Algorithm 1004: The Iisignature Library: Efficient Calculation of Iterated-Integral Signatures and Log Signatures

JEREMY F. REIZENSTEIN, Alan Turing Institute and University of Warwick, UK
BENJAMIN GRAHAM, Facebook AI Research, France

Iterated-integral signatures and log signatures are sequences calculated from a path that characterizes its shape. They originate from the work of K. T. Chen and have become important through Terry Lyons's theory of differential equations driven by rough paths, which is an important developing area of stochastic analysis. They have applications in statistics and machine learning, where there can be a need to calculate finite parts of them quickly for many paths. We introduce the signature and the most basic information (displacement and signed areas) that it contains. We present algorithms for efficiently calculating these signatures. For log signatures this requires consideration of the structure of free Lie algebras. We benchmark the performance of the algorithms. The methods are implemented in C++ and released as a Python extension package, which also supports differentiation. In combination with a machine learning library (Tensorflow, PyTorch, or Theano), this allows end-to-end learning of neural networks involving signatures.

CCS Concepts: • **Computing methodologies** → **Machine learning approaches**; • **Mathematics of computing** → *Mathematical software performance*;

Additional Key Words and Phrases: Iterated-integral signatures, rough paths, free Lie algebras, machine learning, Python library

ACM Reference format:

Jeremy F. Reizenstein and Benjamin Graham. 2020. Algorithm 1004: The Iisignature Library: Efficient Calculation of Iterated-Integral Signatures and Log Signatures. *ACM Trans. Math. Softw.* 46, 1, Article 8 (March 2020), 21 pages.

<https://doi.org/10.1145/3371237>

1 INTRODUCTION

iisignature is a Python package that calculates the iterated-integral signatures and log signatures of paths. The signature is an object that is crucial in the mathematical theory of rough paths [Lyons 1998], and the calculations have proved to be useful in machine learning applications, particularly classification problems where the data itself is a stream or a path in space, ranging from an application to online Chinese handwriting recognition [Graham 2013] to skeleton-based human action recognition [Yang et al. 2017]. Other domains where the data has this form include signals from EEG and other medical monitors, sound, and financial time series, where a vector quantity

Authors' addresses: J. F. Reizenstein and B. Graham, Facebook AI Research, 1 Rathbone Square, London W1T 1FB; emails: {reizenstein, benjamingraham}@fb.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

0098-3500/2020/03-ART8 \$15.00

<https://doi.org/10.1145/3371237>

is varying in time. Often the samples can be noisy and can have varying length, and both local and global structure can be important. A survey of such applications is given in Chevyrev and Kormilitzin [2016].

The focus of the `iisignature` package is the calculation of the signature for piecewise-linear paths in relatively low-dimensional spaces. In these cases, paths typically move in all their dimensions, so only rarely will elements of the signature be zero. We call this the *dense* case. We study the mathematical properties of the free Lie algebra to implement a range of algorithms. We also benchmark the performance of these algorithms and provide an efficient open source implementation.

An existing open source library for calculating signatures is the `esig` package from CoRoPa [Lyons et al. 2010]. However, this package is optimized to operate on a very particular type of path: ones that live in a high-dimensional space, but that only move in certain combinations of input dimensions. Most of the elements of the signatures are zero. We call this the *sparse* case. While `esig`'s algorithm can be applied in the dense case, we will see that dense algorithms can be much more efficient. This article is organized as follows. The rest of this section introduces signatures, log signatures, and the library. Signature algorithms are discussed in Section 2. Log signature methods are introduced in Section 3, the direct method is discussed in Section 4, and the projection method in Section 5. Considerations around the implementation are presented in Section 6. Indicative timings are given in Section 7 and memory usages in Section 8. We briefly discuss other functionality provided by the library in Section 9.

1.1 What Is the Signature of a Path?

The iterated-integral signature of a continuous path is an infinite sequence. It is used in the mathematical theory of differential equations driven by paths. In these problems, a path is the input signal controlling the evolution of a system. One example is a sound wave (the input path) making an eardrum vibrate (the system). It turns out that the signature is exactly the information about a path that you need to predict how the output of the system will behave, using a generalization of Taylor's theorem. It is natural that the signature would also be the right information to extract from a path if we want a machine learning algorithm to understand the shape of the path.

A d -dimensional continuous path is given by a function γ from an interval $[a, b] \subset \mathbb{R}$ to $\mathbb{R}^d$. We restrict attention to cases where γ has bounded variation—in other words it has finite length. Its signature depends on its appearance, i.e., the image of the function γ , and the direction it was created, but not on its speed, i.e., the time parameterization. If a path is modified by adding or removing a section that is a detour that exactly retraces itself, then its signature does not change. If the path has a time dimension along which it always increases (e.g., it is the graph of a function of time), then this exact backtracking is impossible and so any two different paths will have different signatures. For paths that are continuously differentiable at all but finitely many places, such as the paths that we deal with in `iisignature` which are piecewise linear between a set of specified points, the signatures of two different paths that contain no exact backtracking will differ [Chen 1958].

The signature is divided into units called levels, numbered 0, 1, 2, and so on. We cannot store the whole signature of a path on a computer; rather, we calculate a certain number of levels of it. The more levels of a signature are known, the more precisely the shape of the path is determined. If a path changes very slightly, the first few levels of its signature will also only change very slightly. If a path is moved (translated) but retains its shape, its signature will not change.

The number of elements of level m of the signature of a d -dimensional path is d^m . They are the values of iterated integrals that consist of m nested integrals, and they are labeled with m numbers, each corresponding to one of the dimensions. To distinguish these numbers that label the dimensions from other numbers, we write them in bold and in blue, and we will call them

Fig. 1. Concatenating paths and the corresponding total displacements and total signed areas.

letters. For example, a two-dimensional path might be given in coordinates as $(\gamma_1(t), \gamma_2(t))$ as t varies from a to b . Its signature is a function denoted by $X_{a,b}^\gamma$. Level 3 of its signature has eight elements, called $X_{a,b}^\gamma(\mathbf{111})$, $X_{a,b}^\gamma(\mathbf{112})$, and so on. The one indexed by the word $\mathbf{122}$ is the Riemann-Stieltjes integral:

$$X_{a,b}^\gamma(\mathbf{122}) = \int_{t_1=a}^b \int_{t_2=a}^{t_1} \int_{t_3=a}^{t_2} d\gamma_1(t_3) d\gamma_2(t_2) d\gamma_2(t_1), \quad (1)$$

which, if the components are continuously differentiable, is just

$$X_{a,b}^\gamma(\mathbf{122}) = \int_{t_1=a}^b \int_{t_2=a}^{t_1} \int_{t_3=a}^{t_2} \gamma_1'(t_3) dt_3 \gamma_2'(t_2) dt_2 \gamma_2'(t_1) dt_1. \quad (2)$$

The signature can be defined inductively on the length of the word. The signature of the empty word is the single value in level 0, and it is defined to always be 1. If w is a word and $i \in \{1, 2, \dots, d\}$, then $X_{a,b}^\gamma(wi)$ is defined as the Riemann-Stieltjes integral $\int_a^b X_{a,t}^\gamma(w) d\gamma_i(t)$. If γ_i is differentiable, this is just $\int_a^b X_{a,t}^\gamma(w) \gamma_i'(t) dt$. Level m can be thought of as taking values in $(\mathbb{R}^d)^{\otimes m}$, which is a d^m -dimensional real vector space. In this form, the signature is an element of the tensor algebra $T(\mathbb{R}^d) = \bigoplus_{m=0}^{\infty} (\mathbb{R}^d)^{\otimes m}$, the infinite direct sum of tensor powers of $\mathbb{R}^d$.

The information in the first level of the signature is the total displacement of the path, i.e., the direction and distance from its starting point to its ending point. Specifically, the element indexed by letter i is $X_{a,b}^\gamma(i) = \int_{t_1=a}^b d\gamma_i(t_1) = \gamma_i(b) - \gamma_i(a)$, which is the displacement in dimension i . The information that the second level of the signature adds is the *signed area* of the path projected in each plane, as will be explained in the next section. Higher levels of the signature provide more detailed information about the path's shape.

1.2 Signed Area

For a two-dimensional path γ , the following linear combination of elements of level 2 of its signature is known as its *signed area* and is interpreted as the area between the path and the straight line from its beginning to its end, accounting for sign:

$$\frac{1}{2} X_{a,b}^\gamma(\mathbf{12}) - \frac{1}{2} X_{a,b}^\gamma(\mathbf{21}). \quad (3)$$

It turns out that given just level 1 of the signature and the signed area, we can recover the whole of level 2 of the signature, so we say that the information carried by the first two levels of the signature is the total displacement of the path (in the first level, which is two numbers) and its signed area (which is one number). Figure 1 shows this information for two straight lines and their combination, which contains area.

The following is an intuitive definition of the signed area of a path in the plane given by Equation (3). For a simple closed path, i.e., one that has no self-intersections and ends where it starts,

Fig. 2. (a) A complicated closed path with $+$ and $-$ signs indicating for each enclosed region the direction of circumnavigation. (b) Two handwritten 0 digits from the Pendigits training dataset; as is common, both digits are drawn counterclockwise so the generated area is positive. (c) A handwritten digit 8 from the Pendigits training set; although the object is large, the signed area is small due to cancellation between the positive and negative parts.

Fig. 3. Histogram of areas of the first stroke of each 0 and 8 in the training portion of the Pendigits dataset.

the signed area is the area bounded by the curve, accounting for sign, by Green's theorem. For a closed path, which is one that ends where it starts, the signed area is the sum of the signed areas of the regions bounded by the path, which is the area times the number of times the path goes round that region in an anticlockwise manner minus the number of times the path goes round it clockwise (i.e., the winding number). For example, in the path shown in Figure 2(a), regions whose areas count positively are labeled with a $+$ and negatively with a $-$. One region's area counts twice negatively; it is labeled with $--$. For a more general path, its signed area is the signed area of the closed path you get by joining it with a straight line from its end to its start. As an example of how the area can be useful in classifying the shape of the path, consider classifying handwritten digits 0 and 8. Usually these are written with a single stroke that ends near its beginning, so the displacement is insufficient for distinguishing them. However, the encompassed areas are statistically different. The figure 0 is typically formed from a single anticlockwise loop, generating a positive signed area, while the Figure 8 contains two regions with opposite sign, leading to cancellation of the signed area. The diagrams in Figures 2(b) and 2(c) illustrate this. The Pendigits dataset [Alpaydin and Alimoglu 1998] collected the traces of many people writing the digits 0 to 9, and the histogram in Figure 3 shows how different the signed areas of the first (and usually only) strokes of these digits are. This clear separation is an illustration of the potential usefulness of the signature for classification.

1.3 What Is the Log Signature of a Path?

The log signature is a compressed version of the signature. It carries the same information, but in a more compact way. Like the signature itself, it is also divided into levels. Up to level m , the log signature contains fewer numbers than the signature. This is possible because there is some redundancy in the signature. For example, the first two levels of the signature of a two-dimensional path consist of $2 + 2^2 = 6$ numbers, but we saw that this information is the path's total displacement and signed area, which can be stored in three numbers, which are exactly the first two levels of the log signature. In applications, the log signature might be less susceptible to roundoff error. The log signature is defined in terms of the signature, in a way analogous to logarithms of numbers, as we will show explicitly in Section 3, but can be calculated via an independent algorithm.

2 SIGNATURES

Calculating the signature of a path can be done inductively relying on the following two rules:

- If γ is a straight line defined on the interval $[a, b]$, then its signature as a function on words is

$$X_{a,b}^\gamma(i_1 i_2 \dots i_m) = \frac{1}{m!} \prod_{j=1}^m (\gamma_{i_j}(b) - \gamma_{i_j}(a)). \quad (4)$$

Grouped by levels, as tensor powers of $\mathbb{R}^d$, using $x = \gamma(b) - \gamma(a)$ as the displacement, the signature of γ looks like

$$\left(1, x, \frac{x \otimes x}{2!}, \frac{x \otimes x \otimes x}{3!}, \dots\right), \quad (5)$$

where $\otimes$ is the tensor product. Alternatively, if each level is thought of as a vector of numbers, this formula should be read with $\otimes$ denoting the Kronecker product.

- If $a < b < c$, then the result (from Chen [1958]) known as **Chen's identity** allows us to calculate the signature of a path split into two subpaths in terms of the signatures of the subpaths. It states that

$$X_{a,c}^\gamma(i_1 i_2 \dots i_m) = \sum_{j=0}^m X_{a,b}^\gamma(i_1 i_2 \dots i_{j-1}) X_{b,c}^\gamma(i_j i_{j+1} \dots i_m). \quad (6)$$

Grouped by levels, this signature looks like

$$\begin{aligned} (1, X_{a,c}^{(1)}, X_{a,c}^{(2)}, \dots) &= (1, X_{a,b}^{(1)} + X_{b,c}^{(1)}, X_{a,b}^{(2)} + X_{a,b}^{(1)} \otimes X_{b,c}^{(1)} + X_{b,c}^{(2)}, \\ &\quad X_{a,b}^{(3)} + X_{a,b}^{(2)} \otimes X_{b,c}^{(1)} + X_{a,b}^{(1)} \otimes X_{b,c}^{(2)} + X_{b,c}^{(3)}, \dots). \end{aligned} \quad (7)$$

When calculating the signature of a path given as a series of straight-line displacements, we start with the signature of the first displacement (calculated from Equation (4)) and step-by-step concatenate it with the signature of each succeeding displacement using Equation (6). The procedure is summarized in Algorithm 1.

Level m of the signature contains d^m values. Calculating it for a displacement using Equation (4) takes $d + d^m$ multiplications beyond what has already been calculated for lower levels. However, in the signature of a straight line, each level is a symmetric tensor and so level m only contains $\binom{d+m-1}{m}$ distinct values, using the formula for unordered sampling with replacement. It is possible to reorganize the calculations to avoid duplicating operations, but the code becomes more complicated. Only $d + \binom{d+m-1}{m}$ multiplications are required. We implemented this and found it to be slower, so `iisignature` does not use this idea.

ALGORITHM 1: Calculating the signature

```

1: SIGNATURE populated with zeros
2: for each segment SEGMENT in the path do
3:   SEGMENTSIGNATURE  $\leftarrow$  the signature of SEGMENT using Equation (4)
4:   SIGNATURE  $\leftarrow$  the Chen product of SIGNATURE and SEGMENTSIGNATURE using Equation (6)
5: end for
6: return SIGNATURE

```

The signature we wish to return is a value for each word in $\{1, 2, \dots, d\}$ of length less than or equal to m , equivalently a value in truncated tensor space $T_{\leq m}(\mathbb{R}^d) = \mathbb{R} \otimes \mathbb{R}^d \otimes \dots \otimes (\mathbb{R}^d)^m$. The value for the empty word (the first component in the tensor product) is always 1, and so we exclude it from our output. We return the value as a single array of numbers, sorted by level, with the words in alphabetical order within the level. With this ordering, the signature is a single flat sequence of numbers, which we are returning a truncation of.

3 LOG SIGNATURES

Tensor space $T(\mathbb{R}^d)$, in which the signature of a d -dimensional path lives, has a notion of logarithm (Reutenauer [1994], chapter 3), given by

$$\log(1 + T) = \sum_{n \geq 1} \frac{(-1)^{n-1} T^n}{n}, \quad (8)$$

where the multiplication implied by the powers are in the concatenation product. Let S be the set that consists of the signature of every path in $\mathbb{R}^d$ up to level m . S is not the whole of the vector space $T_{\leq m}(\mathbb{R}^d) = \mathbb{R} \oplus (\mathbb{R}^d)^1 \oplus \dots \oplus (\mathbb{R}^d)^{\otimes m}$, although it does *span* $T_{\leq m}(\mathbb{R}^d)$ (see Lemma 8 in Diehl [2013]). In fact, they form a lower-dimensional manifold. The logarithm operation maps this manifold continuously one to one to a linear subspace of $T_{\leq m}(\mathbb{R}^d)$. The representation of the logarithm of the signature in a basis of this subspace is called the **log signature**.

The subspace in which the log signature of a path in $\mathbb{R}^d$ up to level m lives is in fact a realization of the free m -nilpotent Lie algebra of type d , $\mathfrak{n}_{d,m}$. The log signature is like a compressed version of the signature up to the same level. It is, in some sense, a minimal encoding in a linear space: for every value in $\mathfrak{n}_{d,m}$, there is a path with that truncated log signature. The dimensionality of level m of the log signature is given by $\frac{1}{m} \sum_{D|m} \mu(\frac{m}{D}) d^D$, where μ is the Möbius function, a formula known as Witt's formula, and given in Satz 3 of Witt [1937]. This can be approximated by $\frac{1}{m}(d^m - d)$, a formula that is exact when m is prime. The main source for the relevant mathematics is Reutenauer [1994] and an informal introduction is given in Reizenstein [2015].

$\mathfrak{n}_{d,m}$ is a finite dimensional real vector space, but there is no single obvious basis for it. In order to use the log signature as an efficient representation of a path, we need to choose a fixed basis. There are two commonly used bases. They are both *Hall bases* [Hall 1950]. A Hall basis is made up of bracketed expressions, which are built up from letters and a Lie bracket operation¹ $[\cdot, \cdot]$, and it is determined by an ordering of all bracketed expressions. The ordering of single letters in both cases is their natural order $1 < 2 < \dots < d$. Each of the bracketed expressions produces a distinct word, its *foliage*, when all the brackets are removed, and the words that are so obtained are called *Hall words*. The bracketed expression whose Hall word is the word w is denoted P_w , which is a convention that only makes sense if the basis under consideration is understood.

¹When considering a bracketed expression as an element of tensor space, the bracket $[a, b]$ is $ab - ba$.

Table 1. The First Elements of the Two Hall Bases for the Free Lie Algebra on Two Elements Used for Log Signatures of Two-Dimensional Paths

Level	Lyndon Basis	Canonical Hall Basis
1	12	12
2	[1, 2]	[1, 2]
3	[1, [1, 2]] [[1, 2], 2]	[1, [1, 2]] [2, [1, 2]]
4	[1, [1, [1, 2]]] [1, [[1, 2], 2]] [[[1, 2], 2], 2]	[1, [1, [1, 2]]] [2, [1, [1, 2]]] [2, [2, [1, 2]]]
5	[1, [1, [1, [1, 2]]]] [1, [1, [[1, 2], 2]]] [[1, [1, 2]], [1, 2]] [1, [[[1, 2], 2], 2]] [[1, 2], [[1, 2], 2]] [[[[1, 2], 2], 2], 2]	[1, [1, [1, [1, 2]]]] [2, [1, [1, [1, 2]]]] [2, [2, [1, [1, 2]]]] [2, [2, [2, [1, 2]]]] [[1, 2], [1, [1, 2]]] [[1, 2], [2, [1, 2]]]

- The *Lyndon basis* [Shirshov 1953], which is the default in `iisignature`. Each basis element is labeled with a Lyndon word on $\{1, 2, \dots, d\}$, which is a sequence that comes earlier in lexicographic order than any of its *rotations*. (For example, the rotations of **2432** are **2243**, **3224**, and **4322**. **2243** and **1213** are Lyndon words, but **31** and **3224** are not.) Equivalently, a nonempty word is a Lyndon word if whenever it is split into two nonempty parts, the left part comes lexicographically earlier than the right part.
- The standard/canonical Hall basis, which we implement in such a way as to match CoRoPa [Lyons et al. 2010] exactly. The ordering of equal-length expressions $[A, B]$ and $[C, D]$ is defined recursively: $[A, B] < [C, D]$ if either $A < C$ or $(A = C \text{ and } B < D)$.

For the two-dimensional case, the first few elements of each of these bases are displayed in Table 1. In these bases, each basis element is either a letter or a single bracketed expression, whose left and right are basis elements.

Much of the algebra calculations are done once upfront in the `prepare` function. This is a major contribution of `iisignature` and ensures for given d, m and the choice of basis that the calculation is as efficient as possible. This is relevant in machine learning applications where typically many similar calculations are required.

We return the log signature as a single array of numbers, sorted in increasing order of level, using the Hall word order within the level. With this ordering, as for signatures, the log signature is an infinite series of numbers of which we are returning a truncation. We provide two different calculation methods for it, described in the next two sections.

4 CALCULATING LOG SIGNATURES DIRECTLY

The log signature of a straight-line displacement is just the displacement itself in level 1, and zero in every other level. This corresponds to the fact that the signature of a straight line as given by Equation (4) is the tensor space exponential of the displacement, and the tensor exponential and logarithm are inverses. The log signature of the concatenation of two paths is the Baker-Campbell-Hausdorff (BCH) product of the log signatures of the two paths. The direct method for calculating the log signature relies on being able to transform the log signature of a path given in terms of one of the bases above to the log signature of that path concatenated with a fixed-line segment, achieved using the BCH product.

The BCH product is an infinite series in bracketed expressions in two indeterminates, which can be formulated in different equivalent ways. The most straightforward ways express all brackets in the form of some Hall basis of the free Lie algebra of $\mathbb{R}^2$. For example, using the Lyndon basis:

$$\text{bch}(a, b) = a + b + \frac{1}{2}[a, b] + \frac{1}{12}[a, [a, b]] + \frac{1}{12}[[a, b], b] + \frac{1}{24}[a, [[a, b], b]] + \dots \quad (9)$$

The coefficients in this expansion up to terms of depth 20 have been calculated and distributed by Casas and Murua [2009a], using their method described in Casas and Murua [2009b]. We distribute their file as part of `iisignature` and read it when necessary.

We can compute the Lie bracket of each pair of basis elements as a combination of other basis elements, and therefore, given two log signatures as combinations of basis elements (the second known to be just a displacement), we can find the expanded expression of their BCH product as a combination of basis elements. By doing this with indeterminates, that is, objects that represent variables, the library develops an internal representation of the formula for getting the truncated log signature of a path combined with a displacement, given the path's truncated log signature and the displacement. This is analogous to finding a formula by combining other formulae using variables in a computer algebra system. This formula alone is enough to iteratively calculate the log signature of the whole path, as illustrated in Algorithm 2.

ALGORITHM 2: C and O method: Calculating the log signature directly

```

1: LOGSIGNATURE populated with zeros
2: for each segment SEGMENT in the path do
3:   Accumulate SEGMENT onto LOGSIGNATURE in a way equivalent to:
       SEGMENTLOGSIGNATURE ← the displacement of SEGMENT padded with zeros
       LOGSIGNATURE ← bch(LOGSIGNATURE, SEGMENTLOGSIGNATURE)
4: end for
5: return LOGSIGNATURE
  
```

As an example, in the case where the Lyndon basis is used, and we are concerned with two dimensions up to level 2, a log signature looks like

$$a = a_0 \mathbf{1} + a_1 \mathbf{2} + a_2 P_{12},$$

where P_{12} is $[\mathbf{1}, \mathbf{2}]$, the single element of the basis in level 2, which is labeled by the Hall word $\mathbf{12}$. A displacement looks like $b = b_0 \mathbf{1} + b_1 \mathbf{2}$. Applying Equation (9), we get that the log signature of the combined signature up to level 2 is

$$\begin{aligned}
 \text{bch}(a, b) &= a + b + \frac{1}{2}[a, b] + \dots \\
 &= (a_0 \mathbf{1} + a_1 \mathbf{2} + a_2 P_{12}) + (b_0 \mathbf{1} + b_1 \mathbf{2}) + \frac{1}{2}[(a_0 \mathbf{1} + a_1 \mathbf{2} + \dots), (b_0 \mathbf{1} + b_1 \mathbf{2})] + \dots \\
 &= (a_0 + b_0) \mathbf{1} + (a_1 + b_1) \mathbf{2} + \left(a_2 + \frac{1}{2}(a_0 b_1 - a_1 b_0)\right) P_{12} + \dots,
 \end{aligned}$$

where the ellipses denote terms that do not contribute to the first two levels of the result. The inductive step of the algorithm to accumulate log signatures by adding linear segments for $d = m = 2$ is shown in Figure 4.

If we go up to level 3, a general log signature looks like

$$a = a_0 \mathbf{1} + a_1 \mathbf{2} + a_2 P_{12} + a_3 P_{112} + a_4 P_{122},$$

with the final algorithm being as shown in Figure 5.

For a general level, the functions to accumulate log signatures, the equivalents of F22 and F23, have a lot of common structure. First, a sequence of monomials in the input elements is constructed in the temporary array t . Higher-order monomials are calculated inductively from other elements of t to deduplicate the necessary multiplications. Then some members of a are incremented by some multiples of some of the temporary variables. Then the first d elements of a are incremented by all elements of b . These functions are branching-free and can be long. The variable a is modified in place to produce the log signature of the extended path. We do not have a good approximation


```

def F22(a, b):
    # Construct monomials of log signature a and displacement b
    t[0] = b[1] * a[0]
    t[1] = b[0] * a[1]
    # Extend log signature in-place
    a[2] += t[0] / 2
    a[2] += -t[1] / 2
    a[0:2] += b[:]

```

Fig. 4. Algorithm to accumulate a new displacement into a log signature in the Lyndon basis with $d = 2$ and $m = 2$.

```

def F23(a, b): # Log signature a and displacement b
    # Calculate monomials of a and b
    t[0] = b[1] * a[0] # Order 2 monomials
    t[1] = b[1] * a[2]
    t[2] = b[0] * a[1]
    t[3] = b[0] * a[2]
    t[4] = b[1] * t[0] # Order 3 monomials calculated from
    t[5] = b[0] * t[0] # t[i], i<=4
    t[6] = b[1] * t[2]
    t[7] = a[0] * t[0]
    t[8] = a[1] * t[0]
    t[9] = b[0] * t[2]
    t[10] = a[0] * t[2]
    t[11] = a[1] * t[2]
    # Extend log signature in-place
    a[2] += t[0]/2 - t[2]/2
    a[3] += -t[3]/2 - t[5]/12 + t[7]/12 + t[9]/12 - t[10]/12
    a[4] += t[1]/2 + t[4]/12 - t[6]/12 - t[8]/12 + t[11]/12
    a[0:2] += b[:]

```

Fig. 5. Algorithm to accumulate a new displacement into a log signature in the Lyndon basis with $d = 2$ and $m = 3$.

for the complexity of these functions, but a crude overestimate can be given as follows. The number of terms involved in the function is bounded by the number of possible products of up to m of the $O(d^m)$ scalar inputs, so the runtime is $O(d^{2m})$.

The basis (of the free Lie algebra on two symbols) used to express the BCH formula does not change the code we get, because the various equivalent bracketed expressions expand to the same expression when they have been multiplied out. We use the Lyndon basis because it has slightly fewer terms, as Casas and Murua [2009b] describe and partially explain. This choice is independent of the choice of basis (of the free Lie algebra on d symbols) in which the log signature is expressed. In general, we end up with fewer terms and a slightly faster calculation when the Lyndon basis is used for the log signature.

5 CALCULATING LOG SIGNATURES FROM SIGNATURES

A different method for calculating the log signature of a path is to calculate its signature first, and then convert to the log signature, as illustrated in Algorithm 3. The first step in doing the conversion is taking the logarithm itself in tensor space. This explicitly uses the formula in Equation (8), where n only needs to go as high as the required level, and the power is in the concatenation product. This results in the log signature as an element of tensor space (which means we have not used the fact yet that the log signature lives in a lower-dimensional subspace). (We have an option to return this *expanded logsignature*, by calling `logsig` with the “X” (*expanded*) method argument.) The exact order of evaluation of Equation (8) for best efficiency that we use is one that was suggested by Giles [2017]. We expand Equation (8) similarly to Horner’s method for polynomial evaluation as follows:

$$\log(1 + T) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} T^n}{n} = T - T \left(\frac{T}{2} - T \left(\frac{T}{3} - T \left(\frac{T}{4} - T \left(\dots \right) \right) \right) \right) \quad (10)$$

$\begin{matrix} 1 & 2 & 3 & 4 & 4 & 3 & 2 & 1 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \downarrow \downarrow \downarrow & & & \end{matrix}$

We wish to evaluate the logarithm of a signature, which is 1 in level 0, and therefore T will be 0 at level 0. Because of this, if S is an element of tensor space, then each level m' of TS can only be affected by levels $m' - 1$ and below of S . Therefore, we can assume there is 0 inside the parentheses labeled with m , and we only need evaluate the quantity inside each the pair of parentheses labeled m' up to level $m - m'$. The whole logarithm will take time $\Theta(d^m)$.

ALGORITHM 3: S method: Calculating the log signature via the signature

- 1: SIGNATURE populated with zeros
 - 2: **for** each segment SEGMENT in the path **do**
 - 3: SEGMENTSIGNATURE $\leftarrow$ the signature of SEGMENT using Equation (4)
 - 4: SIGNATURE $\leftarrow$ the Chen product of SIGNATURE and SEGMENTSIGNATURE using Equation (6)
 - 5: **end for**
 - 6: EXPANDEDLOGSIGNATURE $\leftarrow$ `log(SIGNATURE)` using Equation (10)
 - 7: LOGSIGNATURE $\leftarrow$ EXPANDEDLOGSIGNATURE projected to desired basis
 - 8: **return** LOGSIGNATURE
-

To express this Lie element in a specified basis, we need to project it. We calculate a projection explicitly. There are known explicit forms for projections, for example, the map given by the Dynkin-Specht-Wever lemma directly (r on page 20 of Reutenauer [1994]), which requires more operations. The `prepare` function calculates a projection upfront.

Given the bracketed expression of a basis element with m letters, we can easily find its expression in expanded space, by multiplying out the brackets. For example, $[[1, 3], 3]$ corresponds to $133 - 2313 + 331$. This gives us the full matrix M_m to transform each level of the log signature to its expanded version. Each column of M_m is labeled with a basis element, and each row is labeled with one of the d^m words of length d . To compress level m of a given expanded log signature x_m to its value c_m in terms of a basis, we just need to solve a least squares problem $M_m c_m = x_m$. This problem is a very overdetermined system that is known to have an exact answer, up to rounding considerations. M_m is tall and skinny.

The words occurring in the terms of the expansion of such a bracketed expression are anagrams of the foliage of the expression; i.e., they contain the same number of each letter as the expression itself. In the terminology of Reutenauer [1994], expanding brackets is a *finely homogenous*

Table 2. The Sizes of the Largest Anagram Sets for Level 10 of $d = 3$ in Decreasing Order of Number of Log Signature Elements (Many More Such Statistics Have Been Tabulated in Blümlein [2004])

Letter Frequencies	Number of Sets	Signature Elements in Each	Log Signature Elements in Each	Total Log Signature Elements
{4, 3, 3}	3	4,200	420	1,260
{4, 4, 2}	3	3,150	312	936
{5, 3, 2}	6	2,520	252	1,512
{5, 4, 1}	6	1,260	126	756
{6, 2, 2}	3	1,260	124	372

operation. In that same example, for instance, **133**, **313**, and **331** are anagrams of **133**. This leads to a lot of sparsity in the matrix M_m . Permuting the rows and columns to gather anagrams makes M_m be a block diagonal matrix with the blocks corresponding to equivalence classes of anagrams, which we call *anagram sets*. We can save time doing the transformation by solving a separate linear system for each anagram set of words of length m .

For the standard Hall basis, this is the whole procedure we follow. In prepare, we determine the mapping matrices between anagram sets of the log signature and its expansion, and then we calculate all their Moore-Penrose pseudoinverses, so that solving the systems is just a matrix multiplication. The number of words in an anagram set containing m letters where the frequency of the i th letter is n_i is given by a multinomial coefficient $\frac{m!}{n_1! \dots n_d!}$. The number of Lie basis elements in an anagram set is given by the second Witt formula of Satz 3 of Witt [1937] as

$$\ell_m(n_1, \dots, n_d) = \frac{1}{m} \sum_{\delta | n_i} \frac{\mu(\delta) \left(\frac{m}{\delta}\right)!}{\left(\frac{n_1}{\delta}\right)! \dots \left(\frac{n_d}{\delta}\right)!}, \quad (11)$$

where δ ranges over all common factors of the n_i and μ is the Möbius function. In the simple special case that the words have m distinct letters, there are $m!$ words and $(m - 1)!$ basis elements. In the case of the Lyndon basis for the Lie algebra, this formula checks out because the Lyndon words in such a set of $m!$ words are just all that begin with the lowest letter. Typically the largest anagram sets are the ones with about the same number of each letter. For them, Equation (11) is just $\frac{1}{m}$ times the number of words in the set because 1 is the only value of δ . For example, looking at level 10 for a three-dimensional path, the signature has 59,049 elements and the log signature 5,880, and there are 63 anagram sets. (The count is $63 = \binom{10+3-1}{10} - 3$ using the formula for unordered sampling with replacement and the fact that no basis element above level 1 has only one distinct letter in it.) The 12 most balanced anagram sets account for 3,708 elements of the log signature, or 63.1% of it, as enumerated in Table 2.

The big anagram sets account for most of the runtime when projecting to the log signature: multiplying a 420×4200 matrix by a 4,200-vector takes 80% more multiplications than multiplying a 312×3150 matrix by a 3,150-vector and so on. The number of basis elements in the largest anagram set is $O(\frac{1}{m} d^m)$, which is the maximum number of times each value in the expanded log signature needs to be read, and so the complexity of projection is at most this much times the size of the signature, so $O(\frac{1}{m} d^{2m})$.

5.1 Lyndon Case

If the Lyndon basis is required, then we have a more efficient implementation, which depends on a special property it has. On pages 89–91 of Reutenauer [1994], the notation P_a is introduced for the Lie polynomial corresponding to the Hall word a , i.e., the polynomial you get by multiplying

out the bracketed expression corresponding to the unique basis element whose foliage is a . This notation is used in the statement of the following.

THEOREM 5.1 (THEOREM 5.1 OF REUTENAUER [1994]). *The set of Lyndon words, ordered alphabetically, is a Hall set. The corresponding Hall basis has the following triangularity property: for each word $w = l_1 \dots l_n$ written as a decreasing product of Lyndon words, the polynomial $P_w = P_{l_1} \dots P_{l_n}$ is equal to w plus a $\mathbb{Z}$ -linear combination of greater words.*

The simplest case of the final statement, where w is itself a single Lyndon word, gives the following useful fact. When the bracketed expression corresponding to a Lyndon word is expanded and terms are ordered in alphabetical order, the first term will be the Lyndon word itself, with coefficient 1. (For an example, consider the Lyndon word **133**; its bracketed expression is $[[\mathbf{1}, \mathbf{3}], \mathbf{3}]$ and we saw earlier that this expands to $\mathbf{133} - 2\mathbf{313} + \mathbf{331}$.) This means that the tall skinny matrix M_m is lower triangular, as are its anagram blocks. If we take such a block and remove all the rows corresponding to words that are not Lyndon, we are left with the mapping from an anagram set in the compressed log signature to the same Lyndon word elements of the expanded signature. It is a square lower triangular matrix with ones on the diagonal. We can now solve the system directly in many fewer operations, with just addition and multiplication, just looking at the Lyndon word elements of the expanded signature. `prepare` determines the necessary indices and matrices, and `logsig` does the solving. This saves a factor m in the complexity.

For example, in level 4 on three dimensions, the following are the three basis elements that contain two **1**s, a **2**, and a **3**:

$$\begin{aligned} [\mathbf{1}, [\mathbf{1}, [\mathbf{2}, \mathbf{3}]]] &= \mathbf{1123} - \mathbf{1132} - 2\mathbf{1231} + 2\mathbf{1321} + \mathbf{2311} - \mathbf{3211} \\ [\mathbf{1}, [[\mathbf{1}, \mathbf{3}], \mathbf{2}]] &= \mathbf{1132} - \mathbf{1213} + \mathbf{1231} - \mathbf{1312} - \mathbf{1321} + \mathbf{2131} - \mathbf{2311} + \mathbf{3121} \\ [[\mathbf{1}, \mathbf{2}], [\mathbf{1}, \mathbf{3}]] &= \mathbf{1213} - \mathbf{1231} - \mathbf{1312} + \mathbf{1321} - \mathbf{2113} + \mathbf{2131} + \mathbf{3112} - \mathbf{3121} \end{aligned}$$

The matrix corresponding to these looks as follows:

$$\begin{array}{c} \begin{array}{c} [\mathbf{1}, [\mathbf{1}, [\mathbf{2}, \mathbf{3}]] \\ [\mathbf{1}, [[\mathbf{1}, \mathbf{3}], \mathbf{2}]] \\ [[\mathbf{1}, \mathbf{2}], [\mathbf{1}, \mathbf{3}]] \end{array} \\ \begin{array}{c} \mathbf{1123} \\ \mathbf{1132} \\ \mathbf{1213} \\ \mathbf{1231} \\ \mathbf{1312} \\ \mathbf{1321} \\ \mathbf{2113} \\ \mathbf{2131} \\ \mathbf{2311} \\ \mathbf{3112} \\ \mathbf{3121} \\ \mathbf{3211} \end{array} \left(\begin{array}{ccc} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \\ -2 & 1 & -1 \\ 0 & -1 & -1 \\ 2 & -1 & 1 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -1 \\ -1 & 0 & 1 \end{array} \right) \end{array}$$

and when we restrict to Lyndon words (which in general are not the first rows), we get a matrix that has $m = 4$ times fewer rows and is a lower triangular square matrix with ones on the diagonal.

$$\begin{array}{c} \text{1123} \\ \text{1132} \\ \text{1213} \end{array} \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}$$

$\begin{array}{c} \text{[1, [1, [2, 3]]} \\ \text{[1, [1, 3], 2]} \\ \text{[[1, 2], [1, 3]]} \end{array}$

If this matrix is called M' , we can solve the equation $M'c' = x'$ directly using

$$c'_1 = x'_1 \quad c'_2 = x'_2 - (-1 c'_1) \quad c'_3 = x'_3 - (0 c'_1 - 1 c'_2),$$

where the coefficients in parentheses come from the elements of M' below the diagonal.

6 IMPLEMENTATION

iisignature is a Python package that is built on numpy [van der Walt et al. 2011], a system that is ubiquitous for dealing with numerical data in Python. The Python ecosystem is very commonly used for deep learning. It is implemented as a C++ extension.

There is a single .cpp file that defines the whole interface with Python. The mathematical functionality resides in header files. This *unity build* structure reduces the time to build the whole library, which matters to users, at the cost of incremental build time.

6.1 Signatures

We store signatures during the calculation with each level in a contiguous block of memory, which means that accesses are efficient. We start with the signature of the first displacement and step-by-step calculate the concatenation product of it with the signature of each succeeding displacement. The concatenation product is done in place, but in simple cases this does not seem to make a difference in performance.

We also wrote an implementation of the signature calculation using a template metaprogramming style, where the dimension and level are template parameters, there is no heap memory allocation, and all loops are constant length. See *Signature.h* in the accompanying code. We compared the methods and learned that the performance is the same. Because we want to allow arbitrary calculations easily for the user, we do not choose that approach.

6.2 Preparing the Direct Calculation of Log Signatures

The internal representation of the calculation required to convert the log signature of a path into the log signature of that path with a line segment concatenated on the end, which is a procedure like those displayed in Figure 4 and Figure 5, is stored in an instance of the *FunctionData* struct in *makeCompiledFunction.hpp*. The calculation depends on the following concepts, which are illustrated in Table 3, whose code is in *bch.hpp*. An *Input* represents an indeterminate, and a *Coefficient* is a polynomial in *Inputs*. Elements of the basis of the free Lie algebra we are using are represented by instances of the *BasisElt* class. These are created once for a whole calculation, and they all live together in memory controlled by a *BasisPool*, which also remembers

Table 3. Summary of the Main Object Types for the Free Lie Algebra (FLA) Calculations

Type	Algebraic Structure	Definition	Instance Represents
double	$\mathbb{R}$ (roughly)	C++ built-in	constant floating-point real
Input	set	indeterminate	numeric input
Coefficient	semigroup ring	polynomial from <code>double[{Inputs}]</code>	formula in terms of the inputs
BasisElt	set	(fixed but complicated)	element of the given basis of the FLA
Polynomial	free vector space augmented with Lie bracket	function from BasisElt to Coefficient	element of FLA

Table 4. Various Signature Calculation Timings in Seconds, for 100 Random Paths of 100 Steps Each in the Given Combinations of Level and Dimension

(d, m)	(2,6)	(2,10)	(3,10)	(5,5)	(10,4)
Python native	10.56	78.66	2458.09	55.95	118.83
iisignature.sig	0.02	0.27	10.78	0.29	0.61
esig.tosig.stream2sig	1.98	55.51	3114.36	56.36	175.38

their order and those of their Lie products (Lie brackets) that happen to be BasisElts.² Elements of the free Lie algebra, Lie polynomials, are represented by the class Polynomial. In a similar way as Lyons et al. [2010], we store the data of a Polynomial separated by levels. This speeds up the multiplication of two of them very much, because it becomes trivial to avoid trying to multiply terms whose combined level will exceed the level we are truncating at. The procedure `makeFunctionDataForBCH` calculates the BCH product of an arbitrary polynomial (one with a separate indeterminate for each basis element, representing an arbitrary log signature) and an arbitrary level 1 polynomial (one that is just a separate indeterminate for each letter, representing the log signature of a single displacement) to produce a Polynomial, `poly`, from which it can extract exactly what the `FunctionData` needs to calculate.

The function `slowExplicitFunction` interprets the `FunctionData` object—it applies the algorithm (like that shown in Figure 5) to particular input. In the “0” (*object*) mode, the `logsig` calculation uses `slowExplicitFunction` with the prepared `FunctionData` object for each displacement in the path to calculate the full log signature.

6.3 On-the-Fly Machine Code Generation for the Direct Calculation

The interpreter `slowExplicitFunction` of the instructions in a `FunctionData` object is not very efficient. Code specifically compiled for the particular calculation required (choice of basis, value of d , value of m) is more efficient. Before this library, we wrote some code (described in Reizenstein [2015] and available in the `LogSignatureDemo` directory of the code accompanying this article) to generate C++ code, which can be compiled to give efficient versions of this function. We learned that while this method is very efficient, it is impractical for many realistic values of d and m because the function can easily get so large that compilers take an unreasonably long time to compile it. Attempting to split them up only helps a small amount. Manual machine code generation

²Note that there are three relevant orderings on `BasisElt`: (1) there is the Hall basis order itself, which is available through the function `lexicographicLess`, and which is needed to construct the basis elements initially; (2) there is the order that is sorted by level and *then* in Hall basis order within each level, which is used for outputting the log signature; and (3) there is the pointer ordering of `BasisElt` objects, which is used for indexing some sorted data structures.

Table 5. Various Log Signature Calculation Timings in Seconds, for 1,000 Random Two-Dimensional Paths of 100 Steps Each for the Given Levels

	Level:	2	3	4	5	6	7	8	9	10	11
C	Lyndon	0.01	0.01	0.02	0.03	0.05	0.14	0.52	1.64	4.90	29.21
C	standard	0.01	0.01	0.02	0.03	0.05	0.15	0.55	1.74	5.31	32.34
O	Lyndon	0.02	0.03	0.05	0.14	0.34	1.21	3.26	10.18	30.46	95.98
O	standard	0.02	0.03	0.05	0.15	0.36	1.25	3.38	10.80	32.89	107.35
S	Lyndon	0.02	0.03	0.06	0.12	0.20	0.37	0.70	1.41	2.87	6.01
S	standard	0.02	0.04	0.06	0.11	0.21	0.37	0.70	1.43	2.92	6.15
esig	standard	0.73	1.61	3.71	8.52	19.26	43.66	98.47	224.83	506.25	1,132.24

avoids this delay. `iisignature` therefore provides the “C” method under which it compiles the `FunctionData` itself on the fly to machine code internally in a buffer in memory during prepare, and all `logsig` needs to do is to run the compiled code for each displacement. This is implemented for x86 and x86-64 architectures, for Windows, Linux, and Mac.

The logic for the compilation is in `makeCompiledFunction.hpp`. The `Mem` object represents a buffer for storing machine code, which is allocated in such a way that execution is enabled. The `FunctionRunner` object is constructed from a `FunctionData` and allocates a `Mem` and compiles the function into it, providing a `go()` function to run the compiled code. The actual compilation is done by the `Maker` class. The comments in that class explain what it is doing in terms of x86 and x86-64 machine code instructions. In the x86 case, we rely on SSE2 instructions for floating-point arithmetic, which enables the logic to be roughly the same as in the 64-bit case.

6.4 Projection from Expanded Log Signature to a Basis

The code for this all lives in `logsig.hpp`, which is the master file for the log signature calculations. The `makeMappingMatrix` function calculates the full matrix (in sparse form) to project from tensor space to the desired basis. The identification of anagram sets is performed in the function `analyseMappingMatrixLevel`. The function `makeSparseLogSigMatrices` identifies all the data needed to do the projection from this information. When, as often happens, a basis element’s anagram set is a singleton, we can just read off its value from the expanded log signature without solving a system. In the standard Hall basis case, we need to calculate the Moore-Penrose pseudoinverses of the identified matrices, and we do this using `numpy` at the interface level in `LeastSquares::inplacePinvMatrix` in `pythonsigs.cpp`. All the data to do this is stored in the object that `prepare` generates and is available to be simply used by `logsig`.

7 INDICATIVE TIMINGS

Using 64-bit Python 3.5 on Ubuntu 16.04LTS with an AMD FX-8320, timings were taken for calculating 100 signatures of randomly generated paths with 100 timesteps in various different ways. They are shown in Table 4. We compare here a native Python signature implementation using `numpy`, the calculation with `iisignature` version 0.20, and the calculation in the package `esig.tosig` of CoRoPa [Lyons et al. 2010], version 0.6.5. Both `iisignature` and `esig` use 64-bit floating point internally, but `iisignature` is taking and returning 32-bit floating-point values in this example, whereas `esig` uses 64-bit throughout. It should be noted that `esig` was not specifically written to make this type of calculation fast, but for other types of flexibility (e.g., the sparse signature case). The dramatic difference shows the advantage of having code written specifically for the dense case. The size of the signature and log signature, both up to level m and of level m itself, are all $\Theta(d^m)$. The runtimes of the signature and the log signature calculations grow

Fig. 6. Various log signature calculation timings in seconds, for 1,000 random two-dimensional paths of 100 steps each for various levels. For *iisignature*, only the Lyndon basis is shown.

Table 6. Various Log Signature Calculation Timings in Seconds, for 1,000 Random Three-Dimensional Paths of 10 Steps Each for the Given Levels

Level:	2	3	4	5	6	7	8	9	10
C Lyndon	0.01	0.01	0.01	0.02	0.08	0.45	4.35	40.80	221.41
C standard	0.01	0.01	0.01	0.02	0.09	0.50	5.36	47.11	255.75
O Lyndon	0.01	0.01	0.03	0.11	0.51	2.76	14.67	84.04	466.12
O standard	0.01	0.01	0.02	0.12	0.53	2.98	16.50	101.06	605.05
S Lyndon	0.01	0.02	0.03	0.05	0.15	0.46	1.52	5.38	19.25
S standard	0.01	0.02	0.03	0.05	0.15	0.51	1.92	8.36	41.06
esig standard	0.13	0.42	1.54	5.59	22.69	86.24	338.08	1,310.86	5,451.14

exponentially in m and polynomially in d , with the polynomial being d^m for sig and unknown to us but at most d^{2m} for the log signature. Runtimes are linear in the length of the path.

Calculating the log signature using the compiled method is quicker than calculating the signature for small depths, but for larger depths the signature becomes significantly faster. As the depth increases, the projection method therefore becomes the best method to obtain the log signature. For two dimensions, performance is shown in Table 5 and plotted in Figure 6, and for three dimensions performance is shown in Table 6 and plotted in Figure 7.

We expect the time taken to increase polynomially in dimension, so we plot the time taken level 5 for various methods as d increases on a log-log plot in Figure 8 and show the timings in Table 7. Ultimately the calculation time would be expected to be quintic in d . For d in the high single figures, we observe much higher growth in the runtime of the compiled code.

The preparation in *iisignature* is slow for the C method when d or m is large. Timings for a single call are illustrated for various levels with $d = 3$ in Figure 9 and for various dimensions with $m = 5$ in Figure 10. There is an advantage in using the Lyndon basis.

While there is always more that can be done to speed up software performance, *iisignature* provides a significant speedup over other options easily available to those using Python for machine learning and doing lots of signature calculations.

Fig. 7. Various log signature calculation timings in seconds, for 1,000 random three-dimensional paths of 10 steps each for various levels. For iisignature, only the Lyndon basis is shown.

Fig. 8. Various level-5 log signature calculation timings in seconds, for 1,000 random paths of 10 steps each of various dimensions. For iisignature, only the Lyndon basis is shown. The graphs look to have roughly reached a straight line for $d \geq 6$. The least-squares line of each is shown with its gradient, which indicates the approximate degree of a polynomial relationship.

8 INDICATIVE MEMORY USAGE

We used the Massif tool [Nethercote et al. 2006] from Valgrind to profile memory usage calculating a single log signature, in particular collecting the peak memory usage. The peak memory usage is interesting because running out of memory is usually what makes certain calculations impossible. There is a background memory cost independent of the algorithm, which includes space to store the BCH coefficients. In order just to measure the algorithm, we ran these calculations from within

Table 7. Various Level-5 Log Signature Calculation Timings in Seconds, for 1,000 Random Paths of 10 Steps Each of Various Dimensions

Dimension:		2	3	4	5	6	7	8	9	10
C	Lyndon	0.01	0.02	0.09	0.29	0.85	3.17	7.24	22.33	37.83
C	standard	0.01	0.02	0.10	0.30	1.03	3.35	7.58	21.64	38.38
O	Lyndon	0.02	0.11	0.51	1.63	4.18	9.34	20.76	39.39	63.67
O	standard	0.02	0.12	0.52	1.69	4.38	9.84	19.72	38.44	64.92
S	Lyndon	0.03	0.05	0.14	0.37	0.84	1.73	3.24	5.66	9.30
S	standard	0.03	0.05	0.15	0.41	0.96	2.02	4.56	6.81	11.60
esig	standard	0.93	5.59	22.75	73.17	190.47	437.35	931.58	1,814.74

Fig. 9. Timings for a single preparation of the three-dimensional log signature calculation for various levels. Smaller marks are used for the standard Hall basis, regular marks for the Lyndon basis. Values for O and C are very similar, so the former are omitted. Very small values are also omitted.

C++ without using Python, and we subtract the memory usage calculating the same signature at level 1 from the observed memory usage.

The values are very consistent across repeated runs. Values for three-dimensional paths are shown in Figure 11. We observe that memory usage is another reason the projection method becomes a better choice for higher levels.

9 OTHER FUNCTIONALITY

We have described the core functionality of *iisignature*, the calculation of signatures and log signatures. There are also simple methods for calculating the concatenation product of a given signature and the signature of a given segment and for transforming a given signature according to the path being scaled by an enlargement in each of the coordinate directions. Counterparts for backpropagating derivatives through the signature and log signature calculations and through these are provided, allowing any of them to occur in the middle of a neural network. The implementations of these are fairly straightforward, and some more details are given in Section 4.9 of Reizenstein [2019]. The library also implements functions for calculating the linear combinations

Fig. 10. Timings for a single preparation of the level-5 log signature calculation for various dimensions. Smaller marks are used for the standard Hall basis, regular marks for the Lyndon basis. Values for 0 and C are very similar, so the former are omitted. Very small values are also omitted.

Fig. 11. Memory usages in bytes for C and S calculations of a three-dimensional path of 10 steps for various levels. Only the Lyndon basis is shown.

of signature elements of 2D paths that are invariant under rotation, which were derived in Diehl [2013] and have been applied to handwriting recognition. The package includes tests at the Python level, including that the various ways to calculate the signature and log signature are consistent with each other.

10 CONCLUSIONS AND FUTURE WORK

We have presented and analyzed efficient methods for computing signatures and log signatures. We have implemented what we considered to be the most promising algorithms.

We have focused on small-dimensional data in our design, because this encompasses many applications where the signature has been used in machine learning, for example, handwriting recognition and EEG data, and calculating the log signature in cases where $d \gg m$ has not been a priority. Some data is naturally a high-dimensional sequence, possibly discrete (making movements of a fixed length in a single dimension at a time), like some representations of music and text, so this is a possible use case. In these cases there is lots of repetition in the calculations. There are potential changes to the code, which would make prepare use significantly less memory (and therefore be usable for larger d) in this regime, at the cost of a little more calculation time in `logsig`. How to evaluate multivariate polynomials to optimize speed and accuracy is an open topic of research. `iisignature` uses simple methods and there is plenty to improve upon.

We have shown that machine code generation directly from the BCH formula is useful in this domain. The calculation is data-access heavy and the order of operations has a big effect, because of memory latency and data dependencies, and there is scope to improve it. We find that adding extra operations to the code without changing the data access does not slow it down, suggesting that it is the effect of data-cache misses that is the main bottleneck. There are subsets of the calculation that are repeated on different parts of the data. Operating in parallel with vector instructions might speed things up. There are avenues for working on these possibilities, for example, using the LLVM system, which brings the advantages of a modern compiler to the code generation.

In machine learning applications, signature calculations are likely to be required in large batches, and the interface of `iisignature` supports calculating batches of signatures and log signatures in a single call. It would be easy to parallelize the calculation over these batches. For the common case where signature features are being used in training a machine learning model, typically these batches of signatures are being calculated on one of a set of identical background data loader threads, meaning that parallelized signature calculations would not be so useful.

ACKNOWLEDGMENTS

We thank Terry Lyons and Joscha Diehl for many helpful suggestions.

REFERENCES

- E. Alpaydin and Fevzi Alimoglu. 1998. Pen-Based Recognition of Handwritten Digits Data Set. Retrieved from <https://archive.ics.uci.edu/ml/datasets/Pen-Based+Recognition+of+Handwritten+Digits>.
- Johannes Blümlein. 2004. Algebraic relations between harmonic sums and associated quantities. *Comput. Phys. Comm.* 159, 1 (2004), 19–54. <https://arxiv.org/abs/hep-ph/0311046>.
- Fernando Casas and Ander Murua. 2009a. The BCH formula and the symmetric BCH formula up to terms of degree 20. Retrieved from <http://www.ehu.es/ccwmurua/bch.html>.
- Fernando Casas and Ander Murua. 2009b. An efficient algorithm for computing the Baker-Campbell-Hausdorff series and some of its applications. *J. Math. Phys.* 50, 3 (March 2009), 033513. arXiv:<http://arxiv.org/abs/0810.2656>.
- Kuo-Tsai Chen. 1958. Integration of paths – a faithful representation of paths by noncommutative formal power series. *Trans. Amer. Math. Soc.* 89, 2 (1958), 395–407. <https://www.jstor.org/stable/1993193>.
- Ilya Chevyrev and Andrey Kormilitzin. 2016. A Primer on the Signature Method in Machine Learning. (2016). <http://arxiv.org/abs/1603.03788>.
- Joscha Diehl. 2013. Rotation invariants of two dimensional curves based on iterated integrals. (2013). <http://arxiv.org/abs/1305.6883>.
- Mike Giles. 2017. Private communication. (2017), via Terry Lyons.
- Benjamin Graham. 2013. Sparse arrays of signatures for online character recognition. (2013). <http://arxiv.org/abs/1308.0371>.
- Marshall Hall. 1950. A basis for free Lie rings and higher commutators in free groups. (1950). Retrieved from <http://www.ams.org/journals/proc/1950-001-05/S0002-9939-1950-0038336-7/home.html>.
- Terry Lyons. 1998. Differential equations driven by rough signals. *Rev. Mat. Iberoam.* 14 (1998), 215–310.
- Terry Lyons, Stephen Buckley, Djalil Chafai, Greg Gyurkó, and Arend Janssen. 2010. CoRoPa Computational Rough Paths (software library). Retrieved from <http://coropa.sourceforge.net/>.

- Nicholas Nethercote, Robert Walsh, and Jeremy Fitzhardinge. 2006. Building workload characterization tools with Valgrind. *IEEE International Symposium on Workload Characterization* (2006). Retrieved from <http://valgrind.org/docs/iiswc2006.pdf>.
- Jeremy Reizenstein. 2015. Calculation of Iterated-Integral Signatures and Log Signatures. (2015). arxiv.org/abs/1712.02757.
- Jeremy Reizenstein. 2019. *Iterated-Integral Signatures in Machine Learning*. Ph.D. Dissertation. Warwick University. Retrieved from <http://wrap.warwick.ac.uk/131162/>.
- Christophe Reutenauer. 1994. *Free Lie Algebras*. Oxford.
- Anatoli Illarionovich Shirshov. 1953. Subalgebras of free Lie algebras. *Mat. Sbornik* N.S. 33 75, 2 (1953), 441–452.
- Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. 2011. The numpy array: A structure for efficient numerical computation. *Comput. Sci. Eng.* 13, 2 (2011), 22–30. Retrieved from <http://aip.scitation.org/doi/abs/10.1109/MCSE.2011.37>.
- Ernst Witt. 1937. Treue darstellung liescher ringe. *J. Reine Angew. Math.* 177 (1937), 152–160.
- Weixin Yang, Terry Lyons, Hao Ni, Cordelia Schmid, Lianwen Jin, and Jiawei Chang. 2017. Leveraging the Path Signature for Skeleton-based Human Action Recognition. (2017). <http://arxiv.org/abs/1707.03993>.

Received June 2018; revised June 2019; accepted November 2019