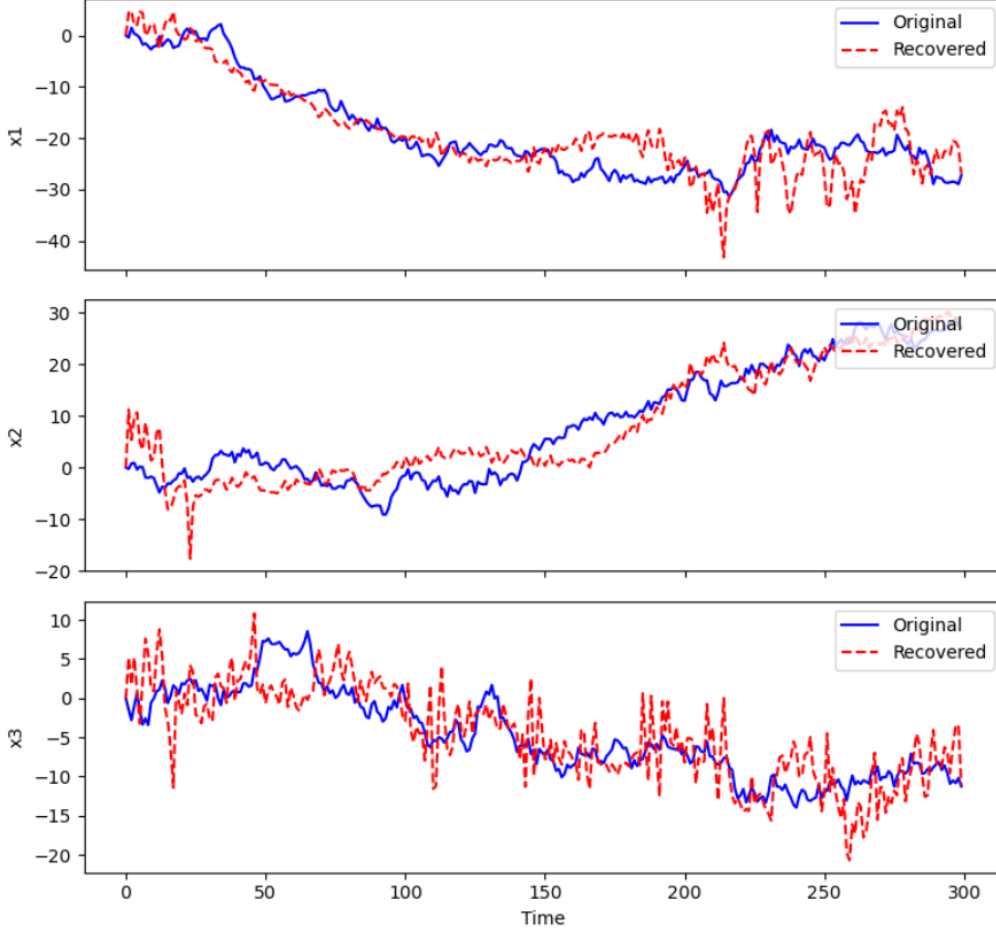


Figure 7: Path reconstruction using standard gradient descent with a learning rate of 0.01 and 4000 iterations on signature coefficients up to depth 5.

Compression Ratio: 12.45

Note that each path  $x_1, \dots, x_n$  consists of  $dT$  scalars so the original data size is  $ndT$ . After applying signature kernel principal component compression, we store  $K$  principal components in feature space, where each principal component has a size of  $D_{N,d} = (d+1) + \dots + (d+1)^N$ , so together  $KD_{N,d}$ . We also store the mean feature vector (contributing  $D_{N,d}$ ) and the  $K$  scores for each sample (contributing  $Kn$ ). We thus have

$$\text{Compression Ratio} = \frac{ndT}{(K+1)D_{N,d} + Kn}$$

It is worth noting that the path length  $T$  only appears in the numerator, so this method of compression is more efficient for longer time series.

### 4.2.2 Sparse Signature Inversion with Gradient Descent

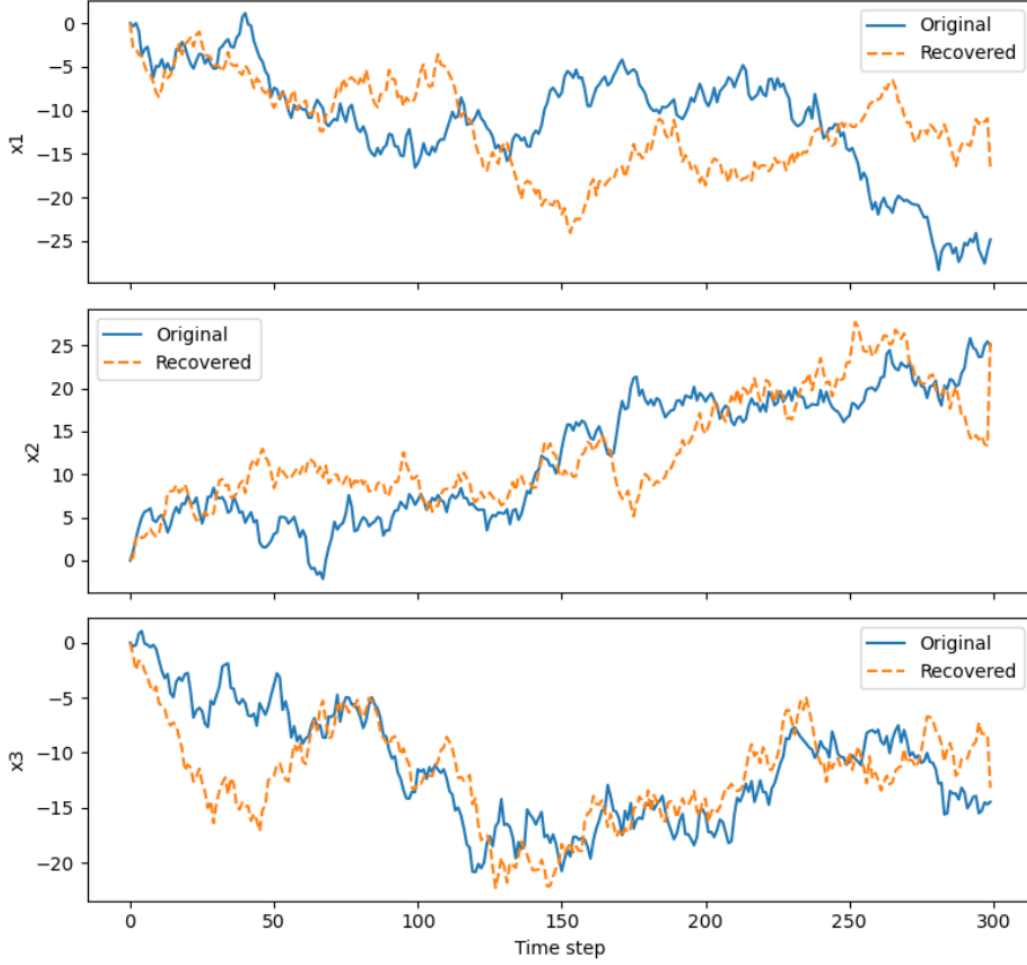


Figure 8: Path reconstruction using gradient descent on selected signature coefficients up to depth 5.

Compression Ratio: 29.54

Recall that the original data size is  $ndT$ . After applying sparse signature kernel principal component compression, we store  $K$  principal components in feature space, where each principal component has a size of  $d + d + \dots + d = dN$ , because we extract  $d$  coefficients from each level of the signature up to level  $N$ , namely those corresponding to the multi-indices:

$$(1, \dots, 1, 2), (1, \dots, 1, 3), \dots, (1, \dots, 1, d + 1).$$

As before, we store the mean feature vector (contributing  $dN$ ) and the  $K$  scores for each sample (contributing  $Kn$ ).

$$\text{Compression Ratio} = \frac{ndT}{(K + 1)dN + Kn}$$

## Conclusion

In this thesis, we have considered the effect of parameters such as path length, path dimension and signature truncation depth on compression performance, and more specifically, the trade-off between reconstruction accuracy and compression ratio.

An interesting observation is that, while the sparse signature reconstruction shows greater overall deviation from the original path than the standard signature reconstruction, it seems to better capture finer details.

With more time, we would have tested our methods on real-world time series and, following the philosophy of kernel methods, aimed to develop a reconstruction scheme that avoids direct evaluation of signature coefficients.

## Bibliography

- [1] Giacomo Chiarot and Claudio Silvestri. “Time Series Compression Survey”. In: *ACM Computing Surveys* 55.10 (2023), pp. 1–32. URL: <https://dl.acm.org/doi/pdf/10.1145/3560814>.
- [2] *Neuralink Compression Challenge*. 2024. URL: <https://content.neuralink.com/compression-challenge/README.html>.
- [3] Terry J. Lyons and Sidorova. “Sound Compression: A Rough Path Approach”. In: *Proceedings of the 4th International Symposium on Information and Communication Technologies (WISICT '05)*. 2005. URL: [https://www.ucl.ac.uk/~ucahnsi/Papers/lyons\\_sidorova\\_capetown.pdf](https://www.ucl.ac.uk/~ucahnsi/Papers/lyons_sidorova_capetown.pdf).
- [4] Arthur Gretton. *Introduction to RKHS, and some simple kernel algorithms*. 2019. URL: [https://www.gatsby.ucl.ac.uk/~gretton/coursefiles/lecture4\\_introToRKHS.pdf](https://www.gatsby.ucl.ac.uk/~gretton/coursefiles/lecture4_introToRKHS.pdf).
- [5] L Gour et al. “Characterization of rice (*Oryza sativa* L.) genotypes using principal component analysis including scree plot & rotated component matrix”. In: *International Journal of Chemical Studies* 5.6 (2017), pp. 686–690. URL: [https://www.researchgate.net/publication/321780566\\_Characterization\\_of\\_rice\\_Oryza\\_sativa\\_L\\_genotypes\\_using\\_principal\\_component\\_analysis\\_including\\_scree\\_plot\\_rotated\\_component\\_matrix](https://www.researchgate.net/publication/321780566_Characterization_of_rice_Oryza_sativa_L_genotypes_using_principal_component_analysis_including_scree_plot_rotated_component_matrix).
- [6] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. New York: Springer, 2006.
- [7] Benjamin Graham. *Sparse Arrays of Signatures for Online Character Recognition*. 2013. URL: <https://arxiv.org/pdf/1308.0371>.
- [8] J. H. Morrill et al. “Utilisation of the signature method to identify the early onset of sepsis from multivariate physiological time series in critical care monitoring”. In: *Critical Care Medicine* 48.10 (2020), e976–e981. URL: <https://ora.ox.ac.uk/objects/uuid:ad5f3c5b-e2c3-402e-a859-67f668a9d250/files/rcc08hf65g>.
- [9] Barbora Barancikova, Zhuoyue Huang, and Cristopher Salvi. “SigDiffusions: Score-Based Diffusion Models for Time Series via Log-Signature Embeddings”. In: *The Thirteenth International Conference on Learning Representations*. 2025. URL: <https://arxiv.org/pdf/2406.10354>.
- [10] Ben Hambly and Terry Lyons. “Uniqueness for the signature of a path of bounded variation and the reduced path group”. In: *Annals of Mathematics* (2005).
- [11] Cristopher Salvi and Thomas Cass. “Lecture notes on Rough Paths and Applications to Machine Learning”. In: *arXiv* (2024). URL: <https://arxiv.org/pdf/2404.06583>.
- [12] Cristopher Salvi et al. “The Signature Kernel is the solution of a Goursat PDE”. In: *arXiv preprint arXiv:2006.14794* (2020). URL: <https://arxiv.org/pdf/2006.14794>.

- [13] Paul Honeine and Cédric Richard. “A closed-form solution for the pre-image problem in kernel-based machines”. In: *Journal of Signal Processing Systems* 65.3 (2011), pp. 289–299. DOI: 10.1007/s11265-010-0482-9.
- [14] Patric Bonnier et al. “Deep Signature Transforms”. In: *arXiv* (2019). Published at NeurIPS 2019. <https://doi.org/10.48550/arXiv.1905.08494>.
- [15] J. A. Nelder and R. Mead. “A Simplex Method for Function Minimization”. In: *The Computer Journal* 7.4 (1965), pp. 308–313. DOI: 10.1093/comjnl/7.4.308. URL: [https://www.ime.unicamp.br/~sandra/MT853/handouts/Ref3\(NelderMead1965\).pdf](https://www.ime.unicamp.br/~sandra/MT853/handouts/Ref3(NelderMead1965).pdf).
- [16] Darrick Lee and Robert Ghrist. “Path Signatures on Lie Groups”. In: *arXiv* (2020). URL: <https://arxiv.org/pdf/2007.06633>.

## Appendix

### 7.1 Definitions

#### Definition (Reducible Path).

This definition is taken from [16]. A path is said to be reducible if it can be constructed from paths  $\alpha, \beta, \zeta$  such that,

$$\gamma = \alpha * \zeta * \zeta^{-1} * \beta.$$

The concatenation  $\alpha * \beta$  is known as a reduction of  $\gamma$ . The pair  $\zeta * \zeta^{-1}$ , a path followed by its reverse, is treated as a cancellation, and is identified with  $c_e$ , the constant path. A path is called irreducible if it admits no such reduction. Any irreducible path obtained via finitely many such reductions from  $\gamma$  is referred to as an irreducible reduction of  $\gamma$ .

#### Definition (Tree-like Path and Tree-like Equivalence).

A path is tree-like if its irreducible reduction is the constant path  $c_e$ . Two paths  $\alpha$  and  $\beta$  are tree-like equivalent, denoted by  $\alpha \sim \beta$ , if the path  $\alpha * \beta^{-1}$  is tree-like.

#### Definition (Bounded p-variation).

Let  $p \geq 1$ . A path  $x : [0, T] \rightarrow \mathbb{R}^d$  has bounded  $p$ -variation if

$$\|x\|_{p\text{-var};[0,T]} := \left( \sup_{\mathcal{P}} \sum_i \|x_{t_{i+1}} - x_{t_i}\|^p \right)^{\frac{1}{p}} < \infty,$$

where the supremum is taken over all partitions  $\mathcal{P} = \{0 = t_0 < t_1 < \dots < t_n = T\}$  of the interval  $[0, T]$ .

## 7.2 Code

GitHub Link: <https://github.com/danielzml/M4R>

### 7.2.1 Eigenvalue Scree Plots

```
1  import iisignature
2  import numpy as np
3  import torch
4  import matplotlib.pyplot as plt
5
6  # Default parameters
7  num_samples      = 1000
8  len_x_default    = 300  # number of points in each path
9  len_x            = len_x_default
10 drift_default    = 0
11 drift            = drift_default
12 volatility_default = 1
13 volatility        = volatility_default
14 time_increment_default = 1
15 time_increment    = time_increment_default
16 depth_default     = 5  # signature truncation level
17 depth            = depth_default
18 d_default         = 3  # path dimension
19 d                = d_default
20 num_pcs_default   = 14  # number of principal components to keep
21 num_pcs          = num_pcs_default
22
23 # Parameter lists (for exploring how PCA-eigenvalue spectra change)
24 path_dimension_list = [2, 3, 4, 5]
25 depth_list          = [3, 4, 5]
26 drift_list           = [0.01, 0.1, 1, 10]
27 volatility_list      = [0.001, 0.01, 0.1, 1, 10]
28 time_increment_list = [0.001, 0.01, 0.1, 1, 10]
29 path_length_list    = [100, 300, 1000]
30
31 parameter_to_vary = 'path dimension'
32
33 if parameter_to_vary == 'depth':
34     parameter_list = depth_list
35 elif parameter_to_vary == 'drift':
36     parameter_list = drift_list
37 elif parameter_to_vary == 'volatility':
38     parameter_list = volatility_list
```

```

39 elif parameter_to_vary == 'time increment':
40     parameter_list = time_increment_list
41 elif parameter_to_vary == 'path length':
42     parameter_list = path_length_list
43 elif parameter_to_vary == 'path dimension':
44     parameter_list = path_dimension_list
45 else:
46     raise ValueError("Unknown parameter_to_vary.")
47
48
49 # -----
50 # 1) Generate a batch of Brownian-motion{type paths (spatial only)}
51 # -----
52 def generate_brownian_motion(num_samples, len_x, drift, volatility,
53                             initial_condition=None, time_increment=0.1):
54     """
55     Returns a torch.Tensor of shape (num_samples, len_x, d):
56     X[:, 0, :] = 0 (if initial_condition is zero)
57     X[:, t, :] = cumulative sum of (dW + drift*t) up to index t.
58     We do NOT append time here; this is purely spatial.
59     """
60     t_x = torch.linspace(0, (len_x - 1) * time_increment, len_x,
61                          dtype=torch.float64).view(1, len_x, 1) # (1, len_x,
62                                                                ↪ 1)
63     dW = torch.randn(num_samples, len_x - 1, d, dtype=torch.float64) \
64         * torch.sqrt(torch.tensor(time_increment, dtype=torch.float64)) \
65         * volatility
66     dW = torch.cat([
67         torch.zeros(num_samples, 1, d, dtype=torch.float64),
68         dW
69     ], dim=1) # (num_samples, len_x, d)
70     X = torch.cumsum(dW, dim=1) + drift * t_x
71     if initial_condition is not None:
72         X = X + initial_condition.view(1, 1, d)
73     return X # (num_samples, len_x+1, d)
74
75 # -----
76 # 2) PCA helper (returns reconstruction too)
77 # -----
78 def perform_pca(X: np.ndarray, n_components: int):
79     """
80     Input:

```



```

81     X: (n_samples, n_features)
82 Returns:
83     eigenvalues:      (n_features,) sorted descending
84     projections:      (n_samples, k)          # principal-component
                        ↪ coefficients
85     X_reconstructed:  (n_samples, n_features) # back-projection into
                        ↪ original basis
86     """
87     mean_X = np.mean(X, axis=0, keepdims=True)          # (1, n_features)
88     X_centered = X - mean_X                             # (n_samples,
                        ↪ n_features)
89     cov = np.cov(X_centered, rowvar=False)              # (n_features,
                        ↪ n_features)
90     eigenvalues, eigenvectors = np.linalg.eigh(cov)
91     idx_desc = np.argsort(eigenvalues)[::-1]
92     eigenvalues = eigenvalues[idx_desc]
93     eigenvectors = eigenvectors[:, idx_desc]             # columns are sorted
                        ↪ eigenvectors
94     n_features = eigenvalues.shape[0]
95     k = min(n_components, n_features)
96     W = eigenvectors[:, :k]                             # (n_features, k)
97     projections = X_centered @ W                        # (n_samples, k)
98     X_reconstructed = projections @ W.T + mean_X        # (n_samples,
                        ↪ n_features)
99     return eigenvalues, projections, X_reconstructed
100
101
102 def cumulative_explained_variance(eigenvalues: np.ndarray, k):
103     total_var = np.sum(eigenvalues)
104     explained = np.cumsum(eigenvalues[:k])
105     return [round(float(ev) / total_var, 4) for ev in explained]
106
107
108 # -----
109 # 3) Augmentation routines (basepoint first, then time)
110 # -----
111 def add_basepoint_zero_spatial_only(paths_spatial: np.ndarray) -> np.ndarray:
112     batch, len_x_, d_ = paths_spatial.shape
113     zero_spatial = np.zeros((batch, 1, d_), dtype=paths_spatial.dtype)
114     return np.concatenate([zero_spatial, paths_spatial], axis=1) # (batch,
                        ↪ len_x+1, d)
115
116

```

```

117 def add_time_after_basepoint(paths_spatial_with_base: np.ndarray,
118                               time_increment: float) -> np.ndarray:
119     batch, len_x_plus1, d_ = paths_spatial_with_base.shape
120     times = np.arange(len_x_plus1, dtype=np.float64).reshape(1, len_x_plus1,
121     ↪ 1) * time_increment
122     times = np.tile(times, (batch, 1, 1)) # (batch, len_x+1, 1)
123     return np.concatenate([times, paths_spatial_with_base], axis=2) # (batch,
124     ↪ len_x+1, d+1)
125
126 # -----
127 # 4) Batch signature / logsignature routines
128 # -----
129 def batch_signature(paths_spatial: np.ndarray, depth: int) -> np.ndarray:
130     paths_with_base = add_basepoint_zero_spatial_only(paths_spatial) #
131     ↪ (batch, len_x+1, d)
132     sigs = []
133     for path in paths_with_base:
134         sig = iisignature.sig(np.ascontiguousarray(path, dtype=np.float64),
135         ↪ depth)
136         sigs.append(sig)
137     return np.array(sigs) # (batch, sig_dim)
138
139 def batch_logsignature(paths_spatial: np.ndarray, depth: int) -> np.ndarray:
140     paths_with_base = add_basepoint_zero_spatial_only(paths_spatial) #
141     ↪ (batch, len_x+1, d)
142     state = iisignature.prepare(paths_spatial.shape[2], depth)
143     logsigs = []
144     for path in paths_with_base:
145         logsig = iisignature.logsig(np.ascontiguousarray(path,
146         ↪ dtype=np.float64), state)
147         logsigs.append(logsig)
148     return np.array(logsigs) # (batch, logsig_dim)
149
150 def batch_signature_time(paths_spatial: np.ndarray,
151                           depth: int,
152                           time_increment: float) -> np.ndarray:
153     paths_with_base = add_basepoint_zero_spatial_only(paths_spatial) #
154     ↪ (batch, len_x+1, d)
155     paths_time = add_time_after_basepoint(paths_with_base, time_increment) #
156     ↪ (batch, len_x+1, d+1)

```

```

152     sigs_time = []
153     for path in paths_time:
154         sig_t = iisignature.sig(np.ascontiguousarray(path, dtype=np.float64),
155                                ↪ depth)
156         sigs_time.append(sig_t)
157     return np.array(sigs_time) # (batch, sig_dim)
158
159 def batch_logsignature_time(paths_spatial: np.ndarray,
160                             depth: int,
161                             time_increment: float) -> np.ndarray:
162     paths_with_base = add_basepoint_zero_spatial_only(paths_spatial) #
163     ↪ (batch, len_x+1, d)
164     paths_time = add_time_after_basepoint(paths_with_base, time_increment) #
165     ↪ (batch, len_x+1, d+1)
166     state = iisignature.prepare(paths_time.shape[2], depth) # channels = d+1
167     logsigs_time = []
168     for path in paths_time:
169         logsig_t = iisignature.logsig(np.ascontiguousarray(path,
170                                                              ↪ dtype=np.float64), state)
171         logsigs_time.append(logsig_t)
172     return np.array(logsigs_time) # (batch, logsig_dim)
173
174 # -----
175 # MAIN LOOP
176 # -----
177 # 1) If varying \depth, build X_fixed one time here; otherwise will generate
178 ↪ inside the loop.
179 if parameter_to_vary == 'depth':
180     initial_condition = torch.zeros(d, dtype=torch.float64)
181     X_fixed = generate_brownian_motion(
182         num_samples, len_x,
183         drift=drift,
184         volatility=volatility,
185         initial_condition=initial_condition,
186         time_increment=time_increment
187     ) # (num_samples, len_x, d)
188
189 m = len(parameter_list)
190 fig, axs = plt.subplots(m, 1, figsize=(10, 5 * m))
191 colours = ['tab:blue', 'tab:orange', 'tab:green', 'tab:purple', 'tab:red']

```

```

190
191 # Dictionaries to store every batch of paths and every reconstruction
192 X_dict = {} # X_dict[param_val] = torch.Tensor of shape (1000,
    ↪ len_x, d)
193 X_np_dict = {} # X_np_dict[param_val] = NumPy array (1000, len_x, d)
194 projections_dict = {} # projections_dict[param_val][method] = (1000,
    ↪ k_method)
195 reconstructions_dict = {} # reconstructions_dict[param_val][method] = (1000,
    ↪ n_features_method)
196
197 for i, parameter_value in enumerate(parameter_list):
198     # 2) Update whichever parameter we're varying
199     if parameter_to_vary == 'depth':
200         depth = parameter_value
201     elif parameter_to_vary == 'drift':
202         drift = parameter_value
203     elif parameter_to_vary == 'volatility':
204         volatility = parameter_value
205     elif parameter_to_vary == 'time increment':
206         time_increment = parameter_value
207     elif parameter_to_vary == 'path length':
208         len_x = parameter_value
209     elif parameter_to_vary == 'path dimension':
210         d = parameter_value
211
212     # 3) Generate X for this iteration
213     if parameter_to_vary == 'depth':
214         # Reuse X_fixed whenever we vary depth
215         X = X_fixed.clone()
216     else:
217         # Regenerate a fresh batch because len_x, drift, etc. may have changed
218         initial_condition = torch.zeros(d, dtype=torch.float64)
219         X = generate_brownian_motion(
220             num_samples, len_x,
221             drift=drift,
222             volatility=volatility,
223             initial_condition=initial_condition,
224             time_increment=time_increment
225         )
226
227     # 4) Store this iteration's paths
228     X_dict[parameter_value] = X.clone()
229     X_np = X.numpy()

```

```

230 X_np_dict[parameter_value] = X_np.copy()
231
232 # 5) Prepare feature spaces
233 X_flat_for_param = X_np.reshape(num_samples, -1) # shape
234   ↪ (num_samples, len_x*d)
235 sig_for_param = batch_signature(X_np, depth) # shape
236   ↪ (num_samples, n_sig)
237 sig_time_for_param = batch_signature_time(X_np, depth, time_increment) #
238   ↪ (num_samples, n_sig_time)
239 logsig_for_param = batch_logsignature(X_np, depth) #
240   ↪ (num_samples, n_logsig)
241 logsig_time_for_param = batch_logsignature_time(X_np, depth,
242   ↪ time_increment) # (num_samples, n_logsig_time)
243
244 # 6) Create entries in both dicts
245 projections_dict[parameter_value] = {}
246 reconstructions_dict[parameter_value] = {}
247
248 # STORE RECONSTRUCTIONS
249
250 # (a) Standard PCA on flattened paths:
251 eigen_flat, proj_flat, recon_flat = perform_pca(X_flat_for_param, num_pcs)
252 projections_dict[parameter_value]["standard"] = proj_flat # shape
253   ↪ = (num_samples, k_std)
254 reconstructions_dict[parameter_value]["standard"] = recon_flat # shape
255   ↪ = (num_samples, len_x*d)
256
257 # (b) Signature PCA:
258 eigen_sig, proj_sig, recon_sig = perform_pca(sig_for_param, num_pcs)
259 projections_dict[parameter_value]["signature"] = proj_sig #
260   ↪ (num_samples, k_sig)
261 reconstructions_dict[parameter_value]["signature"] = recon_sig #
262   ↪ (num_samples, n_sig)
263
264 # (c) Signature+Time PCA:
265 eigen_sig_time, proj_sig_time, recon_sig_time =
266   ↪ perform_pca(sig_time_for_param, num_pcs)
267 projections_dict[parameter_value]["signature+time"] = proj_sig_time #
268   ↪ (num_samples, k_sig_time)
269 reconstructions_dict[parameter_value]["signature+time"] = recon_sig_time
270   ↪ # (num_samples, n_sig_time)
271
272 # (d) Logsignature PCA:

```

```

261 eigen_logsig, proj_logsig, recon_logsig = perform_pca(logsig_for_param,
    ↪ num_pcs)
262 projections_dict[parameter_value]["logsignature"] = proj_logsig      #
    ↪ (num_samples, k_logsig)
263 reconstructions_dict[parameter_value]["logsignature"] = recon_logsig  #
    ↪ (num_samples, n_logsig)
264
265 # (e) Logsignature+Time PCA:
266 eigen_logsig_time, proj_logsig_time, recon_logsig_time =
    ↪ perform_pca(logsig_time_for_param, num_pcs)
267 projections_dict[parameter_value]["logsignature+time"] = proj_logsig_time
    ↪ # (num_samples, k_logsig_time)
268 reconstructions_dict[parameter_value]["logsignature+time"] =
    ↪ recon_logsig_time # (num_samples, n_logsig_time)
269
270 # PLOTTING
271
272 # Choose the correct Axes object (handle single vs. multiple subplots)
273 if m == 1:
274     ax = axs
275 else:
276     ax = axs[i]
277
278 ax.set_title(f"{parameter_to_vary.capitalize()} = {parameter_value}")
279 ax.set_xlabel("Principal Component")
280 ax.set_ylabel("Eigenvalue / Max Eigenvalue")
281 ax.set_xticks(np.arange(1, num_pcs + 1))
282 ax.grid(True)
283
284 # Standard PCA plot
285 ax.plot(
286     np.arange(1, num_pcs + 1),
287     eigen_flat[:num_pcs] / np.max(eigen_flat),
288     label="Standard PCA",
289     marker='o',
290     color=colours[0]
291 )
292
293 # Signature PCA plot
294 ax.plot(
295     np.arange(1, num_pcs + 1),
296     eigen_sig[:num_pcs] / np.max(eigen_sig),
297     label="Signature PCA",

```

```

298         marker='o',
299         color=colours[1]
300     )
301
302     # Signature+Time PCA plot
303     ax.plot(
304         np.arange(1, num_pcs + 1),
305         eigen_sig_time[:num_pcs] / np.max(eigen_sig_time),
306         label="Signature+Time PCA",
307         marker='o',
308         color=colours[2]
309     )
310
311     # Logsignature PCA plot
312     ax.plot(
313         np.arange(1, num_pcs + 1),
314         eigen_logsig[:num_pcs] / np.max(eigen_logsig),
315         label="Logsignature PCA",
316         marker='o',
317         color=colours[3]
318     )
319
320     # Logsignature+Time PCA plot
321     ax.plot(
322         np.arange(1, num_pcs + 1),
323         eigen_logsig_time[:num_pcs] / np.max(eigen_logsig_time),
324         label="Logsignature+Time PCA",
325         marker='o',
326         color=colours[4]
327     )
328
329     # Print the first 4 cumulative explained-variance ratios
330     print(f"\n[{parameter_to_vary.capitalize()} = {parameter_value}]")
331     print("    Standard PCA explained variance ratios:      ",
332           cumulative_explained_variance(eigen_flat, num_pcs))
333     print("    Signature PCA explained variance ratios:      ",
334           cumulative_explained_variance(eigen_sig, num_pcs))
335     print("    Signature+Time PCA explained variance ratios:  ",
336           cumulative_explained_variance(eigen_sig_time, num_pcs))
337     print("    Logsignature PCA explained variance ratios:    ",
338           cumulative_explained_variance(eigen_logsig, num_pcs))
339     print("    Logsignature+Time PCA explained variance ratios:",
340           cumulative_explained_variance(eigen_logsig_time, num_pcs))

```

```

341
342     ax.legend()
343
344 plt.suptitle(
345     f"Distribution of Normalised Eigenvalues\n" +
346     f"across {parameter_to_vary.replace('_', ' ').title()} Levels",
347     fontsize=16
348 )
349 plt.tight_layout(rect=[0, 0.03, 1, 0.97])
350 plt.show()
351
352 # Restore all defaults:
353 len_x = len_x_default
354 drift = drift_default
355 volatility = volatility_default
356 time_increment = time_increment_default
357 depth = depth_default
358 d = d_default

```

## 7.2.2 Signature Inversion with Standard Gradient Descent

```

1  import numpy as np
2  import torch
3  import signatory # Make sure you have this installed: pip install signatory
4
5  def invert_signature(
6      sig_target_np,
7      n_steps,
8      spatial_dim,
9      depth,
10     time_increment=1.0,
11     lr=0.1,
12     n_iter=2000
13 ):
14     """
15     Invert a given signature vector into a path using gradient descent.
16
17     Parameters:
18         sig_target_np: numpy array of shape (signature_length,)
19         n_steps: int, number of steps in the path (without basepoint)
20         spatial_dim: int, spatial dimension of the path
21         depth: int, signature depth
22         time_increment: float, spacing for time augmentation channel

```



```

23         lr: float, learning rate for optimizer
24         n_iter: int, number of optimization iterations
25
26     Returns:
27         recovered_spatial: np.array of shape (n_steps, spatial_dim)
28         losses: list of loss values during optimization
29     """
30
31     # Convert target signature vector to torch tensor, add batch dimension
32     sig_target = torch.tensor(sig_target_np, dtype=torch.float32).unsqueeze(0)
33
34     # Initialise spatial path as a random walk WITHOUT basepoint
35     spatial_noise = np.cumsum(np.random.randn(n_steps,
36         ↪ spatial_dim).astype(np.float32), axis=0)
37
38     # Add basepoint at origin (first point fixed)
39     spatial_path = np.vstack([np.zeros((1, spatial_dim), dtype=np.float32),
40         ↪ spatial_noise]) # shape (n_steps + 1, d)
41
42     # Time augmentation vector
43     n_aug = spatial_path.shape[0]
44     time = (time_increment * np.arange(n_aug)).reshape(-1,
45         ↪ 1).astype(np.float32)
46
47     # Combine time and spatial into full path with time as first channel
48     init_path = np.concatenate([time, spatial_path], axis=1) # shape (n_steps
49         ↪ + 1, spatial_dim + 1)
50
51     y = torch.tensor(init_path, dtype=torch.float32).unsqueeze(0) # shape (1,
52         ↪ n_steps + 1, spatial_dim + 1)
53
54     # Only optimise spatial part, exclude time channel
55     y_spatial = y[:, :, 1:].clone().detach().requires_grad_(True) # shape (1,
56         ↪ n_steps + 1, spatial_dim)
57
58     optimizer = torch.optim.Adam([y_spatial], lr=lr)
59
60     losses = []
61
62     for iter_idx in range(n_iter):
63         optimizer.zero_grad()
64
65         # Fix basepoint (first spatial point) at zero during optimisation

```

```

60     with torch.no_grad():
61         y_spatial[:, 0, :] = 0.0
62
63         # Rebuild full path including time channel
64         y_full = torch.cat([y[:, :, :1], y_spatial], dim=2) # shape (1,
        ↪ n_steps+1, spatial_dim+1)
65
66         # Compute signature at specified depth
67         sig_y = signatory.signature(y_full, depth)
68
69         # Loss = L2 norm between candidate signature and target signature
70         loss = torch.norm(sig_y - sig_target)
71         loss.backward()
72         optimizer.step()
73
74         losses.append(loss.item())
75
76         if iter_idx % 200 == 0 or iter_idx == n_iter - 1:
77             print(f"Iter {iter_idx}: Loss = {loss.item():.6f}")
78
79         # Return recovered spatial path excluding the basepoint
80         recovered_spatial = y_spatial.squeeze(0).detach().cpu().numpy()[1:] #
        ↪ shape (n_steps, spatial_dim)
81
82     return recovered_spatial, losses
83
84
85 # === Example usage ===
86
87 index = 320 # path index
88
89 # Save the shape from the data for that specific series
90 n_steps, spatial_dim = X_dict[param_val][index].shape
91
92 recovered_spatial, loss_history = invert_signature(
93     reconstructed_signature,
94     n_steps,
95     spatial_dim,
96     depth,
97     time_increment=1.0,
98     lr=0.1,
99     n_iter=2000
100 )

```

### 7.2.3 Sparse Signature Inversion with Gradient Descent

```
1  import numpy as np
2  import torch
3  import signatory
4  import matplotlib.pyplot as plt
5
6  def compute_selected_positions(spatial_dim: int, depth: int):
7      """
8      Return a sorted list of 0-based indices in the flattened signature
9      vector for multi-indices  $(1, \dots, 1, k)$ ,  $k > 1$  up to given depth.
10     """
11     channels = spatial_dim + 1
12     selected = []
13     for L in range(1, depth+1):
14         base = sum(channels**l for l in range(1, L))
15         for k in range(2, channels+1):
16             pos = base + (k - 1)
17             selected.append(pos)
18     return sorted(selected)
19
20
21 def invert_signature_stochastic(
22     sig_target_np: np.ndarray,
23     n_steps: int,
24     spatial_dim: int,
25     depth: int,
26     selected_positions: list,
27     sample_size: int = 10,
28     time_increment: float = 1.0,
29     lr: float = 0.1,
30     n_iter: int = 2000
31 ):
32     """
33     Invert a signature by matching only a random subset of `sample_size`
34     coefficients (from selected_positions) each iteration.
35     """
36     device = 'cuda' if torch.cuda.is_available() else 'cpu'
37
38     # full target signature → tensor
39     sig_target = torch.tensor(sig_target_np, dtype=torch.float32,
40                               ↪ device=device)
```

```

41 # init spatial path
42 spatial_noise = np.cumsum(
43     np.random.randn(n_steps, spatial_dim).astype(np.float32),
44     axis=0
45 )
46 spatial_path = np.vstack([np.zeros((1, spatial_dim), dtype=np.float32),
47     spatial_noise])
48 T_aug = spatial_path.shape[0]
49 time_chan = (time_increment * np.arange(T_aug, dtype=np.float32))[:, None]
50
51 init = np.concatenate([time_chan, spatial_path], axis=1) # (T_aug, d+1)
52 y = torch.tensor(init, dtype=torch.float32, device=device).unsqueeze(0)
53
54 y_spatial = y[:, :, 1:].clone().detach().requires_grad_(True)
55 optimizer = torch.optim.Adam([y_spatial], lr=lr)
56 losses = []
57
58 Nsel = len(selected_positions)
59 assert sample_size <= Nsel, "sample_size must be number of selected
    ↪ positions"
60
61 for it in range(1, n_iter+1):
62     optimizer.zero_grad()
63
64     # fix basepoint
65     with torch.no_grad():
66         y_spatial[:, 0, :] = 0.0
67
68     # rebuild full path
69     y_full = torch.cat([y[:, :, :1], y_spatial], dim=2)
70     sig_full = signatory.signature(y_full, depth).squeeze(0) # shape
    ↪ (sig_len,)
71
72     # randomly sample a mini-batch of coefficient positions
73     batch_idx = np.random.choice(selected_positions, size=sample_size,
    ↪ replace=False)
74     batch_idx = torch.tensor(batch_idx, dtype=torch.long, device=device)
75
76     pred_batch = sig_full[batch_idx] # (sample_size,)
77     targ_batch = sig_target[batch_idx] # (sample_size,)
78
79     loss = torch.norm(pred_batch - targ_batch)
80     loss.backward()

```

```

81         optimizer.step()
82
83         losses.append(loss.item())
84         if it % 200 == 0 or it == 1 or it == n_iter:
85             print(f"Iter {it:4d} stochastic-loss = {loss.item():.6f}")
86
87     rec = y_spatial.detach().cpu().squeeze(0).numpy()[1:]
88     return rec, losses
89
90
91     # === Example usage ===
92
93     # 1) Compute the positions you care about once
94     selected_positions = compute_selected_positions(spatial_dim, depth)
95
96     # 2) Call the stochastic inverter, sampling 15 coefficients each iteration
97     recovered_spatial, loss_history = invert_signature_stochastic(
98         sig_target_np      = reconstructed_signature,
99         n_steps             = n_steps,
100        spatial_dim         = spatial_dim,
101        depth               = depth,
102        selected_positions   = selected_positions,
103        sample_size          = 15,          # number of coefficients per iteration
104        time_increment       = 1.0,
105        lr                   = 0.025,
106        n_iter               = 3000
107    )
108
109     # 3) Plot original vs recovered
110     T = recovered_spatial.shape[0]
111     time = np.arange(T)
112     fig, axs = plt.subplots(spatial_dim, 1, figsize=(8, 2.5*spatial_dim),
113         ↪ sharex=True)
114     for i in range(spatial_dim):
115         axs[i].plot(time,
116             X_dict[param_val][index][:, i],
117             label='Original')
118         axs[i].plot(time,
119             recovered_spatial[:, i],
120             '--', label='Recovered')
121         axs[i].set_ylabel(f'x{i+1}')
122         axs[i].legend()
123     axs[-1].set_xlabel('Time step')

```

```
123 plt.tight_layout()
124 plt.show()
```