Imperial College London

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

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 $p = 3$.

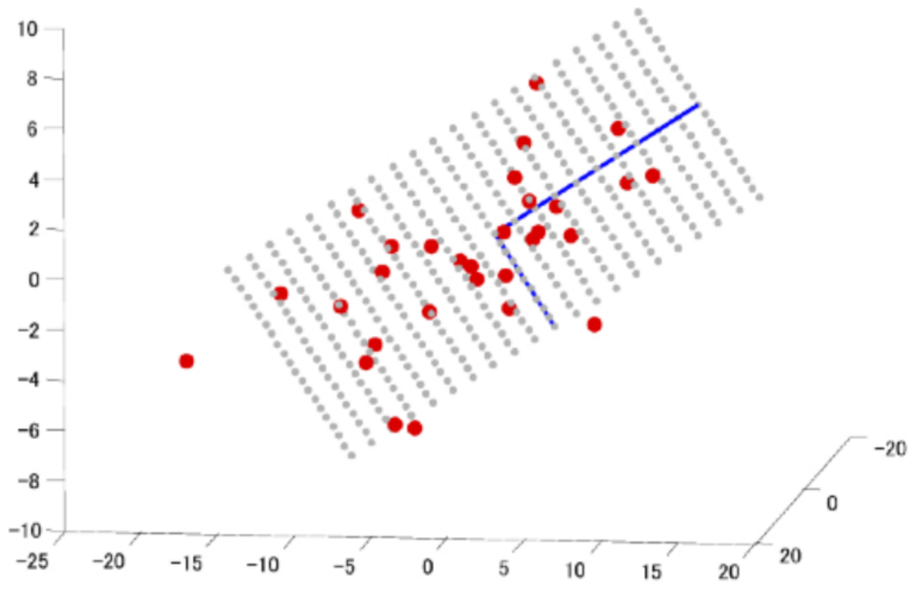

Figure 1: 3D data on a 2D subspace. Red: data points, Blue: principal components. [4]

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 $x_1, ..., x_n \in \mathbb{R}^p$ assumed to have zero mean (since we can always shift the data to $x_1 - \bar{x}, ..., x_n - \bar{x}$ which has zero mean), we construct the $p \times n$ data matrix $X = [x_1|...|x_n]$. The projection of the data $x_1, ..., x_n$ onto a unit vector $a = (a_1, ..., a_p)^\top$ is $(x_1|...|x_n)^T a = X^\top a =: Z$. Note that $\bar{x} = 0$ means $\bar{z} = 0$.

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

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^{th} largest (or simply k^{th}) principal component $a^{(k)}$ can be found iteratively, by solving

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

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

$$Ca = \lambda a,$$

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,

$$\mathrm{Var}(Xa^{(k)}) = {a^{(k)}}^\top C a^{(k)} = {a^{(k)}}^\top \lambda a^{(k)} = \lambda({a^{(k)}}^\top a^{(k)}) = \lambda$$

so the percentage variance explained by the kth principal component equals

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

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 $t_1 < t_2 < ... < t_T$, a time series $\{x_{t_1}, x_{t_2}, ..., x_{t_T}\} \subset R^d$ 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.

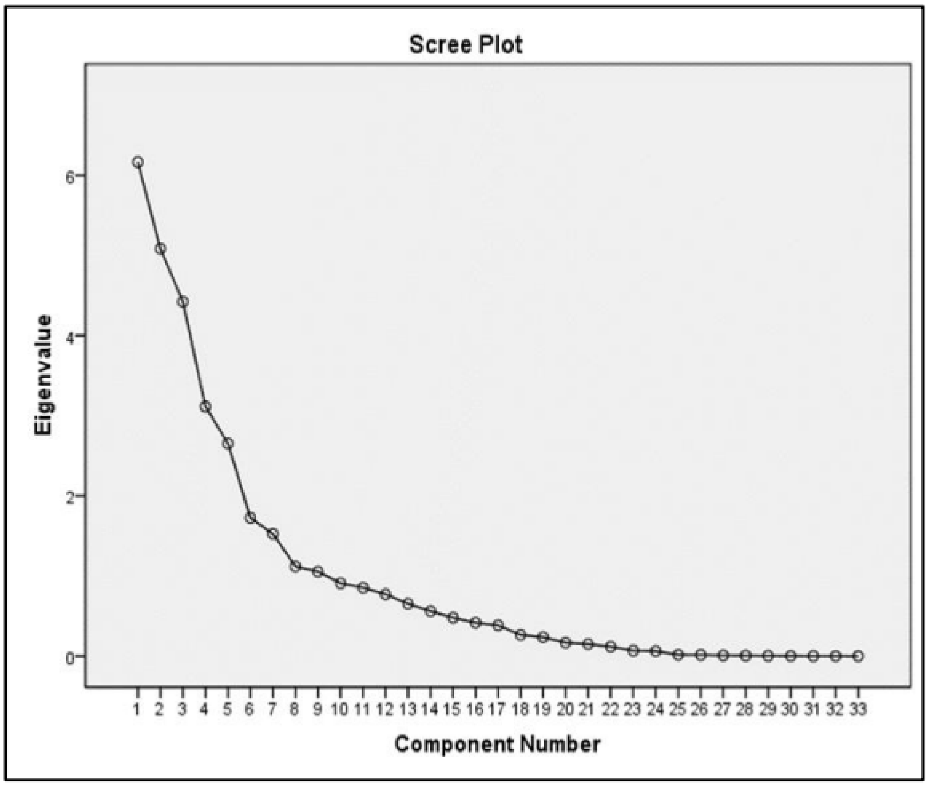

Figure 2: Scree plot of eigenvalues for an example covariance matrix. An elbow occurs around the 9th eigenvalue (1.054). The first 9 eigenvalues explain 81.39% of the total variance [5].

1.2 Kernel Principal Component Analysis

Standard PCA attempts to find a 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]:

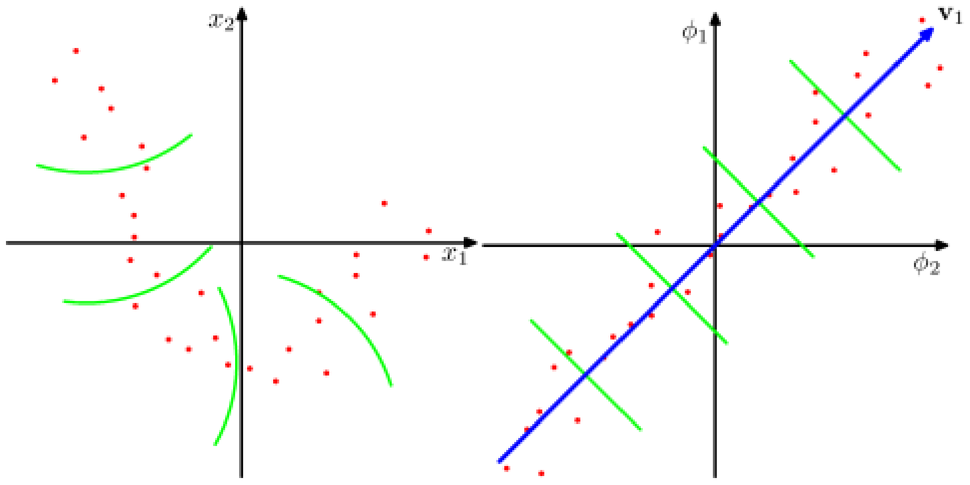

Figure 3: Data lying on a nonlinear subspace.

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 $\phi : \mathbb{R}^p \to \mathbb{R}^q$ 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 $\{x_1, \ldots, x_n\}$ 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:
$$K\boldsymbol{\alpha} = n\lambda\boldsymbol{\alpha}$$
where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)^\top$.
7. A kernel function $k : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}$ 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 α_i^k are the corresponding coefficients.

Literature Review

2.1 Dictionaries

The essence of dictionary-based compresssion 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 $x : [a, b] \to V$ 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:

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

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

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

which can be seen as a measure of interaction between variables i_1, ..., i_k [9].
The set of k^{th}-order coefficients can be expressed more compactly as

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

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

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

and suppose

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

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

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

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

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

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:

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

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 $\varphi : \mathbb{N} \cup \{0\} \to \mathbb{R}_+$, we define the φ-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 π, as explained in [11]. Thus, for two paths $x : [a, b] \to R^d$, $y : [c, d] \to R^d$, we work with the unweighted signature kernel:

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

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]:

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

with boundary conditions

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

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

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

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 $k : X \times X \to \mathbb{R}$ is positive semidefinite if for any $n \in \mathbb{N}$ and any points $x_1, \ldots, x_n \in X$, the Gram matrix

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

is positive semidefinite. That is, for any real coefficients $c_1, \ldots, c_n$, 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 $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a positive semidefinite kernel, then is a unique RKHS 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

There are two properties of the signature kernel that make it a natural choice for time series data. It is universal, which means that for every compact subset $\mathcal{K} \subset C$, $\text{Span}\{k(x, \cdot) : x \in \mathcal{K}\}$ can approximate any continuous function on path space $f : C \to R$ arbitrarily well, and it characterises probability distributions of time series distributions uniquely.

3.3.2 Compression with Signature Kernel PCA

We are now ready to present our proposed method. The main idea is to associate a 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 $x_1, \ldots, x_n$.

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 (λ_k, α_k) to $G\alpha_k = \lambda_k \alpha_k$ and store them in descending order of eigenvalues $\lambda_1 \geq \ldots \geq \lambda_n$. 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 $k = 1, ..., n$. 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 $k = 1, ..., K$, where K is the selected number of principal components, the kth principal component score of path x_j is given by

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

   where $a^k = (a_1^k, ..., a_n^k)$. 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 $X = \{x_1, ..., x_N\}$ 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 $\phi_1 : x \to k(x, \cdot)$, we will instead consider the feature map $\phi_2 : x \to S(x)$, 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 $(1, \ldots, 1, k)$, $k \in \{2, \ldots, d\}$ 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 $S^{1,\ldots,1,k}(x) = S^{1,\ldots,1,k}(y)$ 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

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

then the loss function is

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

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 $\hat{S}^{\leq N}_{\text{SPARSE}}(x)$.

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

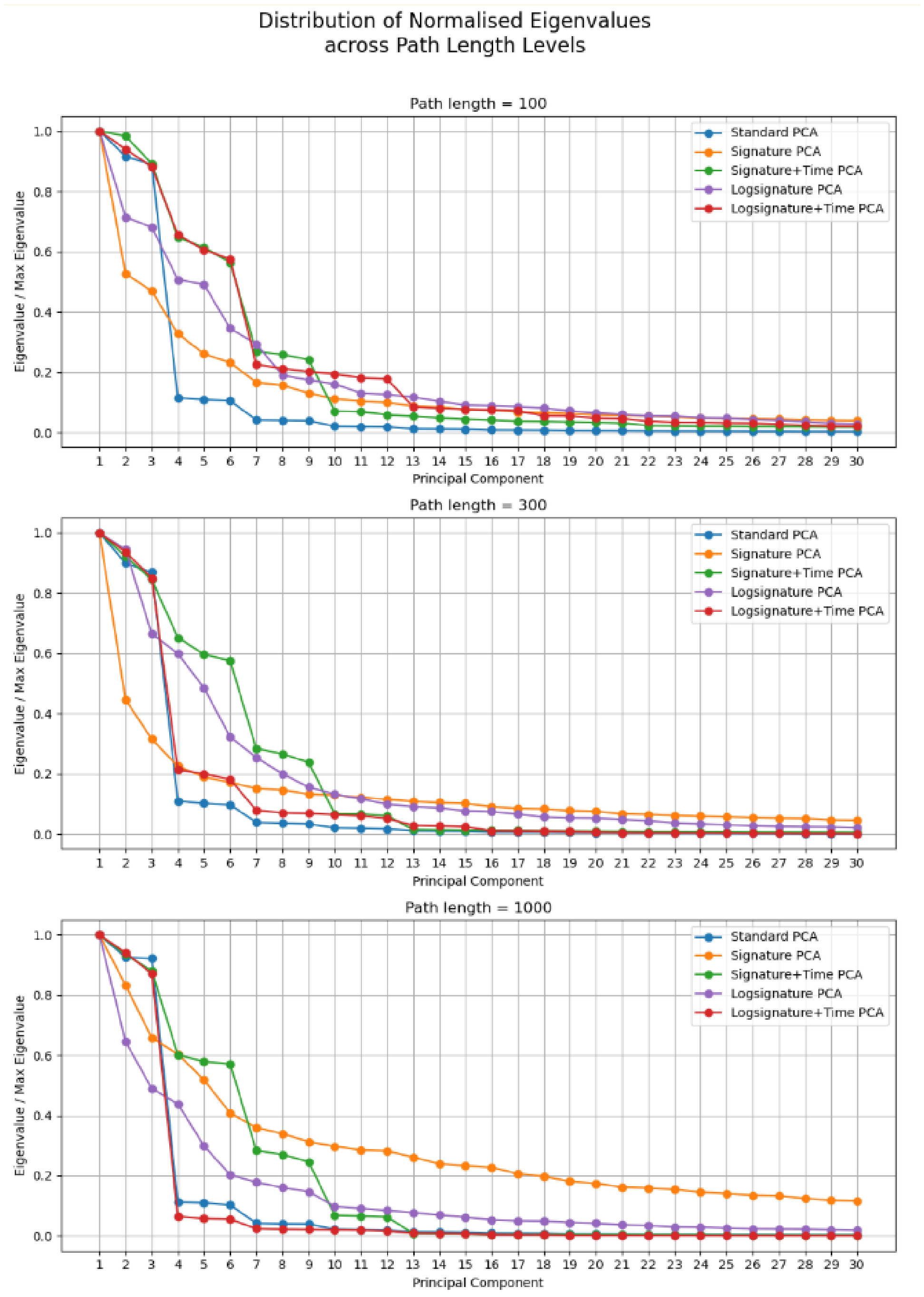

Figure 4: Scree plot showing the distribution of Gram matrix eigenvalues for different path lengths.

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 “effective dimensionality” of the signature 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

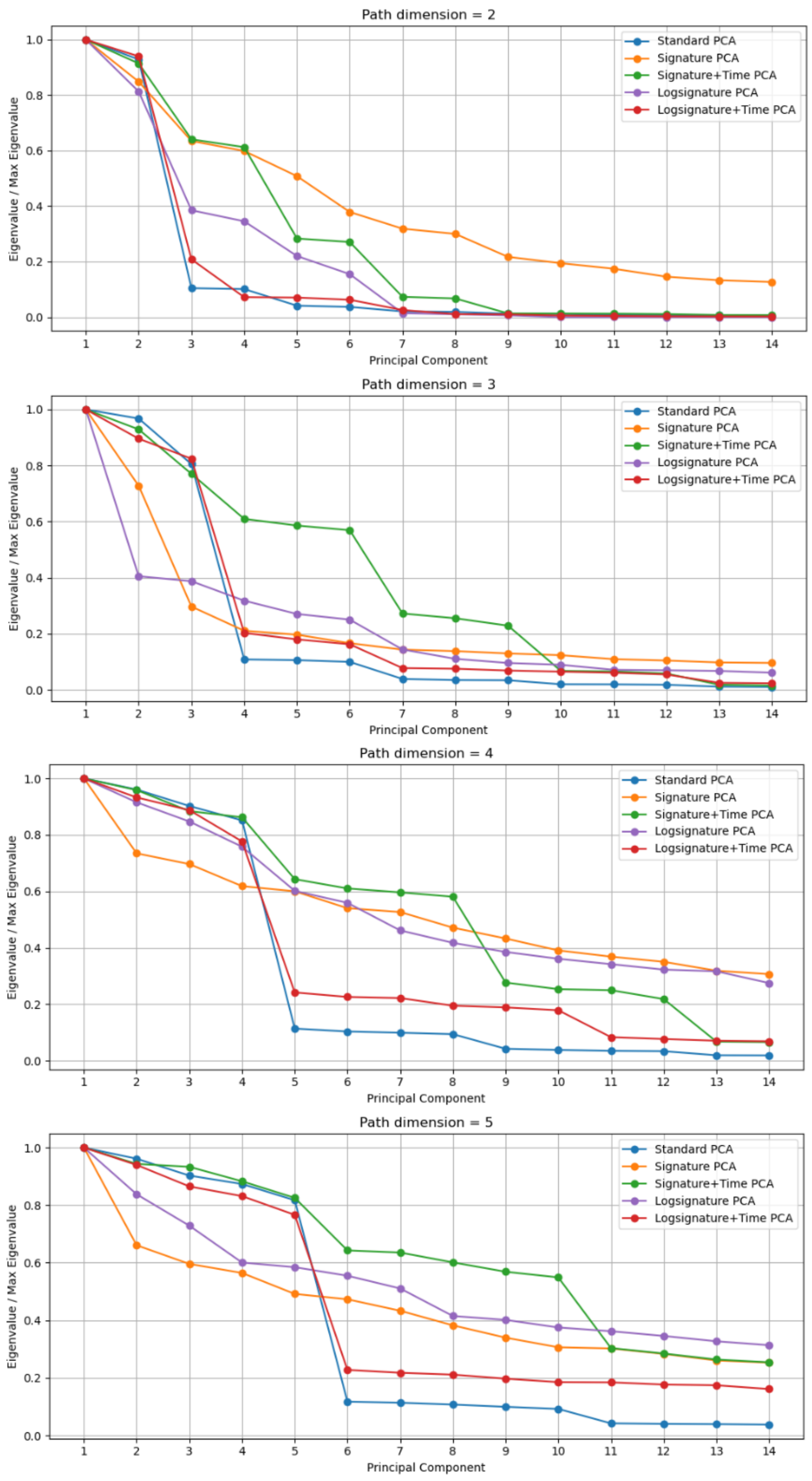

Figure 5: Scree plot showing the distribution of Gram matrix eigenvalues for different path dimensions.

4.1.3 Signature Depth

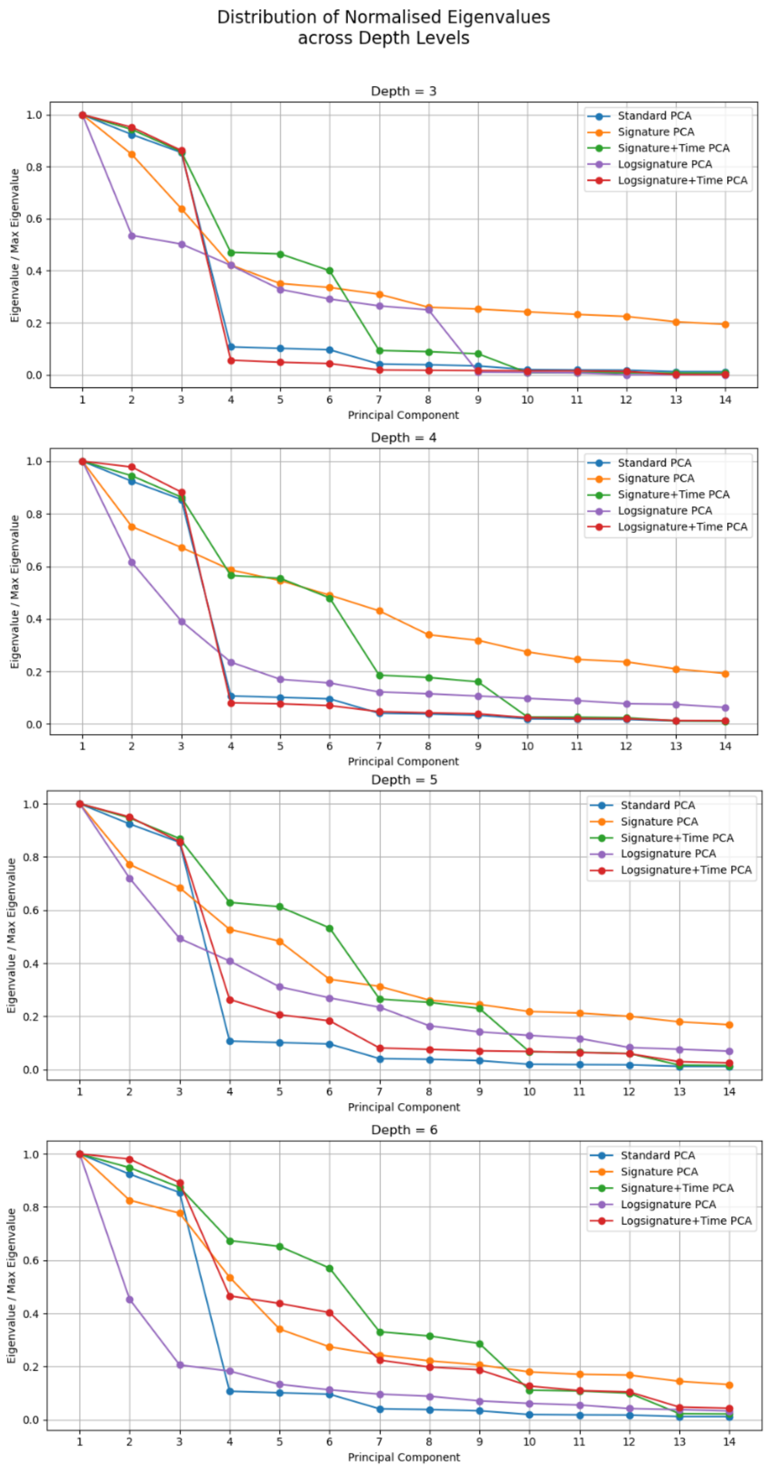

Figure 6: Scree plot showing the distribution of Gram matrix eigenvalues for different signature truncation depths.

We also observe that the number of nontrivial (i.e. not very close to zero) eigenvalues increases with depth, and this may be due to higher depth signatures capturing more details of each path, which are not negligible for jagged paths like Brownian motion.

4.2 Compression Ratio and Recovery Accuracy

4.2.1 Signature Inversion with Gradient Descent

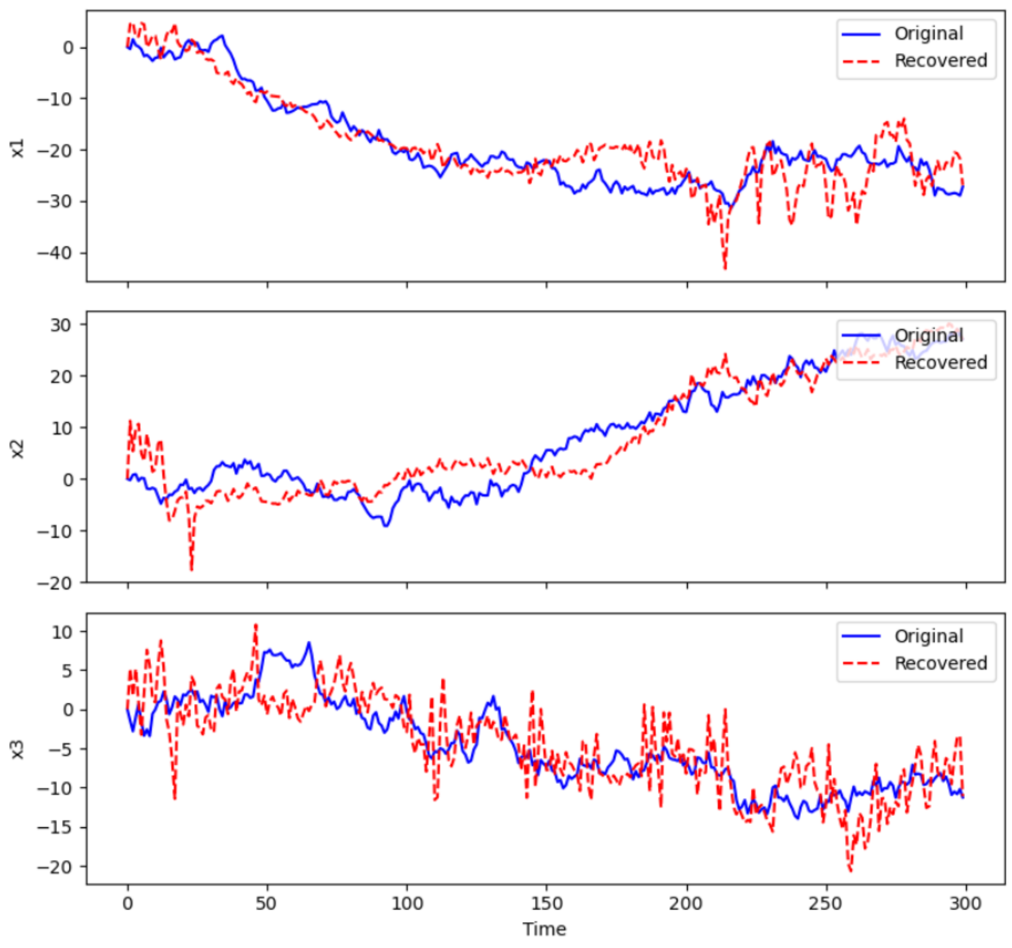

Figure 7: Path reconstruction using standard gradient descent with a learning rate of 0.01 and 4000 iterations on signature coefficients up to depth 5.

Compression Ratio: 12.45
Note that each path $x_1, ..., x_n$ 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 $D_{N,d} = (d+1) + ... + (d+1)^N$, 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

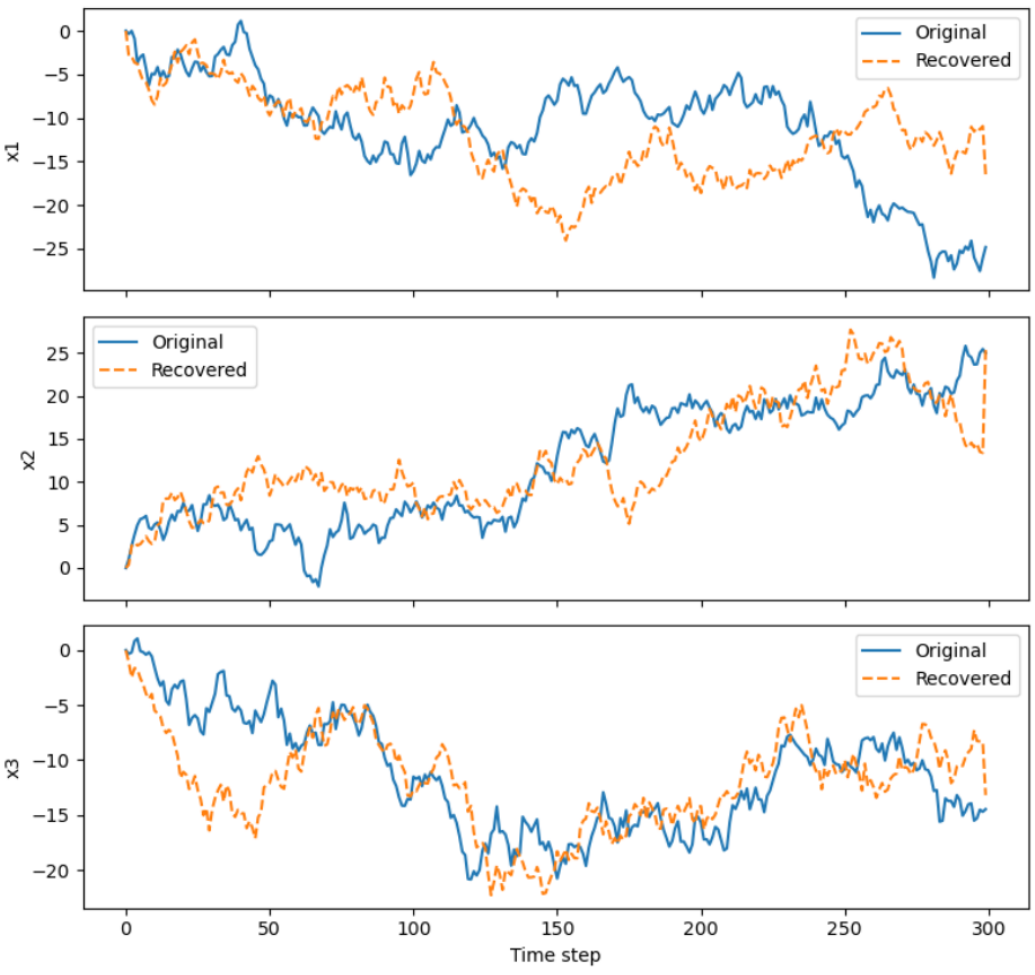

Figure 8: Path reconstruction using gradient descent on selected signature coefficients up to depth 5.

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 $d + d + \cdots + d = dN$, because we extract d coefficients from each level of the signature up to level N, namely those corresponding to the multi-indices:

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

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.

[4] Arthur Gretton. *Introduction to RKHS, and some simple kernel algorithms.* 2019. URL: 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.

[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`.

[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 α, β, ζ such that,

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

The concatenation $\alpha * \beta$ is known as a reduction of γ. 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 γ is referred to as an irreducible reduction of γ.

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 α and β 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 $x : [0, T] \to \mathbb{R}^d$ 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 $\mathcal{P} = \{0 = t_0 < t_1 < \cdots < t_n = T\}$ of the interval $[0, T]$.

7.2 Code

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

7.2.1 Eigenvalue Scree Plots

```python
import iisignature
import numpy as np
import torch
import matplotlib.pyplot as plt

# Default parameters
num_samples     = 1000
len_x_default   = 300  # number of points in each path
len_x           = len_x_default
drift_default   = 0
drift           = drift_default
volatility_default = 1
volatility      = volatility_default
time_increment_default = 1
time_increment = time_increment_default
depth_default   = 5  # signature truncation level
depth           = depth_default
d_default       = 3  # path dimension
d               = d_default
num_pcs_default = 14   # number of principal components to keep
num_pcs         = num_pcs_default

# Parameter lists (for exploring how PCA-eigenvalue spectra change)
path_dimension_list = [2, 3, 4, 5]
depth_list          = [3, 4, 5]
drift_list          = [0.01, 0.1, 1, 10]
volatility_list     = [0.001, 0.01, 0.1, 1, 10]
time_increment_list = [0.001, 0.01, 0.1, 1, 10]
path_length_list    = [100, 300, 1000]

parameter_to_vary = 'path dimension'

if parameter_to_vary == 'depth':
    parameter_list = depth_list
elif parameter_to_vary == 'drift':
    parameter_list = drift_list
elif parameter_to_vary == 'volatility':
    parameter_list = volatility_list
```

```
elif parameter_to_vary == 'time increment':
    parameter_list = time_increment_list
elif parameter_to_vary == 'path length':
    parameter_list = path_length_list
elif parameter_to_vary == 'path dimension':
    parameter_list = path_dimension_list
else:
    raise ValueError("Unknown parameter_to_vary.")


# ---------------------------------------------------------------------------
# 1) Generate a batch of Brownian-motion{type paths (spatial only)
# ---------------------------------------------------------------------------
def generate_brownian_motion(num_samples, len_x, drift, volatility,
                             initial_condition=None, time_increment=0.1):
    """
    Returns a torch.Tensor of shape (num_samples, len_x, d):
      X[*, 0, :] = 0  (if initial_condition is zero)
      X[*, t, :] = cumulative sum of (dW + drift*t) up to index t.
    We do NOT append time here; this is purely spatial.
    """
    t_x = torch.linspace(0, (len_x - 1) * time_increment, len_x,
                         dtype=torch.float64).view(1, len_x, 1)  # (1, len_x,
                         ↪  1)
    dW = torch.randn(num_samples, len_x - 1, d, dtype=torch.float64) \
         * torch.sqrt(torch.tensor(time_increment, dtype=torch.float64)) \
         * volatility
    dW = torch.cat([
        torch.zeros(num_samples, 1, d, dtype=torch.float64),
        dW
    ], dim=1)  # (num_samples, len_x, d)
    X = torch.cumsum(dW, dim=1) + drift * t_x
    if initial_condition is not None:
        X = X + initial_condition.view(1, 1, d)
    return X  # (num_samples, len_x+1, d)


# ---------------------------------------------------------------------------
# 2) PCA helper (returns reconstruction too)
# ---------------------------------------------------------------------------
def perform_pca(X: np.ndarray, n_components: int):
    """
    Input:
      X: (n_samples, n_features)
    Returns:
      eigenvalues:       (n_features,) sorted descending
      projections:      (n_samples, k)         # principal-component
      ↪  coefficients
      X_reconstructed:   (n_samples, n_features) # back-projection into
      ↪  original basis
    """
    mean_X = np.mean(X, axis=0, keepdims=True)         # (1, n_features)
    X_centered = X - mean_X                            # (n_samples,
    ↪  n_features)
    cov = np.cov(X_centered, rowvar=False)             # (n_features,
    ↪  n_features)
    eigenvalues, eigenvectors = np.linalg.eigh(cov)
    idx_desc = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idx_desc]
    eigenvectors = eigenvectors[:, idx_desc]             # columns are sorted
    ↪  eigenvectors
    n_features = eigenvalues.shape[0]
    k = min(n_components, n_features)
    W = eigenvectors[:, :k]                              # (n_features, k)
    projections = X_centered @ W                          # (n_samples, k)
    X_reconstructed = projections @ W.T + mean_X          # (n_samples,
    ↪  n_features)
    return eigenvalues, projections, X_reconstructed


def cumulative_explained_variance(eigenvalues: np.ndarray, k):
    total_var = np.sum(eigenvalues)
    explained = np.cumsum(eigenvalues[:k])
    return [round(float(ev) / total_var, 4) for ev in explained]


# ------------------------------------------------------------------------------
# 3) Augmentation routines (basepoint first, then time)
# ------------------------------------------------------------------------------
def add_basepoint_zero_spatial_only(paths_spatial: np.ndarray) -> np.ndarray:
    batch, len_x_, d_ = paths_spatial.shape
    zero_spatial = np.zeros((batch, 1, d_), dtype=paths_spatial.dtype)
    return np.concatenate([zero_spatial, paths_spatial], axis=1)  # (batch,
    ↪  len_x+1, d)


```

```python
def add_time_after_basepoint(paths_spatial_with_base: np.ndarray,
                             time_increment: float) -> np.ndarray:
    batch, len_x_plus1, d_ = paths_spatial_with_base.shape
    times = np.arange(len_x_plus1, dtype=np.float64).reshape(1, len_x_plus1, 1) * time_increment
    times = np.tile(times, (batch, 1, 1))  # (batch, len_x+1, 1)
    return np.concatenate([times, paths_spatial_with_base], axis=2)  # (batch, len_x+1, d+1)


# ----------------------------------------------------------------------------
# 4) Batch signature / logsignature routines
# ----------------------------------------------------------------------------
def batch_signature(paths_spatial: np.ndarray, depth: int) -> np.ndarray:
    paths_with_base = add_basepoint_zero_spatial_only(paths_spatial)  # (batch, len_x+1, d)
    sigs = []
    for path in paths_with_base:
        sig = iisignature.sig(np.ascontiguousarray(path, dtype=np.float64), depth)
        sigs.append(sig)
    return np.array(sigs)  # (batch, sig_dim)


def batch_logsignature(paths_spatial: np.ndarray, depth: int) -> np.ndarray:
    paths_with_base = add_basepoint_zero_spatial_only(paths_spatial)  # (batch, len_x+1, d)
    state = iisignature.prepare(paths_spatial.shape[2], depth)
    logsigs = []
    for path in paths_with_base:
        logsig = iisignature.logsig(np.ascontiguousarray(path, dtype=np.float64), state)
        logsigs.append(logsig)
    return np.array(logsigs)  # (batch, logsig_dim)


def batch_signature_time(paths_spatial: np.ndarray,
                         depth: int,
                         time_increment: float) -> np.ndarray:
    paths_with_base = add_basepoint_zero_spatial_only(paths_spatial)  # (batch, len_x+1, d)
    paths_time = add_time_after_basepoint(paths_with_base, time_increment)  # (batch, len_x+1, d+1)
    sigs_time = []
    for path in paths_time:
        sig_t = iisignature.sig(np.ascontiguousarray(path, dtype=np.float64), depth)
        sigs_time.append(sig_t)
    return np.array(sigs_time)  # (batch, sig_dim)


def batch_logsignature_time(paths_spatial: np.ndarray,
                            depth: int,
                            time_increment: float) -> np.ndarray:
    paths_with_base = add_basepoint_zero_spatial_only(paths_spatial)  # (batch, len_x+1, d)
    paths_time = add_time_after_basepoint(paths_with_base, time_increment)  # (batch, len_x+1, d+1)
    state = iisignature.prepare(paths_time.shape[2], depth)  # channels = d+1
    logsigs_time = []
    for path in paths_time:
        logsig_t = iisignature.logsig(np.ascontiguousarray(path, dtype=np.float64), state)
        logsigs_time.append(logsig_t)
    return np.array(logsigs_time)  # (batch, logsig_dim)


# -----------------------------------------------------------------------------
#  MAIN LOOP
# -----------------------------------------------------------------------------

# 1) If varying \depth," build X_fixed one time here; otherwise will generate inside the loop.
if parameter_to_vary == 'depth':
    initial_condition = torch.zeros(d, dtype=torch.float64)
    X_fixed = generate_brownian_motion(
        num_samples, len_x,
        drift=drift,
        volatility=volatility,
        initial_condition=initial_condition,
        time_increment=time_increment
    )  # (num_samples, len_x, d)

m = len(parameter_list)
fig, axs = plt.subplots(m, 1, figsize=(10, 5 * m))
colours = ['tab:blue', 'tab:orange', 'tab:green', 'tab:purple', 'tab:red']
```

```python

# Dictionaries to store every batch of paths and every reconstruction
X_dict = {}             # X_dict[param_val] = torch.Tensor of shape (1000,
↪   len_x, d)
X_np_dict = {}          # X_np_dict[param_val] = NumPy array (1000, len_x, d)
projections_dict = {}   # projections_dict[param_val][method] = (1000,
↪   k_method)
reconstructions_dict = {}  # reconstructions_dict[param_val][method] = (1000,
↪   n_features_method)

for i, parameter_value in enumerate(parameter_list):
    # 2) Update whichever parameter we're varying
    if parameter_to_vary == 'depth':
        depth = parameter_value
    elif parameter_to_vary == 'drift':
        drift = parameter_value
    elif parameter_to_vary == 'volatility':
        volatility = parameter_value
    elif parameter_to_vary == 'time increment':
        time_increment = parameter_value
    elif parameter_to_vary == 'path length':
        len_x = parameter_value
    elif parameter_to_vary == 'path dimension':
        d = parameter_value

    # 3) Generate X for this iteration
    if parameter_to_vary == 'depth':
        # Reuse X_fixed whenever we vary depth
        X = X_fixed.clone()
    else:
        # Regenerate a fresh batch because len_x, drift, etc. may have changed
        initial_condition = torch.zeros(d, dtype=torch.float64)
        X = generate_brownian_motion(
            num_samples, len_x,
            drift=drift,
            volatility=volatility,
            initial_condition=initial_condition,
            time_increment=time_increment
        )

    # 4) Store this iteration's paths
    X_dict[parameter_value] = X.clone()
    X_np = X.numpy()
    X_np_dict[parameter_value] = X_np.copy()

    # 5) Prepare feature spaces
    X_flat_for_param     = X_np.reshape(num_samples, -1)                    # shape
    ↪   (num_samples, len_x*d)
    sig_for_param        = batch_signature(X_np, depth)                     # shape
    ↪   (num_samples, n_sig)
    sig_time_for_param   = batch_signature_time(X_np, depth, time_increment)   #
    ↪   (num_samples, n_sig_time)
    logsig_for_param     = batch_logsignature(X_np, depth)                  #
    ↪   (num_samples, n_logsig)
    logsig_time_for_param = batch_logsignature_time(X_np, depth,
    ↪   time_increment) # (num_samples, n_logsig_time)

    # 6) Create entries in both dicts
    projections_dict[parameter_value]     = {}
    reconstructions_dict[parameter_value] = {}

    #  STORE RECONSTRUCTIONS

    # (a) Standard PCA on flattened paths:
    eigen_flat, proj_flat, recon_flat = perform_pca(X_flat_for_param, num_pcs)
    projections_dict[parameter_value]["standard"] = proj_flat          # shape
    ↪   = (num_samples, k_std)
    reconstructions_dict[parameter_value]["standard"] = recon_flat    # shape
    ↪   = (num_samples, len_x*d)

    # (b) Signature PCA:
    eigen_sig, proj_sig, recon_sig = perform_pca(sig_for_param, num_pcs)
    projections_dict[parameter_value]["signature"] = proj_sig         #
    ↪   (num_samples, k_sig)
    reconstructions_dict[parameter_value]["signature"] = recon_sig    #
    ↪   (num_samples, n_sig)

    # (c) Signature+Time PCA:
    eigen_sig_time, proj_sig_time, recon_sig_time =
    ↪   perform_pca(sig_time_for_param, num_pcs)
    projections_dict[parameter_value]["signature+time"] = proj_sig_time    #
    ↪   (num_samples, k_sig_time)
    reconstructions_dict[parameter_value]["signature+time"] = recon_sig_time
    ↪   # (num_samples, n_sig_time)

    # (d) Logsignature PCA:
```

```python
        eigen_logsig, proj_logsig, recon_logsig = perform_pca(logsig_for_param,
        ↪  num_pcs)
        projections_dict[parameter_value]["logsignature"] = proj_logsig       #
        ↪  (num_samples, k_logsig)
        reconstructions_dict[parameter_value]["logsignature"] = recon_logsig  #
        ↪  (num_samples, n_logsig)

        # (e) Logsignature+Time PCA:
        eigen_logsig_time, proj_logsig_time, recon_logsig_time =
        ↪  perform_pca(logsig_time_for_param, num_pcs)
        projections_dict[parameter_value]["logsignature+time"] = proj_logsig_time
        ↪  # (num_samples, k_logsig_time)
        reconstructions_dict[parameter_value]["logsignature+time"] =
        ↪  recon_logsig_time  # (num_samples, n_logsig_time)

        #  PLOTTING

        # Choose the correct Axes object (handle single vs. multiple subplots)
        if m == 1:
            ax = axs
        else:
            ax = axs[i]

        ax.set_title(f"{parameter_to_vary.capitalize()} = {parameter_value}")
        ax.set_xlabel("Principal Component")
        ax.set_ylabel("Eigenvalue / Max Eigenvalue")
        ax.set_xticks(np.arange(1, num_pcs + 1))
        ax.grid(True)

        # Standard PCA plot
        ax.plot(
            np.arange(1, num_pcs + 1),
            eigen_flat[:num_pcs] / np.max(eigen_flat),
            label="Standard PCA",
            marker='o',
            color=colours[0]
        )

        # Signature PCA plot
        ax.plot(
            np.arange(1, num_pcs + 1),
            eigen_sig[:num_pcs] / np.max(eigen_sig),
            label="Signature PCA",
            marker='o',
            color=colours[1]
        )

        # Signature+Time PCA plot
        ax.plot(
            np.arange(1, num_pcs + 1),
            eigen_sig_time[:num_pcs] / np.max(eigen_sig_time),
            label="Signature+Time PCA",
            marker='o',
            color=colours[2]
        )

        # Logsignature PCA plot
        ax.plot(
            np.arange(1, num_pcs + 1),
            eigen_logsig[:num_pcs] / np.max(eigen_logsig),
            label="Logsignature PCA",
            marker='o',
            color=colours[3]
        )

        # Logsignature+Time PCA plot
        ax.plot(
            np.arange(1, num_pcs + 1),
            eigen_logsig_time[:num_pcs] / np.max(eigen_logsig_time),
            label="Logsignature+Time PCA",
            marker='o',
            color=colours[4]
        )

        # Print the first 4 cumulative explained-variance ratios
        print(f"\n[{parameter_to_vary.capitalize()} = {parameter_value}]")
        print("  Standard PCA explained variance ratios:          ",
              cumulative_explained_variance(eigen_flat, num_pcs))
        print("  Signature PCA explained variance ratios:         ",
              cumulative_explained_variance(eigen_sig, num_pcs))
        print("  Signature+Time PCA explained variance ratios:    ",
              cumulative_explained_variance(eigen_sig_time, num_pcs))
        print("  Logsignature PCA explained variance ratios:      ",
              cumulative_explained_variance(eigen_logsig, num_pcs))
        print("  Logsignature+Time PCA explained variance ratios:",
              cumulative_explained_variance(eigen_logsig_time, num_pcs))
```

```python

    ax.legend()

plt.suptitle(
    f"Distribution of Normalised Eigenvalues\n" +
    f"across {parameter_to_vary.replace('_',' ').title()} Levels",
    fontsize=16
)
plt.tight_layout(rect=[0, 0.03, 1, 0.97])
plt.show()

# Restore all defaults:
len_x = len_x_default
drift = drift_default
volatility = volatility_default
time_increment = time_increment_default
depth = depth_default
d = d_default
```

7.2.2 Signature Inversion with Standard Gradient Descent

```python
import numpy as np
import torch
import signatory  # Make sure you have this installed: pip install signatory

def invert_signature(
    sig_target_np,
    n_steps,
    spatial_dim,
    depth,
    time_increment=1.0,
    lr=0.1,
    n_iter=2000
):
    """
    Invert a given signature vector into a path using gradient descent.

    Parameters:
        sig_target_np: numpy array of shape (signature_length,)
        n_steps: int, number of steps in the path (without basepoint)
        spatial_dim: int, spatial dimension of the path
        depth: int, signature depth
        time_increment: float, spacing for time augmentation channel
        lr: float, learning rate for optimizer
        n_iter: int, number of optimization iterations

    Returns:
        recovered_spatial: np.array of shape (n_steps, spatial_dim)
        losses: list of loss values during optimization
    """

    # Convert target signature vector to torch tensor, add batch dimension
    sig_target = torch.tensor(sig_target_np, dtype=torch.float32).unsqueeze(0)

    # Initialise spatial path as a random walk WITHOUT basepoint
    spatial_noise = np.cumsum(np.random.randn(n_steps,
    ↪   spatial_dim).astype(np.float32), axis=0)

    # Add basepoint at origin (first point fixed)
    spatial_path = np.vstack([np.zeros((1, spatial_dim), dtype=np.float32),
    ↪   spatial_noise])  # shape (n_steps + 1, d)

    # Time augmentation vector
    n_aug = spatial_path.shape[0]
    time = (time_increment * np.arange(n_aug)).reshape(-1,
    ↪   1).astype(np.float32)

    # Combine time and spatial into full path with time as first channel
    init_path = np.concatenate([time, spatial_path], axis=1)  # shape (n_steps
    ↪   + 1, spatial_dim + 1)

    y = torch.tensor(init_path, dtype=torch.float32).unsqueeze(0)  # shape (1,
    ↪   n_steps + 1, spatial_dim + 1)

    # Only optimise spatial part, exclude time channel
    y_spatial = y[:, :, 1:].clone().detach().requires_grad_(True)  # shape (1,
    ↪   n_steps + 1, spatial_dim)

    optimizer = torch.optim.Adam([y_spatial], lr=lr)

    losses = []

    for iter_idx in range(n_iter):
        optimizer.zero_grad()

        # Fix basepoint (first spatial point) at zero during optimisation
        with torch.no_grad():
            y_spatial[:, 0, :] = 0.0

        # Rebuild full path including time channel
        y_full = torch.cat([y[:, :, :1], y_spatial], dim=2)  # shape (1, n_steps+1, spatial_dim+1)

        # Compute signature at specified depth
        sig_y = signatory.signature(y_full, depth)

        # Loss = L2 norm between candidate signature and target signature
        loss = torch.norm(sig_y - sig_target)
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

        if iter_idx % 200 == 0 or iter_idx == n_iter - 1:
            print(f"Iter {iter_idx}: Loss = {loss.item():.6f}")

    # Return recovered spatial path excluding the basepoint
    recovered_spatial = y_spatial.squeeze(0).detach().cpu().numpy()[1:]  # shape (n_steps, spatial_dim)

    return recovered_spatial, losses


# === Example usage ===

index = 320  # path index

# Save the shape from the data for that specific series
n_steps, spatial_dim = X_dict[param_val][index].shape

recovered_spatial, loss_history = invert_signature(
    reconstructed_signature,
    n_steps,
    spatial_dim,
    depth,
    time_increment=1.0,
    lr=0.1,
    n_iter=2000
)
```

7.2.3 Sparse Signature Inversion with Gradient Descent

```python
import numpy as np
import torch
import signatory
import matplotlib.pyplot as plt

def compute_selected_positions(spatial_dim: int, depth: int):
    """
    Return a sorted list of 0-based indices in the flattened signature
    vector for multi-indices (1,...,1,k), k > 1 up to given depth.
    """
    channels = spatial_dim + 1
    selected = []
    for L in range(1, depth+1):
        base = sum(channels**l for l in range(1, L))
        for k in range(2, channels+1):
            pos = base + (k - 1)
            selected.append(pos)
    return sorted(selected)


def invert_signature_stochastic(
    sig_target_np: np.ndarray,
    n_steps: int,
    spatial_dim: int,
    depth: int,
    selected_positions: list,
    sample_size: int = 10,
    time_increment: float = 1.0,
    lr: float = 0.1,
    n_iter: int = 2000
):
    """
    Invert a signature by matching only a random subset of `sample_size`
    coefficients (from selected_positions) each iteration.
    """
    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    # full target signature → tensor
    sig_target = torch.tensor(sig_target_np, dtype=torch.float32,
    ↪  device=device)

    # init spatial path
    spatial_noise = np.cumsum(
        np.random.randn(n_steps, spatial_dim).astype(np.float32),
        axis=0
    )
    spatial_path = np.vstack([np.zeros((1, spatial_dim), dtype=np.float32),
                              spatial_noise])
    T_aug = spatial_path.shape[0]
    time_chan = (time_increment * np.arange(T_aug, dtype=np.float32))[:, None]

    init = np.concatenate([time_chan, spatial_path], axis=1)  # (T_aug, d+1)
    y    = torch.tensor(init, dtype=torch.float32, device=device).unsqueeze(0)

    y_spatial = y[:, :, 1:].clone().detach().requires_grad_(True)
    optimizer = torch.optim.Adam([y_spatial], lr=lr)
    losses = []

    Nsel = len(selected_positions)
    assert sample_size <= Nsel, "sample_size must be  number of selected
    ↪  positions"

    for it in range(1, n_iter+1):
        optimizer.zero_grad()

        # fix basepoint
        with torch.no_grad():
            y_spatial[:, 0, :] = 0.0

        # rebuild full path
        y_full = torch.cat([y[:, :, :1], y_spatial], dim=2)
        sig_full = signatory.signature(y_full, depth).squeeze(0)  # shape
        ↪  (sig_len,)

        # randomly sample a mini-batch of coefficient positions
        batch_idxs = np.random.choice(selected_positions, size=sample_size,
        ↪  replace=False)
        batch_idxs = torch.tensor(batch_idxs, dtype=torch.long, device=device)

        pred_batch = sig_full[batch_idxs]           # (sample_size,)
        targ_batch = sig_target[batch_idxs]         # (sample_size,)

        loss = torch.norm(pred_batch - targ_batch)
        loss.backward()
        optimizer.step()

        losses.append(loss.item())
        if it % 200 == 0 or it == 1 or it == n_iter:
            print(f"Iter {it:4d}  stochastic-loss = {loss.item():.6f}")

    rec = y_spatial.detach().cpu().squeeze(0).numpy()[1:]
    return rec, losses


# === Example usage ===

# 1) Compute the positions you care about once
selected_positions = compute_selected_positions(spatial_dim, depth)

# 2) Call the stochastic inverter, sampling 15 coefficients each iteration
recovered_spatial, loss_history = invert_signature_stochastic(
    sig_target_np       = reconstructed_signature,
    n_steps             = n_steps,
    spatial_dim         = spatial_dim,
    depth               = depth,
    selected_positions  = selected_positions,
    sample_size         = 15,       # number of coefficients per iteration
    time_increment      = 1.0,
    lr                  = 0.025,
    n_iter              = 3000
)

# 3) Plot original vs recovered
T = recovered_spatial.shape[0]
time = np.arange(T)
fig, axs = plt.subplots(spatial_dim, 1, figsize=(8, 2.5*spatial_dim),
 ↪  sharex=True)
for i in range(spatial_dim):
    axs[i].plot(time,
                X_dict[param_val][index][:, i],
                label='Original')
    axs[i].plot(time,
                recovered_spatial[:, i],
                '--', label='Recovered')
    axs[i].set_ylabel(f'x{i+1}')
    axs[i].legend()
axs[-1].set_xlabel('Time step')
```

```
plt.tight_layout()
plt.show()
```