A FURTHER DETAILS OF ALGORITHMIC IMPROVEMENTS

A.1 FUSED MULTIPLY-EXPONENTIATE

The conventional way to compute a signature is to iterate through the computation described by equation (3): for each new increment, take its exponential, and $\boxtimes$ it on to what has already been computed; repeat.

Our proposed alternate way is to fuse the exponential and $\boxtimes$ into a single operation, and then iteratively perform this fused operation.

We now count the number of multiplications required to compute

$$\left(\prod_{k=1}^{N} \left(\mathbb{R}^d\right)^{\otimes k}\right) \times \mathbb{R}^d \to \prod_{k=1}^{N} \left(\mathbb{R}^d\right)^{\otimes k},$$
$$A, z \mapsto A \boxtimes \exp(z)$$

for each approach.

We will establish that the fused operation uses fewer multiplications for all possible $d \geq 1$ and $N \geq 1$. We will then demonstrate that it is in fact of a lower asymptotic complexity.

A.1.1 THE CONVENTIONAL WAY

The exponential is defined as

$$\exp\colon \mathbb{R}^d \to \prod_{k=1}^{N} \left(\mathbb{R}^d\right)^{\otimes k},$$
$$\exp\colon x \mapsto \left(x, \frac{x^{\otimes 2}}{2!}, \frac{x^{\otimes 3}}{3!}, \ldots, \frac{x^{\otimes N}}{N!}\right),$$

see Bonnier et al. (2019, Proposition 15).

Note that every tensor in the exponential is symmetric, and so in principle requires less work to compute than its number of elements would suggest. For the purposes of this analysis, to give the benefit of the doubt to a competing method, we shall assume that this is done (although taking advantage of this in practice is actually quite hard (Reizenstein & Graham, 2018, Section 2)). This takes

$$\sum_{k=2}^{N} \left(d + \binom{d+k-1}{k}\right)$$

scalar multiplications, using the formula for unordered sampling with replacement (Reizenstein & Graham, 2018, Section 2), under the assumption that each division by a scalar costs the same as a multiplication (which can be accomplished by precomputing their reciprocals and then multiplying by them).

Next, we need to count the number of multiplications to perform a single $\boxtimes$.

Let

$$A, B \in \prod_{k=1}^{N} \left(\mathbb{R}^d\right)^{\otimes k}.$$

Let $A = (A_1, \ldots, A_N)$, with

$$A_i = (A_i^{j_1,\ldots,j_i})_{1 \leq j_1,\ldots,j_i \leq d},$$

and every $A_i^{j_1,\ldots,j_i} \in \mathbb{R}$. Additionally let $A_0 = 1$. Similarly for B. Then $\boxtimes$ is defined by

$$\boxtimes\colon \left(\prod_{k=1}^{N} \left(\mathbb{R}^d\right)^{\otimes k}\right) \times \left(\prod_{k=1}^{N} \left(\mathbb{R}^d\right)^{\otimes k}\right) \to \prod_{k=1}^{N} \left(\mathbb{R}^d\right)^{\otimes k},$$
$$\boxtimes\colon A, B \mapsto \left(\sum_{i=0}^{k} A_i \otimes B_{k-i}\right)_{1 \leq k \leq N}, \tag{8}$$

where each

$$A_i \otimes B_{k-i} = \left(A_i^{j_1,\ldots,j_i} B_{k-i}^{\hat{j}_1,\ldots,\hat{j}_{k-i}}\right)_{1\le j_1,\ldots,j_i,\hat{j}_1,\ldots,\hat{j}_{k-i}\le d}$$

is the usual tensor product, the result is thought of as a tensor in $(\mathbb{R}^d)^{\otimes k}$, and the summation is taken in this space. See Bonnier et al. (2019, Definition A.13).

To the authors' knowledge there has been no formal analysis of a lower bound on the computational complexity of $\boxtimes$, and there is no better way to compute it than naiïvely following this definition.

This, then, requires

$$\begin{aligned}\sum_{k=1}^{N}\sum_{i=1}^{k-1}\sum_{j_1,\ldots,j_i=1}^{d}\sum_{\hat{j}_1,\ldots,\hat{j}_{k-i}=1}^{d} 1 &= \sum_{k=1}^{N}\sum_{i=1}^{k-1} d^k \\ &= \sum_{k=1}^{N}(k-1)d^k\end{aligned}$$

scalar multiplications.

Thus the overall cost of the conventional way involves

$$\mathcal{C}(d,N) = \sum_{k=2}^{N}\left(d + \binom{d+k-1}{k}\right) + \sum_{k=1}^{N}(k-1)d^k \tag{9}$$

scalar multiplications.

A.1.2 The fused operation

Let $A \in \prod_{k=1}^{N}\left(\mathbb{R}^d\right)^{\otimes k}$ and $z \in \mathbb{R}^d$. Then

$$A \boxtimes \exp(z) = \left(\sum_{i=0}^{k} A_i \otimes \frac{z^{\otimes(k-i)}}{(k-i)!}\right)_{1\le k\le N},$$

where the k-th term may be computed by a scheme in the style of Horner's method:

$$\begin{aligned}&\sum_{i=0}^{k} A_i \otimes \frac{z^{\otimes(k-i)}}{(k-i)!} = \\ &\quad\left(\left(\cdots\left(\left(\frac{z}{k} + A_1\right)\otimes\frac{z}{k-1} + A_2\right)\otimes\frac{z}{k-2} + \cdots\right)\otimes\frac{z}{2} + A_{k-1}\right)\otimes z + A_k.\end{aligned} \tag{10}$$

As before, we assume that the reciprocals $\frac{1}{2},\ldots,\frac{1}{N}$ have been precomputed, so that each division costs the same as a multiplication.

Then we begin by computing $z/2,\ldots,z/N$, which takes $d(N-1)$ multiplications.

Computing the k-th term as in equation (10) then involves $d^2 + d^3 + \cdots + d^k$ multiplications. This is because, working from innermost bracket to outermost, the first $\otimes$ produces a $d \times d$ matrix as the outer product of two size d vectors, and may thus be computed with d^2 multiplications; the second $\otimes$ produces a $d \times d \times d$ tensor from a $d \times d$ matrix and a size d vector, and may thus be computed with d^3 multiplications; and so on.

Thus the overall cost of a fused multiply-exponentiate is

$$\mathcal{F}(d,N) = d(N-1) + \sum_{k=1}^{N}\sum_{i=2}^{k} d^i \tag{11}$$

scalar multiplications.

A.1.3 COMPARISON

We begin by establishing the uniform bound $\mathcal{F}(d, N) \leq \mathcal{C}(d, N)$ for all $d \geq 1$ and $N \geq 1$.

First suppose $d = 1$. Then

$$\begin{aligned}\mathcal{F}(1, N) &= (N-1) + \sum_{k=1}^{N}(k-1) \\ &\leq 2(N-1) + \sum_{k=1}^{N}(k-1) \\ &= \mathcal{C}(1, N).\end{aligned}$$

Now suppose $N = 1$. Then

$$\mathcal{F}(d, 1) = 0 = \mathcal{C}(d, 1).$$

Now suppose $N = 2$. Then

$$\begin{aligned}\mathcal{F}(d, 2) &= d + d^2 \\ &\leq d + \binom{d+1}{2} + d^2 \\ &= \mathcal{C}(d, 2)\end{aligned}$$

Now suppose $d \geq 2$ and $N \geq 3$. Then

$$\begin{aligned}\mathcal{F}(d, N) &= d(N-1) + \sum_{k=1}^{N}\sum_{i=2}^{k} d^i \\ &= \frac{d^{N+2} - d^3 - (N-1)d^2 + (N-1)d}{(d-1)^2}.\end{aligned} \tag{12}$$

And

$$\begin{aligned}\mathcal{C}(d, N) &= \sum_{k=2}^{N}\left(d + \binom{d+k-1}{k}\right) + \sum_{k=1}^{N}(k-1)d^k \\ &\geq \sum_{k=1}^{N}(k-1)d^k \\ &= \frac{(N-1)d^{N+2} - Nd^{N+1} + d^2}{(d-1)^2}.\end{aligned} \tag{13}$$

Thus we see that it suffices to show that

$$d^{N+2} - d^3 - (N-1)d^2 + (N-1)d \leq (N-1)d^{N+2} - Nd^{N+1} + d^2,$$

for $d \geq 2$ and $N \geq 3$. That is,

$$0 \leq d^{N+1}(d(N-2) - N) + d(d^2 + N(d^2-1) + 1). \tag{14}$$

At this point $d = 2$, $N = 3$ must be handled as a special case, and may be verified by direct evaluation of equation (14). So now assume $d \geq 2$, $N \geq 3$, and that $d = 2$, $N = 3$ does not occur jointly. Then we see that equation (14) is implied by

$$0 \leq d(N-2) - N \quad \text{and} \quad 0 \leq d^2 + N(d^2-1) + 1.$$

The second condition is trivially true. The first condition rearranges to $N/(N-2) \leq d$, which is now true for $d \geq 2$, $N \geq 3$ with $d = 2$, $N = 3$ not jointly true.

This establishes the uniform bound $\mathcal{F}(d, N) \leq \mathcal{C}(d, N)$.

Checking the asymptotic complexity is much more straightforward. Consulting equations (12) and (13) shows that $\mathcal{F}(d, n) = \mathcal{O}(d^N)$ whilst $\mathcal{C}(d, N) = \Omega(Nd^N)$. And in fact as $\binom{d+k-1}{k} \leq d^k$ then equation (9) demonstrates that $\mathcal{C}(d, N) = \mathcal{O}(Nd^N)$.

A.2 LOGSIGNATURE BASES

We move on to describing our new more efficient basis for the logsignature.

A.2.1 WORDS, LYNDON WORDS, AND LYNDON BRACKETS

Let $\mathcal{A} = \{a_1, \dots, a_d\}$ be a set of d letters. Let $\mathcal{A}^{+N}$ be the set of all words in these letters, of length between 1 and N inclusive. For example $a_1a_4 \in \mathcal{A}^{+N}$ is a word of length two.

Impose the order $a_1 < a_2 < \cdots < a_d$ on $\mathcal{A}$, and extend it to the lexicographic order on words in $\mathcal{A}^{+N}$ of the same length as each other, so that for example $a_1a_2 < a_1a_3 < a_2a_1$. Then a *Lyndon word* (Lalonde & Ram, 1995) is a word which comes earlier in lexicographic order than any of its rotations, where rotation corresponds to moving some number of letters from the start of the word to the end of the word. For example $a_2a_2a_3a_4$ is a Lyndon word, as it is lexicographically earlier than $a_2a_3a_4a_2$, $a_3a_4a_2a_2$ and $a_4a_2a_2a_3$. Denote by $\mathcal{L}\left(\mathcal{A}^{+N}\right)$ the set of all Lyndon words of length between 1 and N.

Given any Lyndon word $w_1 \cdots w_n$ with $n \geq 2$ and $w_i \in \mathcal{A}$, we may consider its *longest Lyndon suffix*; that is, the smallest j for which $w_j \cdots w_n$ is a Lyndon word. (It is guaranteed to exist as w_n alone is a Lyndon word.) It is a fact (Lalonde & Ram, 1995) that $w_1 \cdots w_{j-1}$ is then also a Lyndon word. Given a Lyndon word w, we denote by w^b its longest Lyndon suffix, and by w^a the corresponding prefix.

Considering spans with respect to $\mathbb{R}$, let

$$[\cdot, \cdot] \colon \operatorname{span}(\mathcal{A}^{+N}) \times \operatorname{span}(\mathcal{A}^{+N}) \to \operatorname{span}(\mathcal{A}^{+N})$$

be the commutator given by

$$[w, z] = wz - zw,$$

where wz denotes concatenation of words, distributed over the addition, as w and z belong to a span and thus may be linear combinations of words. For example $w = 2a_1a_2 + a_1$ and $z = a_1 + a_3$ gives $wz = 2a_1a_2a_1 + 2a_1a_2a_3 + a_1a_1 + a_1a_3$.

Then define

$$\phi \colon \mathcal{L}\left(\mathcal{A}^{+N}\right) \to \operatorname{span}(\mathcal{A}^{+N})$$

by $\phi(w) = w$ if w is a word of only a single letter, and by

$$\phi(w) = [\phi(w^a), \phi(w^b)]$$

otherwise. For example,

$$\begin{aligned}\phi(a_1a_2a_2) &= [[a_1, a_2], a_2] \\ &= [a_1a_2 - a_2a_1, a_2] \\ &= a_1a_2a_2 - 2a_2a_1a_2 + a_2a_2a_1.\end{aligned}$$

Now extend ϕ by linearity from $\mathcal{L}\left(\mathcal{A}^{+N}\right)$ to $\operatorname{span}(\mathcal{L}\left(\mathcal{A}^{+N}\right))$, so that

$$\phi \colon \operatorname{span}(\mathcal{L}\left(\mathcal{A}^{+N}\right)) \to \operatorname{span}(\mathcal{A}^{+N})$$

is a linear map between finite dimensional real vector spaces, from a lower dimensional space to a higher dimensional space.

Next, let

$$\psi \colon \mathcal{A}^{+N} \to \operatorname{span}(\mathcal{L}\left(\mathcal{A}^{+N}\right))$$

be such that $\psi(w) = w$ if $w \in \mathcal{L}\left(\mathcal{A}^{+N}\right)$, and $\psi(w) = 0$ otherwise. Extend ψ by linearity to $\operatorname{span}(\mathcal{A}^{+N})$, so that

$$\psi \colon \operatorname{span}(\mathcal{A}^{+N}) \to \operatorname{span}(\mathcal{L}\left(\mathcal{A}^{+N}\right))$$

is a linear map between finite dimensional real vector spaces, from a higher dimensional space to a lower dimensional space.

A.2.2 A BASIS FOR SIGNATURES

Recall that the signature transform maps between spaces as follows.

$$\mathrm{Sig}^N : \mathcal{S}\left(\mathbb{R}^d\right) \to \prod_{k=1}^{N} \left(\mathbb{R}^d\right)^{\otimes k}.$$

Let $\{e_i \mid 1 \leq i \leq d\}$ be the usual basis for $\mathbb{R}^d$. Then

$$\{e_{i_1} \otimes \cdots \otimes e_{i_k} \mid i \leq i_1, \ldots i_k \leq d\}$$

is a basis for $(\mathbb{R}^d)^{\otimes k}$. An arbitrary element of $\prod_{k=1}^{N} \left(\mathbb{R}^d\right)^{\otimes k}$ may be written as

$$\left(\sum_{i_1,\ldots i_k=1}^{d} \alpha_{i_1,\ldots,i_k} e_{i_1} \otimes \cdots \otimes e_{i_k} \right)_{1 \leq k \leq N} \tag{15}$$

for some $\alpha_{i_1,\ldots,i_k}$.

Then $\mathcal{A}^{+N}$ may be used to represent a basis for $\prod_{k=1}^{N} \left(\mathbb{R}^d\right)^{\otimes k}$. Identify $e_{i_1} \otimes \cdots \otimes e_{i_k}$ with $a_{i_1} \cdots a_{i_k}$. Extend linearly, so as to identify expression (15) with the formal sum of words

$$\sum_{k=1}^{N} \sum_{i_1,\ldots i_k=1}^{d} \alpha_{i_1,\ldots,i_k} a_{i_1} \cdots a_{i_k}.$$

With this identification,

$$\mathrm{span}(\mathcal{A}^{+N}) \cong \prod_{k=1}^{N} \left(\mathbb{R}^d\right)^{\otimes k} \tag{16}$$

A.2.3 BASES FOR LOGSIGNATURES

Suppose we have some $\mathbf{x} \in \mathcal{S}\left(\mathbb{R}^d\right)$. Using the identification in equation (16), then we may attempt to seek some $x \in \mathrm{span}(\mathcal{L}\left(\mathcal{A}^{+N}\right))$ such that

$$\phi(x) = \log\left(\mathrm{Sig}^N(\mathbf{x})\right). \tag{17}$$

This is an overdetermined linear system. As a matrix ϕ is tall and thin. However it turns out that $\mathrm{image}\,(\log) = \mathrm{image}\,(\phi)$ and moreover there exists a unique solution (Reizenstein & Graham, 2018). (That it is an overdetermined system is typically the point of the logsignature transform over the signature transform, as it then represents the same information in less space.)

If $x = \sum_{\ell \in \mathcal{L}(\mathcal{A}^{+N})} \alpha_\ell \ell$, with $\alpha_\ell \in \mathbb{R}$, then by linearity

$$\sum_{\ell \in \mathcal{L}(\mathcal{A}^{+N})} \alpha_\ell \phi(\ell) = \log\left(\mathrm{Sig}^N(\mathbf{x})\right),$$

so that $\phi(\mathcal{L}\left(\mathcal{A}^{+N}\right))$ is a basis, called the Lyndon basis, of $\mathrm{image}\,(\log)$. When calculating the logsignature transform in a computer, then the collection of α_ℓ are a sensible choice for representing the result, and indeed, this is what is done by `iisignature`. See Reizenstein & Graham (2018) for details of this procedure.

However, it turns out that this is unnecessarily expensive. In deep learning, it is typical to apply a learnt linear transformation after a nonlinearity - in which case we largely do not care in what basis we represent the logsignature, and it turns out that we can find a more efficient one.

The Lyndon basis exhibits a particular triangularity property (Reutenauer, 1993, Theorem 5.1), (Reizenstein, 2019, Theorem 32), meaning that for all $\ell \in \mathcal{L}\left(\mathcal{A}^{+N}\right)$, then $\phi(\ell)$ has coefficient zero for any Lyndon word lexicographically earlier than ℓ. This property has already been exploited by `iisignature` to solve (17) efficiently, but we can do better: it means that

$$\psi \circ \phi : \mathrm{span}\left(\mathcal{L}\left(\mathcal{A}^{+N}\right)\right) \to \mathrm{span}\left(\mathcal{L}\left(\mathcal{A}^{+N}\right)\right)$$

is a triangular linear map, and so in particular it is invertible, and defines a change of basis; it is this alternate basis that we shall use instead. Instead of seeking x as in equation (17), we may now instead seek $z \in \operatorname{span}\left(\mathcal{L}\left(\mathcal{A}^{+N}\right)\right)$ such that

$$(\phi \circ (\psi \circ \phi)^{-1})(z) = \log\left(\operatorname{Sig}^N(\mathbf{x})\right).$$

But now by simply applying ψ to both sides:

$$z = \psi\left(\log\left(\operatorname{Sig}^N(\mathbf{x})\right)\right).$$

This is now incredibly easy to compute. Once $\log\left(\operatorname{Sig}^N(\mathbf{x})\right)$ has been computed, and interpreted as in equation (16), then the operation of ψ is simply to extract the coefficients of all the Lyndon words, and we are done.

B LIBTORCH VS CUDA

LibTorch is the C++ equivalent to the PyTorch library. GPU support in Signatory was provided by using the operations provided by LibTorch.

It was a deliberate choice not to write custom CUDA kernels. The reason for this is as follows. We have to make a choice between distributing source code and distributing precompiled binaries. If we distribute source code, then we rely on users being able to compile CUDA, which is far from a guarantee.

Meanwhile, distributing precompiled binaries is unfortunately not feasible on Linux. C/C++ extensions for Python are typically compiled for Linux using the 'manylinux' specification, and indeed PyPI will only host binaries claiming to be compiled according to this specification. Unfortunately, based on our inspection of its build scripts, PyTorch appears not to conform to this specification. It instead compiles against a later version of Centos than is supported by manylinux, and then subsequently modifies things so as to *seem* compatible with the manylinux specification.

Unpicking precisely how PyTorch does this so that we might duplicate the necessary functionality (as we must necessarily remain compatible with PyTorch as well) was judged a finickity task full of hard-to-test edge cases, that is an implementation detail of PyTorch that should not be relied upon, and that may not remain stable across future versions.

C FURTHER BENCHMARKS

C.1 CODE FOR REPRODUCABILITY

The benchmarks may be reproduced with the following code on a Linux system. First we install the necessary packages.

```
pip install numpy==1.18.0 matplotlib==3.0.3 torch==1.5.0
pip install iisignature==0.24 esig==0.6.31 signatory==1.2.1.1.5.0
git clone https://github.com/[redacted].git
cd signatory
```

Note that numpy must be installed before `iisignature`, and PyTorch must be installed before Signatory. The unusually long version number for Signatory is necessary to specify both the version of Signatory, and the version of PyTorch that it is for. The `git clone` is necessary as the benchmarking code is not distributed via `pip`.

For this anonymised version: the complete source code, including benchmarking code, is instead attached as supplementary material.

Now run

```
python command.py benchmark -help
```

for further details on how to run any particular benchmark. For example,

```
python command.py benchmark -m time -f sigf -t channels -o graph
```

will reproduce Figure 1a.

C.2 MEMORY BENCHMARKS

Our benchmark scripts offer some limited ability to benchmark memory consumption, via the `-m memory` flag to the benchmark scripts.

The usual approach to such benchmarking, using `valgrind`'s `massif`, necessarily includes measuring the set-up code. As this includes loading both the Python interpreter and PyTorch, measuring the memory usage of our code becomes tricky.

As such we use an alternate method, in which the memory usage is sampled at intervals, using the Python package `memory_profiler`, which may be installed via `pip install memory_profiler`. This in turn has the limitation that it may miss a peak in memory usage; for small calculations it may miss the entire calculation. Furthermore, the values reported are inconsistent with those reported in Reizenstein & Graham (2018).

Nonetheless, when compared against `iisignature` using `memory_profiler`, on larger computations where peaks are less likely to go unobserved, then Signatory typically uses at an order of magnitude less memory. However due to the limitations above, we have chosen not report quantitative memory benchmarks here.

C.3 SIGNATURE TRANSFORM BENCHMARKS

The precise values of the points of figures 1 and 2 are shown in Tables 1–4.

For convenience, the ratio between the speed of Signatory and the speed of `iisignature` is also shown.

C.4 LOGSIGNATURE TRANSFORM BENCHMARKS

See Figure 3 for the graphs of the benchmarks for the logsignature transform.

The computer and runtime environment used was as described in Section 5.

We observe similar behaviour to the benchmarks for the signature transform. `iisignature` is slightly faster for some very small computations, but that as problem size increases, Signatory swiftly overtakes `iisignature`, and is orders of magnitude faster for larger computations.

The precise values of the points on these graphs are shown in Tables 5–8. Times are given in seconds. Also shown is the ratio between the speed of Signatory and the speed of `iisignature`. A dash indicates that `esig` does not support that operation.

Table 1: Signature forward, varying channels. Times are given in seconds. A dash indicates that `esig` does not support that operation.

Channels	2	3	4	5	6	7
`esig`	0.531	9.34	-	-	-	-
`iisignature`	0.00775	0.0632	0.375	1.97	7.19	20.9
Signatory CPU (no parallel)	0.00327	0.0198	0.101	0.402	1.45	3.8
Signatory CPU (parallel)	0.00286	0.00504	0.00975	0.0577	0.21	1.22
Signatory GPU	0.0129	0.0135	0.0182	0.0222	0.0599	0.158
Ratio CPU (no parallel)	2.37	3.19	3.71	4.89	4.95	5.49
Ratio CPU (parallel)	2.71	12.5	38.5	34.1	34.2	17.0
Ratio GPU	0.602	4.68	20.6	88.7	120	132

Table 2: Signature backward, varying channels. Times are given in seconds. A dash indicates that `esig` does not support that operation.

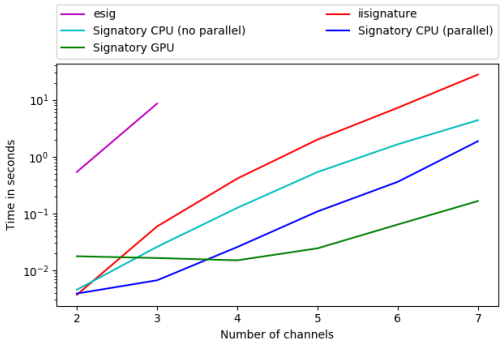
Channels	2	3	4	5	6	7
`esig`	-	-	-	-	-	-
`iisignature`	0.026	0.248	1.59	7.78	27.6	128
Signatory CPU (no parallel)	0.0222	0.106	0.428	1.54	4.97	13.7
Signatory CPU (parallel)	0.00922	0.0623	0.265	1.01	3.49	9.0
Signatory GPU	0.0472	0.0413	0.0534	0.119	0.314	0.772
Ratio CPU (no parallel)	1.17	2.34	3.7	5.07	5.56	9.38
Ratio CPU (parallel)	2.82	3.97	6.0	7.69	7.92	14.2
Ratio GPU	0.551	6.0	29.7	65.2	87.9	166

Table 3: Signature forward, varying depths. Times are given in seconds. A dash indicates that `esig` does not support that operation.

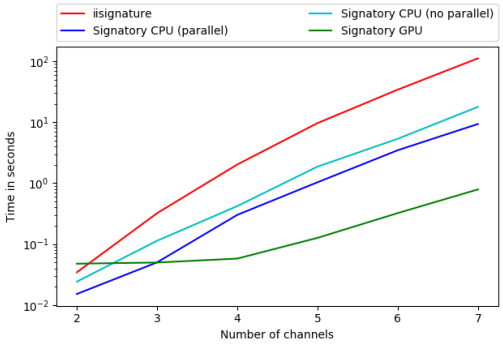
Depth	2	3	4	5	6	7	8	9
`esig`	0.019	0.0954	0.527	2.73	15.5	-	-	-
`iisignature`	0.000468	0.00145	0.00485	0.0199	0.0859	0.376	1.83	8.16
Signatory CPU (no parallel)	0.000708	0.00129	0.0022	0.00765	0.027	0.104	0.402	1.68
Signatory CPU (parallel)	0.000722	0.00242	0.00279	0.00321	0.00546	0.0161	0.0408	0.381
Signatory GPU	0.00172	0.00326	0.00484	0.00735	0.0104	0.0132	0.0232	0.0773
Ratio CPU (no parallel)	0.661	1.12	2.2	2.61	3.18	3.6	4.55	4.86
Ratio CPU (parallel)	0.649	0.597	1.74	6.21	15.7	23.3	44.8	21.4
Ratio GPU	0.273	0.443	1.0	2.71	8.24	28.3	79.0	106

Table 4: Signature backward, varying depths. Times are given in seconds. A dash indicates that `esig` does not support that operation.

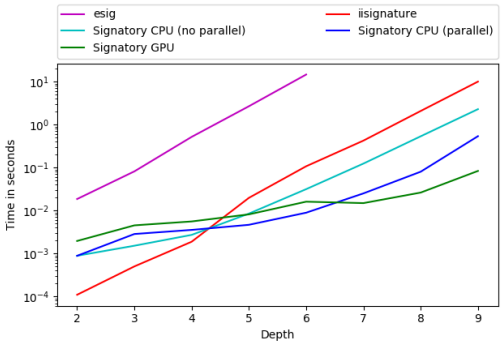
Depth	2	3	4	5	6	7	8	9
`esig`	-	-	-	-	-	-	-	-
`iisignature`	0.00149	0.00438	0.0179	0.0954	0.366	1.59	7.72	34.7
Signatory CPU (no parallel)	0.00322	0.00518	0.0123	0.0347	0.109	0.437	1.8	6.31
Signatory CPU (parallel)	0.00354	0.00409	0.0089	0.0152	0.0563	0.175	0.839	4.06
Signatory GPU	0.00525	0.00916	0.015	0.0216	0.0324	0.05	0.144	0.495
Ratio CPU (no parallel)	0.464	0.845	1.45	2.75	3.37	3.63	4.28	5.49
Ratio CPU (parallel)	0.422	1.07	2.02	6.28	6.51	9.07	9.21	8.54
Ratio GPU	0.284	0.478	1.19	4.42	11.3	31.7	53.8	70.1



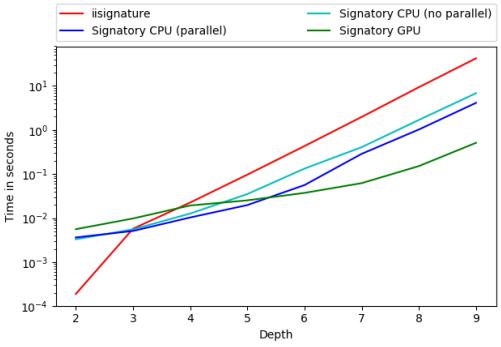
(a) Logsignature forward, varying channels



(b) Logsignature backward, varying channels



(c) Logsignature forward, varying depths



(d) Logsignature backward, varying depths

Figure 3: Time taken on benchmark computations to compute the specified operation. In all cases the input was a batch of 32 sequences of data, each of length 128. For varying channels, the depth was fixed at 7. For varying depths, the channels was fixed at 4. Every test case was repeated 50 times and the fastest time taken. Note that `esig` is only shown for certain operations as it is incapable of computing large operations or of computing backward operations. Note the logarithmic scale.

Table 5: Logsignature forward, varying channels. Times are given in seconds. A dash indicates that `esig` does not support that operation.

Channels	2	3	4	5	6	7
`esig`	0.539	8.61	-	-	-	-
`iisignature`	0.0037	0.0591	0.412	2.0	7.26	27.9
Signatory CPU (no parallel)	0.00454	0.0258	0.125	0.536	1.65	4.4
Signatory CPU (parallel)	0.00388	0.00665	0.0256	0.108	0.36	1.87
Signatory GPU	0.0176	0.0164	0.0149	0.0243	0.0638	0.165
Ratio CPU (no parallel)	0.815	2.29	3.28	3.73	4.4	6.36
Ratio CPU (parallel)	0.952	8.89	16.1	18.4	20.1	14.9
Ratio GPU	0.21	3.6	27.5	82.2	114	169

Table 6: Logsignature backward, varying channels. Times are given in seconds. A dash indicates that `esig` does not support that operation.

Channels	2	3	4	5	6	7
`esig`	-	-	-	-	-	-
`iisignature`	0.0346	0.322	2.02	9.66	34.1	111
Signatory CPU (no parallel)	0.0243	0.114	0.421	1.87	5.34	17.9
Signatory CPU (parallel)	0.0152	0.0503	0.302	1.03	3.47	9.34
Signatory GPU	0.0478	0.05	0.058	0.127	0.323	0.79
Ratio CPU (no parallel)	1.42	2.83	4.81	5.17	6.4	6.24
Ratio CPU (parallel)	2.27	6.4	6.69	9.34	9.85	11.9
Ratio GPU	0.723	6.44	34.9	76.2	106	141

Table 7: Logsignature forward, varying depths. Times are given in seconds. A dash indicates that `esig` does not support that operation.

Depth	2	3	4	5	6	7	8	9
`esig`	0.0185	0.0815	0.517	2.67	14.6	-	-	-
`iisignature`	0.00011	0.000502	0.00188	0.0197	0.107	0.423	2.07	9.98
Signatory CPU (no parallel)	0.000888	0.00152	0.00272	0.00849	0.0315	0.124	0.534	2.28
Signatory CPU (parallel)	0.000886	0.00285	0.00355	0.00466	0.00891	0.0251	0.0801	0.536
Signatory GPU	0.00196	0.00453	0.00558	0.00815	0.0161	0.0149	0.0262	0.0834
Ratio CPU (no parallel)	0.124	0.33	0.693	2.33	3.41	3.42	3.88	4.37
Ratio CPU (parallel)	0.124	0.176	0.531	4.23	12.0	16.9	25.9	18.6
Ratio GPU	0.0561	0.111	0.337	2.42	6.67	28.4	79.0	120

Table 8: Logsignature backward, varying depths. Times are given in seconds. A dash indicates that `esig` does not support that operation.

Depth	2	3	4	5	6	7	8	9
`esig`	-	-	-	-	-	-	-	-
`iisignature`	0.000189	0.00572	0.0225	0.0968	0.432	1.98	9.36	42.3
Signatory CPU (no parallel)	0.0033	0.00555	0.0127	0.0351	0.133	0.408	1.69	6.84
Signatory CPU (parallel)	0.00363	0.0051	0.0103	0.0197	0.0562	0.287	1.02	4.13
Signatory GPU	0.00559	0.00979	0.0193	0.0253	0.0373	0.0621	0.151	0.511
Ratio CPU (no parallel)	0.0573	1.03	1.78	2.76	3.25	4.86	5.54	6.19
Ratio CPU (parallel)	0.0521	1.12	2.18	4.91	7.69	6.89	9.13	10.2
Ratio GPU	0.0338	0.584	1.17	3.83	11.6	31.9	61.8	82.8