iisignature (version 0.23)

Jeremy Reizenstein, Centre for Complexity Science*
Ben Graham, Department of Statistics and Centre for Complexity Science
University of Warwick

August 2018

Abstract

The `iisignature` Python package is designed to provide an easy-to-use reasonably efficient implementation of iterated-integral signatures and log-signatures of piecewise linear paths. Some motivation as to the usefulness of these things in machine learning may be found in [3].

1 Installation into Python

First ensure you have `numpy` installed and available, version 1.7 (from 2013) or later. If you are not on Windows, for which prebuilt binaries are available from PyPI for Pythons 3.5, 3.6 and 3.7, you will need to be able to build `C++` python extensions. Then `pip install iisignature` will install it from `PyPI`. This is working in Python 3.5, 3.6 and 3.7 on Windows (where you might need to be in an Administrator command prompt), and Pythons 3.4 (and later) and 2.7 on Linux.

(You can also run `pip install --user iisignature` to install for your user only. On Windows, this doesn't need administrator rights, but you can't install for both 64 bit and 32 bit builds of the same version of Python in this way as doing one breaks the other.)

For Python 2.7 and older on Windows, the build is not working perfectly and you might want to get in touch if you want to use this combination.[1]

On a mac, you will need to have installed Xcode and the Xcode command line tools first.

*Supported by the Engineering and Physical Sciences Research Council

[1] I find it basically works building with `python setup.py build -c mingw32`, where the compiler is (MinGW from `https://nuwen.net/mingw.html` for 64 bit Python and standard MinGW for 32 bit Python). However, there are problems with running the tests, at least when you have no debugger attached.

2 Quick example

To generate a random 3 dimensional path made up of 20 points and get its signature and log signature up to level 4 do this:

```
import iisignature
import numpy as np

path = np.random.uniform(size=(20,3))
signature = iisignature.sig(path,4)
s = iisignature.prepare(3,4)
logsignature = iisignature.logsig(path,s)
```

3 Usage

Many of the functions require a path as input. A path in d dimensions which is specified as a series of n points should be given as a `numpy` array of either `float32` or (preferably) `float64` with shape `(n,d)`, or anything (like a list) which can be converted to one.

Batching: Many of the functions which accept arrays can do the same operation multiple times in a single call by stacking the inputs, i.e. adding extra initial dimensions. This is supported on any of the functions where this document shows an ellipsis (`...`) in a shape. If there are multiple array inputs to a function, their extra dimensions must match each other (broadcasting is not done). The result will have all the initial dimensions in it. Having any of these extra dimensions is always optional. A few of the functions are assumed to be most useful in the case where batching is in use.

`version()`

Return the version number of `iisignature`.

3.1 Signatures

For the purposes of `iisignature`, a signature up to level m is the concatenation of levels 1 to m as a single one dimensional array. The constant 1 which is always level 0 of a signature is never included.

`siglength(d, m)`

The length of the signature up to level m of a d-dimensional path. This has the value

$$\sum_{i=1}^{m} d^i = \frac{d(d^m - 1)}{d - 1}.$$

`sig(path, m, format=0)`

The signature of the d-dimensional path `path` up to level m is returned. The output is a `numpy` array of shape `(...,siglength(d,m))`. (If `format` is supplied as `1`, then the output for a single path is given as a list of `numpy` arrays, one for each level, for the moment.) (If `format` is supplied as `2`, then we return not just the signature of the whole path, but the signature of all the partial paths from the start to each point of the path. If `path` has shape `(...,n,d)` then the result has shape `(...,n-1,siglength(d,m))`.)

`sigjacobian(path, m)`

This function provides the Jacobian matrix of the `sig` function with respect to `path`. If `path` has shape `(n,d)` then an array of shape `(n,d,siglength(d,m))` is returned.

$$\texttt{sigjacobian(path,m)[a,b,c]} \approx \frac{\partial\left(\texttt{sig(path,m)[c]}\right)}{\partial\left(\texttt{path[a,b]}\right)}$$

`sigbackprop(s, path, m)`

This function does the basic calculation necessary to backpropagate derivatives through the `sig` function. If `path` has shape `(...,n,d)` and we are trying to calculate the derivatives of a scalar function F, and we have its derivatives with respect to `sig(X,m)` stored in an array `s` of shape `(...,siglength(d,m))`, then this function returns its derivatives with respect to `path` as an array of shape `(...,n,d)`.

$$\texttt{sigbackprop(s,path,m)} \approx \texttt{numpy.dot(sigjacobian(path,m),s)}$$

$$\texttt{sigbackprop(array(}\frac{\partial F}{\partial\texttt{sig(path,m)}}\texttt{),path,m)[a,b]} \approx \frac{\partial F}{\partial\texttt{path[a,b]}}$$

`sigjoin(sigs, segments, m, fixedLast=float("nan"))`

Given the signatures of paths in dimension d up to level m in a shape `(...,siglength(d,m))` array and an extra displacement for each path, stored as an array of shape `(...,d)`, returns the signatures of each of the paths concatenated with the extra displacement as an array of shape `(...,siglength(d,m))`. If the optional last argument `fixedLast` is provided, then it provides a common value for the last element of each of the displacements, and `segments` should have shape `(...,d-1)` – this is a way to create a time dimension automatically.

`sigjoinbackprop(derivs, sigs, segments, m, fixedLast=float("nan"))`

Returns the derivatives of some scalar function F with respect to both `sigs` and `segments` as a tuple, given the derivatives of F with respect to `sigjoin(sigs, segments, m, fixedLast)`. Returns both an array of the same shape as `sigs` and an array of the same shape as `segments`. If `fixedLast` is provided, also returns the derivative with respect to it in the same tuple.

`sigscale(sigs, scales, m)`

Given the signatures of paths in dimension d up to level m in a shape `(...,siglength(d,m))` array and a scaling factor for each dimension for each path, stored as an array of shape `(...,d)`, returns the signatures of each of the paths scaled in each dimension by the relevant scaling factor as an array of shape `(...,siglength(d,m))`.

`sigscalebackprop(derivs, sigs, segments, m)`

Returns the derivatives of some scalar function F with respect to both `sigs` and `scales` as a tuple, given the derivatives of F with respect to `sigscale(sigs, scales, m, fixedLast)`. Returns both an array of the same shape as `sigs` and an array of the same shape as `scales`.

3.2 Log signatures

Quick summary: To get the log signature of a `d`-dimensional path `p` up to level `m`, there are two steps, as follows.

```
s=iisignature.prepare(d,m)
logsignature=iisignature.logsig(p,s)
```

The rest of this section gives more details.

The algebra for calculating log signatures is explained in [6]. Several methods are available for calculating the log signature, which are identified by a letter.

D The default method, which is one of the methods C or S below, chosen automatically depending on the dimension and level requested.

C The compiled method, under which machine code is generated to calculate the BCH formula explicitly. Currently, the generated code is designed for both `x86` and `x86-64` on both Linux (System V, Mac etc.) and Windows systems. I don't know anyone using other systems, but the result is likely to be a crash.

O The BCH formula is expanded explicitly, but stored simply as a normal object. The object's instructions are followed to calculate the log signature. No code is written. This is simpler and potentially slower than the default method. It makes no particular assumptions about the platform, and so may be more broadly applicable.

S The log signature is calculated by first calculating the signature of the path, then explicitly evaluating its logarithm, and then projecting on to the basis.This is observed to be faster than using the BCH formula when the log signature is large (for example level 10 with dimension 3 or higher dimension). It may be more generally faster when the path has very many steps.

X The log signature is calculated by first calculating the signature of the path, then explicitly evaluating its logarithm. This logarithm is returned expanded in tensor space. This is used for testing purposes.

Log signatures are by default reported in the Lyndon basis in ascending order of level, with alphabetical ordering of Lyndon words within each level. A version of the standard or classical Hall basis is available instead by requesting it in the `prepare` function.

`logsiglength(d, m)`

The length of the log signature up to level m of a d-dimensional path. This value can be calculated using Witt's formula (see [7]). It is

$$\sum_{l=1}^{m} \frac{1}{l} \sum_{x|l} \mu\left(\frac{l}{x}\right) d^x$$

where μ is the Möbius function.

`prepare(d, m, methods=None)`

This does preliminary calculations and produces an object which is used for calculating log signatures of d-dimensional paths up to level m. The object produced is opaque. It is only used as an input to the `basis`, `info` and `logsig` functions.

It is a capsule or, on old versions of Python, a CObject. It cannot be pickled. This means that if you are using `multiprocessing`, you cannot pass it between "threads". You can run the function before creating the "threads", and use it in any thread - this works because it is fork-safe on non Windows platforms. On Windows, the function will be run separately in each background "thread".

The calculation can take a long time when d or m is large. The Global Interpreter Lock is not held during most of the calculation, so you can profitably do it in the background of a slow unrelated part of the initialization of your program. For example:

```
import iisignature, threading
def f():
        global s
        s=iisignature.prepare(2,10,"CS")
t = threading.Thread(target=f)
t.run()
#slow activity: theano.function, another prepare(),
#                keras compile, ...
t.join()
```

This function by default prepares to use only the default method. You can change this by supplying a string containing the letters of the methods you wish to use. For example, for $d = 3$ and $m = 10$, if you really wanted all methods to be available, you might run `prepare(3,10,"COSX")` – this takes a long time, because the BCH calculation is big.

If you want results in the standard Hall basis instead of in the Lyndon word basis, add an H anywhere into the method string. For example `prepare(2,4,"DH")`. When this is done, the result should be directly comparable to the output from the CoRoPa library [5].

The object returned never changes once it is created, except that `logsigbackprop` can add the preparation for the **S** method later.

`logsig(path, s, methods=None)`

The log signature of the d-dimensional path `path` up to level m, returned as a `numpy` array of shape `(...,logsiglength(d,m))`, where `s` is the result of calling `prepare(d,m[,...])`.

By default, this uses the calculation method (**C**, **O** or **S**) which is supported by `s` and comes first in the table above. You can restrict this by supplying as the final argument a string containing the letters of the methods you wish to be considered, this will probably be a one-letter string.

If you wish to use the **X** method, you have to ask for it here, and the output will have shape `(...,siglength(d,m))`.

`logsigbackprop(derivs, path, s, methods=None)`

Returns the derivatives of some scalar function F with respect to `path`, given the derivatives of F with respect to `logsig(path, s, methods)`. Returns an array of the same shape as `path`. The only methods supported are **S** (the default) and **X** (which is only used if `methods` is `'X'`). If the `'X'` method is not requested and `s` does not support the **S** method, then `s` is modified so it *does* support the **S** method.

`basis(s)`

The basis of bracketed expressions given as a tuple of unicode strings, for words of length no more than m on d letters, where `s` is the result of calling `prepare(d,m[,...])`. These are the bracketed expressions which the coefficients returned by `logsig` refer to.

If $d > 9$, the output of this function is not yet fixed. An example of how to parse the output of this function can be seen in the tests.

`info(s)`

If `s` is the result of calling `prepare(d,m[,...])`, then this returns a dictionary of properties of `s`, including a list of methods which it supports. This may be a useful diagnostic.

3.3 Linear rotational invariants

> **Quick summary:** To get all the linear rotational invariants of a two dimensional path `p` up to level `m`, there are two steps, as follows.
>
> ```
> s=iisignature.rotinv2dprepare(m,"a")
> invariants=iisignature.rotinv2d(p,s)
> ```
>
> The rest of this section gives more details.

The paper [4] explains how to find the linear subspace of signature space of two dimensional paths which is invariant under rotations of the path. The subspace is spanned by a set of vectors each of which lives in a single signature level, and all those levels are even. The functions in this section calculate them, and are a bit experimental. You may need to rescale them in a deep learning context.

`rotinv2dprepare(m,type)`

This prepares the way to find linear rotational invariants of signatures up to level `m` of 2d paths. The returned opaque object is used (but not modified) by the other functions in this section. `m` should be a small even number. `type` should be `"a"` if you want to return **a**ll the invariants.

Some invariants do not add information, because their values are products of other invariants. Set `type` to `"s"` if you want to exclude them. Internally, at each level, a basis for the invariant subspace with the known elements quotiented out is found using **s**ingular value decomposition from `numpy`. The exact result in this case is not guaranteed to be stable between versions. (An alternative, potentially faster, method of doing the same calculation uses the **Q**R decomposition as provided by `scipy`. You can get this by setting `type` to `"q"`. This will only work if you have `scipy` available, and may generate a strange but harmless warning message.[2]) (In addition, setting `type` to `"k"` means that *only* the invariants which are already **k**nown based on lower levels will be returned. This is used for testing.)

If m exceeds 10 this function can take a lot of time and memory.

`rotinv2d(path, s)`

The rotational invariants of the signature of the 2-dimensional path `path` up to level m, where `s` comes from `rotinv2dprepare(m,...)`. The result is returned as a `numpy` array of shape `(...,rotinv2dlength(s))`.

`rotinv2dlength(s)`

The number of rotational invariants which are found by the calculation defined by `s`, where `s` is the result of calling `rotinv2dprepare(m,type)`. When the type is `"a"`, this is just

$$\sum_{i=1}^{m/2} \binom{2i}{i}.$$

In common cases, the result is given in this table:

`m`	`"a"`	`"s"`	`"k"`
2	2	2	0
4	8	5	3
6	28	15	15
8	98	46	76
10	350	154	336
12	1274	522	1470
14	4706	1838	6230

[2] I have observed this only on Python 3.6 on Windows, and do not understand it. `https://stackoverflow.com/questions/45054949/calling-scipy-from-c-extension`

`rotinv2dcoeffs(s)`

The basis of rotational invariants which are found by the calculation defined by `s`, where `s` is the result of calling `rotinv2dprepare(...)`. The result is given as a tuple of 2d `numpy` arrays, where each row of element `i` is an element of the basis within level $(2i + 2)$ of the signature.

4 Implementation notes

The source code is easily found at `https://github.com/bottler/iisignature`. The extension module is defined in a single translation unit, `src/pythonsigs.cpp`. Here I explain the structure of the implementation which is located in various header files in the same directory. If you want to use the functionality of the library from `C++`, it should be easy just to include these header files.

`calcSignature.hpp` implements the functions `sig`, `sigbackprop`, `sigjacobian`, `sigjoin` and `sigjoinbackprop`, and `sigscale` and `sigscalebackprop`.

`logSigLength.hpp` implements the function `siglength` and `logsiglength`.

`iisignature_data/bchLyndon20.dat` is the file of Baker-Campbell-Hausdorff coefficients from Fernando Casas and Ander Murua available from [2]. It was calculated using their method described in [1]. You need to open this file and point the global variable `g_bchLyndon20_dat` to its entire contents.

`readBCHCoeffs.hpp` has facilities for reading the coefficients in `bchLyndon20.dat`.

`bch.hpp` implements calculations which manipulate elements of the Free Lie Algebra with generic coefficient objects. This uses `readBCHCoeffs.hpp`. The procedures are as explained in [6], and the design is similar to the python code `logsignature.py`.

`makeCompiledFunction.hpp` defines a structure `FunctionData` which describes a function which does some arithmetic on two arrays which is generic enough to concatenate log signatures. It has the ability to run such a function (`slowExplicitFunction`) and the ability to create an object `FunctionRunner` which represents a compiled version of the function. Currently no particular recent CPU capability is assumed - SSE2 is being used.

`logsig.hpp` uses `bch.hpp` and `makeCompiledFunction.hpp` to implement the `prepare` function.

The code required to solve the linear systems to convert a signature to a log signature (for the **S** method) is not provided. The addin relies on `numpy` to do this.

If you want to call this from your own `C++` code, you will need to provide a value for `interrupt`. A function which does nothing is fine. The idea is that it should be a function which returns if the calculation should continue and throws an exception if it wants the calculation to abort. None of these header files catch any exceptions - you can safely catch it in the calling code.

`rotationalInvariants.hpp` has functions to identify linear rotational invariants and their shuffle products.

5 Example code, Theano, Tensorflow, Torch and Keras

Simple examples of using many of the functions are in the test file at `https://github.com/bottler/iisignature/blob/master/tests/test_sig.py`.

`Theano` and `tensorflow` are Python frameworks for constructing calculation graphs, of the sort which is useful for deep learning. `Keras` is a Python framework for deep learning which uses either of them for its calculations. `iisignature` does not depend on these libraries. Pure Python code using `iisignature` inside `tensorflow`, `Theano` and `Keras` can be found in the source repository at `https://github.com/bottler/iisignature/tree/master/examples`.

- The modules `iisignature_theano`, `iisignature_tensorflow` and `iisignature_torch` provide operations `Sig`, `LogSig`, `SigJoin` and `SigScale` in `Theano`, `tensorflow` and `PyTorch` respectively which mirror the functions `sig`, `logsig`, `sigjoin` and `sigscale` in `iisignature`. The calculations are done via `iisignature`, and therefore always on the CPU.
- `iisignature_recurrent_keras` is a module which provides a recurrent layer in `Keras` using the `Theano` and `tensorflow` operations. There are variants of this file for compatibility with older versions of `Keras`. A similar module for `PyTorch` is given in `iisignature_recurrent_torch`.

Other files with names beginning with `demo` in the same directory provide demonstrations of this functionality.

6 Version history

Revision	Date	Highlights
0.23	2018-08-21	2 option for `sig` for partial signatures, return 64 bit floats from `sig`, Python 3.7 and Numpy 1.15 support
0.22	2017-10-02	mac installation fix, fix issues with 0.21 release
0.21	2017-09-20	`logsigbackprop`, speedups: Horner method for `'S'`, store `BasisElt` in order
0.20	2017-08-09	Batching, triangle optimised `'S'`, Mac install fix, `sigjoinbackprop` to `fixedLast`, `logsig` bug for paths of one point, `QR` method
0.19	2017-06-27	Rotational invariants, 32bit linux `'C'`, inputs don't have to be numpy arrays
0.18	2017-03-28	Hall basis, `info`, fix memory leak in `sigjoinbackprop` and `sigscalebackprop`, fix `sigscalebackprop`
0.17	2017-01-22	`sigscale`, fix `sigjoinbackprop` on Windows (code was miscompiled)
0.16	2016-08-23	Derivatives of signature
0.15	2016-06-20	Windows build

Improvements to the example code are not listed here, they can be seen on `github`.

Index of functions

References

[1] Fernando Casas and Ander Murua. "An efficient algorithm for computing the Baker-Campbell-Hausdorff series and some of its applications". In: *Journal of Mathematical Physics* 50.3 (Mar. 2009), p. 033513. eprint: http://arxiv.org/abs/0810.2656 (cit. on p. 7).

[2] Fernando Casas and Ander Murua. *The BCH formula and the symmetric BCH formula up to terms of degree 20*. http://www.ehu.eus/ccwmuura/bch.html (cit. on p. 7).

[3] Ilya Chevyrev and Andrey Kormilitzin. "A Primer on the Signature Method in Machine Learning". 2016. http://arxiv.org/abs/1603.03788 (cit. on p. 1).

[4] Joscha Diehl. "Rotation Invariants of Two Dimensional Curves Based on Iterated Integrals". 2013. http://arxiv.org/abs/1305.6883 (cit. on p. 6).

[5] Terry Lyons et al. *CoRoPa Computational Rough Paths (software library)*. 2010. http://coropa.sourceforge.net/ (cit. on p. 5).

[6] Jeremy Reizenstein. "Calculation of Iterated-Integral Signatures and Log Signatures". 2015. http://arxiv.org/abs/1712.02757 (cit. on pp. 4, 7).

[7] Wikipedia. *Necklace polynomial*. 2015. http://en.wikipedia.org/wiki/Necklace_polynomial (cit. on p. 4).