# Some algorithmic considerations inside iisignature

Jeremy Reizenstein, Centre for Complexity Science[*]
University of Warwick

February 2017

**Abstract**

An explanation of some algorithmic ideas used in the implementation of `iisignature`. Refer to its documentation for an explanation of the functionality.

## Contents

[*]Supported by the Engineering and Physical Sciences Research Council and The Alan Turing Institute under the EPSRC grant EP/N510129/1

## 1 Backpropagation thoughts

Having a backpropagation function along with a library function can often be more efficient than defining the whole operation in the language of an autodifferentiation facility like those in `tensorflow`, `torch` or `theano`. There are lots of steps in signature calculations which would each have to be analysed separately, and this process would probably happen every time a program was run. Each calculation we provide together with its backprop counterpart can be thought of by the deep learning library as a sealed unit. Often we can save memory this way - we don't have to store every single intermediate value, and at runtime we don't need to work out which intermediate values need to be stored.

The most common style is for a backprop function to receive the original functions inputs. We have functions which have optional positional arguments. Therefore it is most consistent to take the gradient as the first argument of backprop functions followed by the original inputs. In practice, a backprop function could additionally accept original function's output and the autodifferentiation systems would be able to provide it, but we don't have a need to make use of this. Ideally state which is internal to the calculation should not need to be remembered.

In some cases, as described in this document, by thinking carefully about the backwards operation we can find a more sensible way to calculate it than the obvious way. This means that our own function can be more efficient than that which an autodifferentiation system is likely to have come up with. In practice, it seems that our backwards operations pleasingly take about the same time as the corresponding forwards operation, and it would be unlikely that the backwards operation would be much quicker in any case.

When we have a calculation structure which looks like Figure 1 to backpropagate derivatives through, we need the value of the output of each calculation of function $f$ as we pass derivatives down the tree, in order to send the derivative to the input which is a sibling of that $f$. There are basically two ways to make this available. The standard way is that we evaluate the calculation forwards storing all the intermediate $f$ values. This may require extra memory and copying of data to store them (unless we repeatedly do parts of the calculation). This is classic reverse-mode automatic differentiation in action, and is what typical deep learning packages do by default to implement backpropagation. Some of the storage can be omitted in return for redoing parts of the calculation. In some cases, however, we might be lucky, in that both $f$ and (right multiplication by something fixed) are invertible, and so we can find the output of each $f$ down the tree starting from the output of the calculation. By doing this in tandem with the backpropagation of derivatives, we don't need any storage for each level of the calculation graph. This idea works for `sig` but not for `logsigfromsig`, fundamentally because among group-like elements of tensor space, multiplication by a given element is invertible, but among Lie elements, multiplication by a given element is not.

$$\begin{tikzcd}
\text{input} \arrow[r] & \times \arrow[r] & f \arrow[r] & \times \arrow[r] & f \arrow[r] & \times \arrow[r] & f \arrow[r] & \times \arrow[r] & \text{output} \\
\text{input} \arrow[ur] & & \text{input} \arrow[ur] & & \text{input} \arrow[ur] & & \text{input} \arrow[ur] & &
\end{tikzcd}$$

Figure 1: A calculation dependency, where $f$ is some function and $\times$ is something like multiplication.

## 2 sig

There is nothing so new here, but it helps to write this function's calculation out to fix notation for when we come to the derivative.

`sig(A,m)` calculates the signature of a piecewise linear path by combining the signatures of each straight-line-piece (which are the exponents of the displacement) using Chen's formula.

If the $d$-dimensional input was $\left((p_{jk})_{j=0}^{l}\right)_{k=1}^{d}$ I could express the calculation like this. The displacement vectors are given by

$$(q_j)_k = p_{jk} - p_{(j-1),k} \tag{1}$$

and the signature of the $j$ segment is

$$r_j(i_1 \dots i_n) = \frac{1}{n!} \prod_{h=1}^{n} (q_j)_{i_h} \tag{2}$$

and using Chen's relation we find the signature of the path up to point $j$

$$\begin{aligned} s_1(i_1 \dots i_n) &= r_1(i_1 \dots i_n) \\ s_j(i_1 \dots i_n) &= s_{j-1}(i_1 \dots i_n) + \left[ \sum_{p=1}^{n-1} s_{j-1}(i_1 \dots i_p) r_j(i_{p+1} \dots i_n) \right] + r_j(i_1 \dots i_n). \end{aligned} \tag{3}$$

If you'll tolerate an invalid ellipsis as an empty word and remember that the signature of an empty word is always 1, this can be written simply

$$s_j(i_1 \dots i_n) = \sum_{p=0}^{n} s_{j-1}(i_1 \dots i_p) r_j(i_{p+1} \dots i_n).$$

Each signature object is calculated all at once (i.e. its value for all words is calculated in one go, for all words up to length $m$), and stored in a vector (one for each level/length of word) of vectors (of words in alphabetical order). The output of the function is all the values of $s_l$. For example, for a path given as six points, the calculation dependency looks like this, taking the displacement vectors and signature objects as single entities and neglecting the actual calculation of the displacements. The calculation order needn't be like this, the tree could be constructed differently, but the number of operations would be unchanged. A nested calculation order like this requires only storing one $s_j$ at a time.

$$\begin{tikzcd}
 & & & & s_5 \arrow[dl, dash] \arrow[dr, dash] & \\
 & & & s_4 \arrow[dl, dash] \arrow[dr, dash] & & r_5 \arrow[d, dash] \\
 & & s_3 \arrow[dl, dash] \arrow[dr, dash] & & r_4 \arrow[d, dash] & q_5 \\
 & s_2 \arrow[dl, dash] \arrow[dr, dash] & & r_3 \arrow[d, dash] & q_4 & \\
 s_1 = r_1 \arrow[d, dash] & & r_2 \arrow[d, dash] & q_3 & & \\
 q_1 & & q_2 & & &
\end{tikzcd}$$

Figure 2: Sig calculation dependency

I use the $\otimes$ symbol for the kronecker product of two column vectors as a vector – so it takes a vector of length $x$ and a vector of length $y$ to a vector of length $xy$. If $u$ has length $x$ and $v$ has length $y$ then $uv$ has elements $(u \otimes v)_{(i-1)y+j} = u_i v_j$. If $a$ is a signature, then I write $a^m$ for the vector of its elements at level $m$. To calculate $r_j$ I do the following

**Algorithm 1** Segment signature

```
1: $r_j^1 \leftarrow q_j$
2: for $n \leftarrow 2, m$ do
3:     $r_j^n \leftarrow \frac{1}{n}(r_j^{n-1} \otimes q_j)$
4: end for
```

The algorithm for the full signature is like this.

**Algorithm 2** sig

```
1: Calculate $r_1$
2: for $n \leftarrow 1, m$ do
3:     $s^n \leftarrow r_1^n$
4: end for
5: for $j \leftarrow 2, l$ do
6:     Calculate $r_j$
7:     for $n \leftarrow m, 1$ do                      ▷ make $s^n$ go from being $s^n_{j-1}$ to $s^n_j$
8:         for $n' \leftarrow (n-1), 1$ do
9:             $s^n \leftarrow s^n + s^{n'} \otimes r_j^{n-n'}$
10:        end for
11:        $s^n \leftarrow s^n + r_j^n$
12:    end for
13: end for
14: $s^n$ is now $s_l^n$ for each $n$
```

It feels wasteful to store each $r_j^n$ as a whole vector of $d^n$ numbers, because they are really repeats. The signature of a straight line takes the same value on all permutations of a word, so there are only $\binom{d+n-1}{n}$ distinct values. But trying to exploit this led me to slower, more complicated code.

## 3 sigbackprop

Naively, given a calculation tree to define an output in terms of simple calculations, we can backpropagate derivatives through the tree using the chain rule. If we know the value of every node in the tree (having stored them while calculating the output) then we can pass a derivative though each leg of a multiplication node $M$ by multiplying it by the stored values in the other inputs to $M$. We send a derivative both ways through an addition unchanged. There is naturally a time/memory tradeoff in such a calculation, because each value could just be recalculated from inputs when it is needed, instead of having been remembered. We can in fact do much better than storing all the intermediate values, or even just all the $s_j$.

In general, if $X$ is the signature of a path $\gamma$, then the signature $X'$ of the reversed path $\gamma^{-1}$ is a permutation with some elements negated.

$$X'(a_1 \dots a_n) = (-1)^n X(a_n \dots a_1) \tag{4}$$

If $\gamma$ is a straight line, then the component of the signature is the same on all permutations of a word, and so we have the simpler

$$X'(a_1 \dots a_n) = (-1)^n X(a_1 \dots a_n) \tag{5}$$

Because all the $r_j$ are easy to calculate as the difference of inputs, we can easily use this to calculate $s_{j-1}$ from $s_j$. This we can perform at the same time as the backpropagation of derivatives with respect to $s_j$. Let the vector containing level $n$ of $r_j$ be $r_j^n$, with the derivatives with respect to its elements being $R_j^n$. Similarly $s_j^n$ and $S_j^n$ for $s_j$.

It is convenient to define the corresponding backpropagation operations to $\otimes$. If $u$ has length $x$, $v$ has length $y$, and $z$ has length $xy$, then $z \underset{\leftarrow}{\otimes} v$ has length $x$ with elements $(z \underset{\leftarrow}{\otimes} v)_i = \sum_j z_{(i-1)y+j} v_j$ and $u \underset{\rightarrow}{\otimes} z$ has length $y$ with elements $(u \underset{\rightarrow}{\otimes} z)_j = \sum_i u_i z_{(i-1)y+j}$.

The algorithm, based in the function `sigBackwards`, proceeds as follows.

**Algorithm 3** sigBackwards

```
1: for $n \leftarrow 1, m$ do
2:     $s^n \leftarrow s^n_l$, calculated using sig
3:     $S^n \leftarrow S^n_l$, an input
4: end for
5: for $j \leftarrow l, 1$ do
       ($s^n$ is now $s^n_j$ and $S^n$ is now $S^n_j$ for each $n$.)
6:     Calculate $r_j$.
7:     Use (5) to make $s^n$ be $s^n_{j-1}$ for each $n$.
           for $n \leftarrow m, 1$ do
               for $n' \leftarrow (n-1), 1$ do
                   $s^n \leftarrow s^n + (-1)^{n-n'} s^{n'} \otimes r_j^{n-n'}$
               end for
               $s^n \leftarrow s^n + (-1)^n r^n_j$
           end for
8:     Calculate $R_j$.
           for $n \leftarrow 1, m$ do
               $R^n_j \leftarrow S^n$
           end for
           for $n \leftarrow 1, m$ do
               for $n' \leftarrow (n-1), 1$ do
                   $R_j^{n-n'} \leftarrow R_j^{n-n'} + s^{n'} \underset{\rightarrow}{\otimes} S^n$
               end for
           end for
9:     Backpropagate $R^n_j$ through (2).
10:    Make $S^n$ be $S^n_{j-1}$, doable in place:
           for $n' \leftarrow 1, (m-1)$ do
               for $n \leftarrow (n'+1, m)$ do
                   $S^{n'} \leftarrow S^{n'} + S^n \underset{\leftarrow}{\otimes} r^{n-n'}$
               end for
           end for
11: end for
```

Some of the same logic is used for `sigjoinbackprop`.

## 4 sigscalebackprop

Here we present a general sensible idea to use when differentiating products of terms sampled with replacement from a list: *you keep the simple code you would have written if replacement was not allowed.*

Consider `sigscalebackprop` called in $d$ dimensions up to level $m$, in particular its action for a single level $l \leq m$. Each level's calculation produces derivatives with respect to its part of the original signature, and contributes to derivatives with respect to the scales. The data are $a$, the $l$th level of the original signature, $b$, the scales, and $E$, the supplied derivatives with respect to the $l$th level of the output. Denote the output of the expression by $e$ and the to-be-calculated derivatives with respect to $a$ and $b$ by $A$ and $B$ respectively. I denote subscripting of the inputs and outputs with square brackets, and unlike earlier where I indexed a level of a signature as a vector, here I am indexing it as a tensor.

The expression to be differentiated is as follows.

$$e[i_1,\dots,i_l] = \Big(\prod_{j=1}^{l} b[i_j]\Big) a[i_1,\dots,i_l] \qquad \text{for} \quad (i_1,\dots,i_l) \in \{1,\dots,d\}^l \tag{6}$$

Using standard rules for differentiation, the derivative calculations are as follows.

$$A[i_1,\dots,i_l] = \Big(\prod_{j=1}^{l} b[i_j]\Big) E[i_1,\dots,i_l] \qquad \text{for} \quad (i_1,\dots,i_l) \in \{1,\dots,d\}^l \tag{7}$$

$$B[k] = \sum_{\substack{(i_1,\dots,i_l)\in \\ \{1,\dots,d\}^l}} B[k; i_1,\dots,i_l] \qquad \text{for} \quad k \in \{1,\dots,d\}, \tag{8}$$

where the single contribution of a product is

$$B[k; i_1,\dots,i_l] = \Big(\prod_{\substack{j=1 \\ i_j \neq k}}^{l} b[i_j]\Big) a[i_1,\dots,i_l] E[i_1,\dots,i_l]\; C^k_{i_1\dots i_l} b[k]^{C^k_{i_1\dots i_l}-1} \tag{9}$$

and $C^k_{i_1\dots i_l}$ is the number of $j$ for which $i_j = k$.

Evaluating $e$ according to (6) will take time $d^l l$ but evaluating $B$ naively according to this procedure takes at least $d^{l+1}d$, because evaluating the $d$ counts $C^{\cdot}_{i_1\dots i_l}$ requires $d$ time for every $(i_1 \dots i_l)$, even though most are zero. A couple of rearrangements:

$$B[k; i_1,\dots,i_l] = \Big(\prod_{j=1}^{l} b[i_j]\Big) a[i_1,\dots,i_l] E[i_1,\dots,i_l]\; C^k_{i_1\dots i_l} b[k]^{-1} \tag{10}$$

$$B[k; i_1,\dots,i_l] = \sum_{h=1}^{l} 1_{\{i_h = k\}} \Big(\prod_{j=1}^{l} b[i_j]\Big) a[i_1,\dots,i_l] E[i_1,\dots,i_l]\; b[i_h]^{-1} \tag{11}$$

Organising the terms differently by making $k$ range among $(i_1,\dots,i_l)$ instead of $\{1,\dots,d\}$ means we split the $C^{\cdot}_{\cdot}$ occurrences of each multiplication. This leads to the following algorithm, which only takes time $d^l l$.

**Algorithm 4** single level of sigscalebackprop

```
1: for $(i_1,\dots,i_l) \in \{1,\dots,d\}^l$ do
2:     $prod \leftarrow \prod_{j=1}^{l} b[i_j]$
3:     $A[i_1,\dots,i_l] \leftarrow prod\; E[i_1,\dots,i_l]$
4:     for $h \in \{1,\dots,l\}$ do
5:         $B[i_h] \leftarrow B[i_h] + prod\; a[i_1,\dots,i_l] E[i_1,\dots,i_l] / b[i_h]$
6:     end for
7: end for
```

## 5 Two Dimensional Rotational Invariants

### 5.1 Raw

The paper [2] identifies the linear combinations of signature elements of a 2d path which are invariant under rotations of the path. The library allows a basis of these combinations for elements up to a given level to be calculated efficiently by the `rotinv2d` function, which first calculates the signature of the given path, and then extracts the combinations requested.

A basis for the rotational invariants is given by Theorem 2 of [2]. It has the property that each element is a combination of elements of level $2k$ of the signature for some $k$. This is true of the basis we return, and we provide it in increasing order of $k$. Specifically, in that paper's notation, if $x_1$ and $x_2$ are abstract noncommuting variables representing the dimensions, and $i_1 \cdots i_{2k}$ is a sequence of $k$ ones and $k$ twos, then the real and imaginary parts of the expression $c_{i_1 \cdots i_{2k}}$ where

$$\begin{aligned} c_{i_1 \cdots i_{2k}} &:= z_{i_1} \cdot z_{i_2} \cdot \ldots \cdot z_{i_{2k}} \\ z_1 &:= x_1 + i x_2 \\ z_2 &:= x_1 - i x_2 \end{aligned}$$

give rotational invariants at level $2k$, and such invariants span the space of invariants.

A few remarks. Replacing each 1 with a 2 and each 2 with a 1 in the sequence $i_1 \cdots i_{2k}$ takes $c_{i_1 \cdots i_{2k}}$ to its complex conjugate, so it is enough to consider sequences where $i_1 = 1$. With this done, I call the invariants we get *raw invariants*. Since the $c_{i_1 \cdots i_{2k}}$ for **all** $i_1 \cdots i_{2k} \in \{1, 2\}^{2k}$ form a *complex* basis for $\mathbb{C}^{2k}$ (Lemma 7 of [2]), the raw invariants are linearly independent (in $\mathbb{R}^{2k}$). They are thus a basis for the invariants at level $2k$.

Consider the elements of level $2k$ of the signature as a zero-based array of length $2^{2k}$. Then the binary expansion of the index of an element indicates the word which the signature element represents. For example the signature element for the word **2122** will be element $1011_2 = 11_{10}$ of level 4.

The terms in the raw invariants which come from the imaginary part will be each of the length-$2k$ strings which have an odd number of $x_1$s and an odd number of $x_2$s in, each with a factor of 1 or $-1$. Such invariants will consist of signature elements whose indices have an odd number of ones in their binary expansion, i.e. are *odious numbers* in the colourful taxonomy of [1]. I call them *odious invariants.*

The terms in the raw invariants which come from the real part will be each of the length-$2k$ strings which have an even number of $x_1$s and an even number of $x_2$s in, each with a factor of 1 or $-1$. Such invariants will consist of signature elements whose indices have an even number of ones in their binary expansion, i.e. are *evil numbers.* I call them *evil invariants.*

There are $\binom{2k}{k}$ raw invariants at level $k$, of which half are odious and half are evil. In the `"a"` (*all*) mode, it is the raw invariants of a path which are returned.

Each raw invariant is a sum of $2^{m-1}$ elements in level $m$ of the signature, with some negated. This may mean that the appropriate scaling of these data as input to a machine learning algorithm may differ by this factor from the appropriate scaling of the signature.

### 5.2 Reduced

As described in [2], due to the shuffle-product property of signatures, the values of some rotational invariants are known given the values of others at lower levels. Just as the log signature is useful as a minimal set of features from the signature, it may be useful to have a minimal representation of the rotational invariants, that is, a minimal representation of the signature information assigned to an equivalence class under rotation. We can achieve this at each level by finding the known invariants as shuffle products of lower raw invariants, and quotienting the span of the raw invariants by their span.

If $\mathcal{A}$ is the set of raw invariant vectors at level $2k$, and $\mathcal{B}$ is the set of known invariants at level $2k$, we can find a basis for the quotient as follows. First find a basis $\mathcal{B}'$ for the known invariants using the singular value decomposition. Then we project each element of $a$ of $\mathcal{A}$ away from each element $b$ of $\mathcal{B}'$ by replacing $a$ by $a - b \cdot a$. Finally, we get a basis for the projected $a$s using singular value decomposition again.

The shuffle product of two raw invariants is odious if exactly one of them is odious, and evil otherwise. Thus, by keeping track of odious and evil invariants everywhere, we can apply the procedure in the previous paragraph separately for the two cases. This means that although at each level the procedure must happen twice, the vectors concerned can be compressed to half as long, $\mathcal{A}$ is half the size, and $\mathcal{B}$ is about half the size in each case, which is a major time saving. This calculation is performed upfront when the `"s"` (*SVD*) mode is requested, so that the `rotinv2d` function can return the reduced set only.

Singular value decomposition may not be the quickest way to find a basis for the span of a set of vectors. QR decomposition with pivoting may be quicker, and it is availably in `scipy`[3]. If the user has `scipy` available, they can specify the method as `"q"` and get the same calculation done that way.

Unlike the raw invariants, the reduced invariants as returned by `iisignature` are unit vectors in (the dual of) each level of the signature, so it is plausible that the same scaling used for a calculation with the signature could be used with the reduced invariants.

## References

[1] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning ways for your mathematical plays (vol 1).* 1982 (cit. on p. 7).

[2] Joscha Diehl. "Rotation Invariants of Two Dimensional Curves Based on Iterated Integrals". 2013. URL: `http://arxiv.org/abs/1305.6883` (cit. on p. 7).

[3] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python.* 2001–. URL: `http://www.scipy.org/` (cit. on p. 8).