Singal Flow Analysis Documentation

Release 0.1.0

Daewon Lee

Mar 07, 2023

CONTENTS

SFA is a simulation framework, which provides useful data structures and functions for efficiently analyzing signal flow in complex networks.

CHAPTER

ONE

FEATURES

- Convenient data structures for analyzing multiple datasets with multiple algorithms.
- Support for visualizing simulation results and signal flow.
- Parallel simulations using multiprocessing.
- User-defined algorithms or datasets.

CHAPTER

TWO

DOCUMENTATION

2.1 Install

Currently, we recommend installing from distributed repositories such as GitHub. First, download a recent version of the repository as follows.

```
$ git clone https://github.com/dwgoon/sfa.git sfa
```

Now, you can install SFA Python package from the cloned directory.

```
$ cd sfa
$ python setup.py install
```

If you want to easily update the most recent stable version of the package from the repository, use `develop` option insead of `install`.

```
$ python setup.py develop
```

Now, running `git pull origin master` is enough to update the package from the repository.

If you don't have permission to the global `site-packages` directory, you can use the following flags: `--user`, `--home`, and `--prefix`. Refer to the official document in more details for using the flags.

For example, you can simply install the package with `--user` flag.

```
$ python setup.py install --user
```

Otherwise, you can also consider Python virtual environemnts.

2.2 Tutorials

2.2.1 Signal flow analysis

This brief tutorial will guide you to start utilizing SFA. We wrote this tutorial assuming users already have the overall knowledge about the original journal paper.

Creating algorithm object

`sfa.AlgorithmSet` deals with creating and managing the algorithm objects in SFA. Thus, we first need to create `sfa.AlgorithmSet` object.

```
>>> import sfa
>>> algs = sfa.AlgorithmSet()
```

Now, we can create algorithm objects with `sfa.AlgorithmSet`. Create *Signal Propagation (SP)* algorithm, which is designated by its abbreviation, `SP`.

```
>>> alg = algs.create('SP')
SP algorithm has been created.
>>> alg.abbr
'SP'
>>> alg
SignalPropagation object
```

As `sfa.AlgorithmSet` has the functionality of dictionary, we can also access the created algorithms using the abbreviations as keys.

```
>>> algs['SP']
SignalPropagation object
```

Setting hyperparameter values

Algorithms in SFA have hyperparameters that adjust and constrain the behavior of the algorithms. `ParameterSet`, a nested object defined in `sfa.Algorithm`, have member variables that contain the information about the various hyperparameters. The below shows the examples of the parameters.

```
>>> alg.params
<sfa.algorithms.sp.SignalPropagation.ParameterSet at 0x25b5a7e5550>
>>> alg.params.alpha
0.5
```

We can see the default value of `alpha` is 0.5. `alpha` is a hyperparameter that controls the proportion of signal flow in determining the next system state, *x(t+1)*, in the following formula.

$$x(t+1) = \alpha W x(t) + (1-\alpha) b$$

Thus, `0.5` means the algorithm reflects the effects of signal flow on half of estimating *x(t+1)*.

We can easily change the value of `alpha` by assigning a real value between 0 and 1.

```
>>> alg.params.alpha = 0.5
0.5
>>> alg.params.alpha = 0.9
>>> alg.params.alpha
0.9
```

Another hyperparameter is `apply_weight_norm`, which designates whether to use link weight normalization. The default value is `False`, but it is recommended to set it as `True`.

```
>>> alg.params.apply_weight_norm
False
>>> alg.params.apply_weight_norm = True
```

Refer to the documentation for more details about the other hyperparameters.

Creating data object

Creating and handling data objects in SFA are similar to those of algorithms. A data object is also designated by its abbreviation, as in the algorithm. For example, the datasets for Borisov et al. can be created using `BORISOV_2009` as follows.

```
>>> ds = sfa.DataSet()
>>> mdata = ds.create('BORISOV_2009')
BORISOV_2009 data has been created.
>>> mdata  # Multiple datasets.
{'120m_AUC_EGF=0.001+I=0.1': BorisovData object,
 '120m_AUC_EGF=0.001+I=1': BorisovData object,
 '120m_AUC_EGF=0.001+I=10': BorisovData object,
...
```

The above `mdata` or `ds['BORISOV_2009']` is a `dict` that contains multiple dataset objects with different conditions. For example, `120m_AUC_EGF=0.001+I=0.1` denotes the dataset was created by performing a simulation under the stimulation of 0.001M EGF and 0.1M insulin using the original ODE model, where the activity of a biomolecule was calculated by estimating the area under the curve (AUC) of the time profile.

We can select a dataset object by using the abbreviation.

```
>>> data = mdata['120m_AUC_EGF=0.001+I=0.1']
>>> data.abbr
'120m_AUC_EGF=0.001+I=0.1'
```

We can also consider a utility function in SFA, `sfa.get_avalue`, which arbitrarily selects a dataset object from the dictionary.

```
>>> data = sfa.get_avalue(mdata)
>>> data.abbr
'120m_AUC_EGF=0.001+I=0.1'
```

Actually, `sfa.get_avalue` returns the first item by applying the next() built-in fuction to a given `dict` object.

Accessing the members of data object

The data object (instantiated with a subclass of `sfa.Data`) has various data structures that are required for using `sfa.Algorithm`. For example, `sfa.Data` object has the information about network topology in `A` (adjacency matrix in `numpy`'s ndarray), `dg` (`NetworkX`'s DiGraph), and `n2i` (`dict` for mapping names to the indices of `A`).

```
>>> data.n2i  # Name to index mapper.
{'AKT': 0,
 'EGF': 1,
 'EGFR': 2,
 'ERK': 3,
```

(continues on next page)

(continued from previous page)

```
 'GAB1': 4,
 'GAB1_SHP2': 5,
 'GAB1_pSHP2': 6,
 'GS': 7,
 'I': 8,
 'IR': 9,
 'IRS': 10,
 'IRS_SHP2': 11,
 'MEK': 12,
 'PDK1': 13,
 'PI3K': 14,
 'PIP3': 15,
 'RAF': 16,
 'RAS': 17,
 'RasGAP': 18,
 'SFK': 19,
 'SHC': 20,
 'mTOR': 21}
>>> data.A[n2i['ERK'], n2i['MEK']]  # MEK -> ERK
1
>>> data.A[n2i['GAB1'], n2i['ERK']]  # ERK -| GAB1
-1
>>> data.A[n2i['ERK'], n2i['EGFR']]  # No link between EGFR and ERK.
0
>>> for src, trg, attr in data.dg.edges(data=True):
...     if attr['SIGN'] > 0:
...         print('%s -> %s'%(src, trg))
...     elif attr['SIGN'] < 0:
...         print('%s -| %s'%(src, trg))
...
AKT -> mTOR
AKT -| RAF
EGF -> EGFR
EGFR -> RasGAP
EGFR -> SFK
EGFR -> PI3K
EGFR -> GAB1
EGFR -> GAB1_pSHP2
EGFR -> SHC
EGFR -> GS
ERK -| GAB1
ERK -| GS
GAB1 -> GAB1_SHP2
GAB1 -> GAB1_pSHP2
GAB1 -> PI3K
GAB1 -> GS
GAB1 -> RasGAP
GAB1_SHP2 -> GAB1_pSHP2
GAB1_SHP2 -| RasGAP
GAB1_pSHP2 -> GS
GAB1_pSHP2 -| RasGAP
GS -> RAS
```

(continues on next page)

(continued from previous page)

```
I -> IR
IR -> RasGAP
IR -> IRS
IR -> SFK
IR -> PI3K
IRS -> IRS_SHP2
IRS -> GS
IRS -> PI3K
IRS_SHP2 -| RasGAP
MEK -> ERK
PDK1 -> AKT
PI3K -> PIP3
PIP3 -> PDK1
PIP3 -> IRS
PIP3 -> GAB1
RAF -> MEK
RAS -> RAF
RasGAP -| RAS
SFK -> IRS
SFK -> GAB1
SFK -> GAB1_pSHP2
SFK -> RAF
SHC -> GS
mTOR -> AKT
mTOR -| IRS
```

Analyzing data with algorithm

To make `sfa.Algorithm` work with `sfa.Data`, we should first assign the data object to the algorithm object.

```
>>> alg.params.alpha = 0.5
>>> alg.params.apply_weight_norm = True
>>> alg.data = data  # Assign the data object to the algorithm.
>>> alg.initialize()  # Initialize the algorithm object.
```

In the initization of the algorithm (calling `sfa.Algorithm.initialize`), the algorithm prepares estimaing signal flow by performing some necessary tasks such as link weight normalization.

```
>>> data.A[data.n2i['GAB1'], data.n2i['EGFR']]
1
>>> alg.W[data.n2i['GAB1'], data.n2i['EGFR']]
0.1889822365046136
```

Note that the element of the weight matrix is different from that of adjacency matrix.

One of the important tasks is to determine the values of the basal activity before analyzing signal flow. The effects of input stimulation or perturbation are basically reflected to the basal activity vector, b. For example, EGF stimulation can be reflected to b as follows.

```
>>> import numpy as np
>>> N = data.dg.number_of_nodes()  # The number of nodes; data.A.shape[0]
```

(continues on next page)

(continued from previous page)

```
>>> b = np.zeros((N,), dtype=np.float)
>>> b[data.n2i['EGF']] = 1
```

Now, we can perform the estimation of signal flow, and examine how the two outputs, ERK and AKT, have changed.

```
>>> xs1 = alg.compute(b) # xs: x at steady-state
>>> xs1
array([0.00155625, 0.5       , 0.25      , 0.00165546, 0.02951243,
       0.00659918, 0.03226491, 0.0367612 , 0.        , 0.        ,
       0.00608503, 0.0017566 , 0.00331091, 0.00401268, 0.02780067,
       0.01390033, 0.00662182, 0.00733528, 0.01601391, 0.03340766,
       0.04724556, 0.00055022])
>>> xs1[data.n2i['ERK']]
0.0016554557287082902
>>> xs1[data.n2i['AKT']]
0.0015562514037656679
```

We can see the signs of the two outputs are positive, which means ERK and AKT are upregulated by EGF stimulation.

Next, let's apply an inhibitory perturbation to the network. For example, we can perturb MEK by setting its basal activity as follows.

```
>>> b[data.n2i['MEK']] = -1
>>> b[data.n2i['EGF']], b[data.n2i['MEK']]
(1.0, -1.0)
>>> b
array([ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., -1.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

>>> xs2 = alg.compute(b)
>>> xs2[data.n2i['MEK']]
-0.4947084519007513
>>> xs2[data.n2i['ERK']]
-0.24735422595037565
>>> xs2[data.n2i['AKT']]
0.0018367951619137 94
```

At this time, the sign of ERK is negative, which means it is downregulated by MEK inhibition. On the other hand, AKT is not downregulated by the inhibition under EGF stimulation.

If we want to examine how the inhibition of MEK affects each node, we take the difference between the vectors of two results.

```
>>> dxs = xs2 - xs1  # Difference between the two results.
>>> ind_up = np.where(dxs > 0)[0]  # Indices of upregulated nodes
>>> ind_dn = np.where(dxs < 0)[0]  # Indices of downregulated nodes
>>> for idx in ind_up:
...     print(data.i2n[idx])  # data.i2n: Index to name mapper.
AKT
GAB1
GAB1_SHP2
GAB1_pSHP2
GS
IRS
```

(continues on next page)

(continued from previous page)

```
IRS_SHP2
PDK1
PI3K
PIP3
RAF
RAS
RasGAP
mTOR
>>> for idx in ind_dn:
...     print(data.i2n[idx])
ERK
MEK
```

This result shows that only MEK and ERK are upregulated by the inhibition of MEK under EGF stimulation.

Applying perturbation to link

In some cases, a perturbation should be reflected to link weight, not basal activity. For example, if we want to examine what happens when PI3K cannot send signal to its downstreams (i.e., the out-links of PI3K are removed).

```
>>> b = np.zeros((N,), dtype=np.float)
>>> b[data.n2i['EGF']] = 1
>>> alg.W[:, data.n2i['PI3K']] *= 0  # Remove all the out-link weights.
>>> xs3 = alg.compute(b)
>>> xs3[data.n2i['ERK']]
0.00172210494367554
>>> xs3[data.n2i['AKT']]
0.0
>>> dxs = xs3 - xs1  # xs1 is the same as the previously computed one.
>>> dxs[data.n2i['ERK']]
6.664921496724974e-05
>>> dxs[data.n2i['AKT']]
-0.0015562514037656679
```

We can see that AKT is downregulated if all out-links of PI3K are lost.

Estimating signal flows

The estimation of signal flow is defined as the multiplication of link weight and activity of the source node. The activity is usually is the steady-state activity

$$F(t)_{ij} = W_{ij} \cdot x(t)_j$$

Follwing the definition, we can compute the signal flow as follows.

```
>>> alg.initialize()  # Obtain the intact weight matrix.
>>> W1 = alg.W.copy()  # Get a copy of the weight matrix.
>>> F1 = W1*xs1  # Element-wise multiplication of each row and xs1
```

Note that the above code snippet is not matix-vector multiplication, but it is element-wise multiplication of vectors (ndarray in NumPy). The following shows some of the estimated signal flows.

```
>>> F1[data.n2i['PIP3'], data.n2i['PI3K']]
0.02780066830505488
>>> F1[data.n2i['ERK'], data.n2i['MEK']]
0.0033109114574165805
>>> F1[data.n2i['GAB1'], data.n2i['ERK']]
-0.0005852919858618747
```

If we want to compare the two conditions, we can compute the net signal flow as follows.

$$F_{net} = F_{c2} - F_{c1}$$

Let's use the PI3K example of "*Applying perturbation to link*" again.

```
>> W3 = alg.W.copy()  # alg.W is the intact one.
>> W3[:, data.n2i['PI3K']] *= 0  # Apply the PI3K
>>> F3 = W3*xs3
>>> Fnet = F3 - F1  # Net signal flow.
>>> Fnet[:, data.n2i['PI3K']]
>>> ir, ic = data.A.nonzero()
>>> for i in range(ir.size):
...     idx_trg, idx_src = ir[i], ic[i]
...     src = data.i2n[idx_src]
...     trg = data.i2n[idx_trg]
...     sf = Fnet[idx_trg, idx_src]  # Signal flow
...     print("Net signal flow from %s to %s: %f"%(src, trg, sf))
Net signal flow from PDK1 to AKT: -0.002837
Net signal flow from mTOR to AKT: -0.000275
Net signal flow from EGF to EGFR: 0.000000
Net signal flow from MEK to ERK: 0.000133
Net signal flow from EGFR to GAB1: 0.000000
Net signal flow from ERK to GAB1: -0.000024
Net signal flow from PIP3 to GAB1: -0.004013
Net signal flow from SFK to GAB1: 0.000000
Net signal flow from GAB1 to GAB1_SHP2: -0.000903
Net signal flow from EGFR to GAB1_pSHP2: 0.000000
Net signal flow from GAB1 to GAB1_pSHP2: -0.000451
Net signal flow from GAB1_SHP2 to GAB1_pSHP2: -0.000160
Net signal flow from SFK to GAB1_pSHP2: 0.000000
Net signal flow from EGFR to GS: 0.000000
Net signal flow from ERK to GS: -0.000019
Net signal flow from GAB1 to GS: -0.000368
Net signal flow from GAB1_pSHP2 to GS: -0.000088
Net signal flow from IRS to GS: -0.000450
Net signal flow from SHC to GS: 0.000000
Net signal flow from I to IR: 0.000000
Net signal flow from IR to IRS: 0.000000
Net signal flow from PIP3 to IRS: -0.004013
Net signal flow from SFK to IRS: 0.000000
Net signal flow from mTOR to IRS: 0.000195
Net signal flow from IRS to IRS_SHP2: -0.001102
Net signal flow from RAF to MEK: 0.000267
Net signal flow from PIP3 to PDK1: -0.008025
Net signal flow from EGFR to PI3K: 0.000000
```

(continues on next page)

(continued from previous page)

```
Net signal flow from GAB1 to PI3K: -0.000451
Net signal flow from IR to PI3K: 0.000000
Net signal flow from IRS to PI3K: -0.000551
Net signal flow from PI3K to PIP3: -0.027801
Net signal flow from AKT to RAF: 0.000635
Net signal flow from RAS to RAF: -0.000102
Net signal flow from SFK to RAF: 0.000000
Net signal flow from GS to RAS: -0.000327
Net signal flow from RasGAP to RAS: -0.000027
Net signal flow from EGFR to RasGAP: 0.000000
Net signal flow from GAB1 to RasGAP: -0.000368
Net signal flow from GAB1_SHP2 to RasGAP: 0.000130
Net signal flow from GAB1_pSHP2 to RasGAP: 0.000088
Net signal flow from IR to RasGAP: 0.000000
Net signal flow from IRS_SHP2 to RasGAP: 0.000225
Net signal flow from EGFR to SFK: 0.000000
Net signal flow from IR to SFK: 0.000000
Net signal flow from EGFR to SHC: 0.000000
Net signal flow from AKT to mTOR: -0.001100
```

We can see some links have no change in their signal flows between the two conditions. Obviously, the signal flow from PI3K to PIP3 has decreased due to the perturbation. However, the depletion of all out-links of PI3K has upregulated the signal flow from MEK to ERK (i.e., positive value).

Creating a dataset with network structure

- Describe how to define own datasets only with network topology.
- Explanation for the members of Data class.

2.2.2 Discovery of control targets

to be updated...

2.3 Data

2.3.1 Defining a new data with network structure

To create a new data with a network structure, it is required to define a derived class of `sfa.base.Data` class. In `__init__` of the class, we need to create four objects.

#	Member	Description
1	_A	2-dimensional `numpy.ndarray` for adjacency matrix.
2	_dg	`NetworkX.DiGraph` object.
3	_n2i	`dict` for mapping name to index.
4	_i2n	`dict` for mapping index to name.

The underscore _ of member name means the member is a protected member, which is defined not to be directly

accessed (refer to this stackoverflow answer). Instead, the four members can be accessed through property. The underscored members (protected members) should be used only in the methods of the class such as `__init__`.

```
>>> obj._A  # We can access like this, but defined not to.
>>> obj.A  # Instead, access the member like this.
```

Now, let's define a child class of `sfa.base.Data` for a simple 3-node cascade as a toy example.

```
import numpy as np
import networkx as nx

import sfa

class ThreeNodeCascade(sfa.base.Data):
    def __init__(self):
        super().__init__()
        self._abbr = "TNC"  # Abbreviation for this data.
        self._name = "A simple three node cascade"  # Full name

        # Create name to index mapper and index to name mapper.
        self._n2i = {"A": 0, "B": 1, "C": 2}  # Name to index
        self._i2n = {idx: name  for name, idx in self._n2i.items()}

        # Create Directed graph object of NetworkX.
        self._dg = nx.DiGraph()
        self._dg.add_edge('A', 'B', attr_dict={"sign": +1})
        self._dg.add_edge('B', 'C', attr_dict={"sign": +1})

        # Create adjacency matrix with signs.
        n = self._dg.number_of_nodes()
        self._A = np.zeros((n, n), dtype=np.float)

        for (src, tgt, attr) in self._dg.edges(data=True):
            isrc = self._n2i[src]
            itgt = self._n2i[tgt]
            sign = attr['attr_dict']['sign']
            self._A[itgt, isrc] = sign

    # end of def __init__
# end of def class

if __name__ == "__main__":
    data = ThreeNodeCascade()

    algs = sfa.AlgorithmSet()
    alg = algs.create('SP')
    alg.data = data
    alg.params.apply_weight_norm = True
    alg.initialize()

    n = data.A.shape[0]
    b = np.zeros((n,))  # Basal activity

    # Set node A to be activated
```

(continues on next page)

(continued from previous page)

```
    # by assigning 1 to its basal activity.
    b[data.n2i['A']] = 1

    # Compute the activity at steady-state.
    x = alg.compute(b)

    print("[Activity at steady-state]")
    print(x)
```

The above code is very straightforward, if you are familiar with the functionalities of `numpy` and `networkx` packages. The following is the result of executing the code.

```
SP algorithm has been created.
[Activity at steady-state]
[0.5   0.25  0.125]
```

If you want to see both the name and its activity, use `n2i` or `i2n`.

```
>>> for i, act in enumerate(x):
...     print(data.i2n[i], act)
A 0.5
B 0.25
C 0.125
>>> idx = data.n2i['B']
>>> x[idx]
0.25
```

Now, it's a little bit better to read.

For a large-scale network, it is almost impossible to write node names and their relationships one by one in the code. Thus, `sfa` provides some utility functions to create the data class.

If network structure information is defined in a text file such as SIF file, we can utilize `sfa.read_sif` function. `sfa.read_sif` reads the text file and returns `A`, `n2i` and `dg` objects that are required to define the data class.

Let's go back to the toy example.

```
A   +   B
B   +   C
```

The network structure can be described in SIF format like the above.

```
import os
import sfa

class ThreeNodeCascade(sfa.base.Data):
    def __init__(self):
        super().__init__()
        self._abbr = "TNC"
        self._name = "A simple three node cascade"

        # Specify the file path for network file in SIF.
        dpath = os.path.dirname(__file__)
        fpath = os.path.join(dpath, 'network.sif')
```

(continues on next page)

(continued from previous page)

```
        # Use read_sif function.
        A, n2i, dg = sfa.read_sif(fpath, as_nx=True)
        self._A = A
        self._n2i = n2i
        self._dg = dg
        self._i2n = {idx: name for name, idx in n2i.items()}
    # end of def __init__
# end of def class

if __name__ == "__main__":
    data = ThreeNodeCascade()

    algs = sfa.AlgorithmSet()
    alg = algs.create('SP')
    alg.data = data
    alg.params.apply_weight_norm = True
    alg.initialize()

    n = data.A.shape[0]
    b = np.zeros((n,))

    # Activate node B at this time.
    b[data.n2i['B']] = 1
    x = alg.compute(b)

    print("[Activity at steady-state]")
    for i, act in enumerate(x):
        print("[Node %s] %f"%(data.i2n[i], act))
```

In the above code, all you need to do is just putting the file path of network in `sfa.read_sif`. I recommend utilizing the above code snippet as a template for creating your own network structure data. The result of the above code is as follows.

```
SP algorithm has been created.
[Activity at steady-state]
[Node A] 0.000000
[Node B] 0.500000
[Node C] 0.250000
```

In this example, positive and negative signs of links are defined as + and -, respectively, in the file. However, if the sign or interaction information is defined differently, you can specify it with `signs` keyword argument of `sfa.read_sif`. For example, if a network file have `activates` and `inhibits` as the signs for positive and negative links, respectively, we can call `sfa.read_sif` function as follows.

```
>>> signs = {'activates':1, 'inhibits':-1}
>>> sfa.read_sif("network.sif", signs=signs, as_nx=True)
(array([[ 0,  0,  0],
        [ 1,  0,  0],
        [ 0, -1,  0]]),
 {'A': 0, 'B': 1, 'C': 2},
 <networkx.classes.digraph.DiGraph at 0x2ce1503c7b8>)
```

If your network structure is defined in a different file format (not SIF), you should write some code lines for parsing the network strcuture data.

2.3.2 Defining a new data for validating algorithm

- Describe how to define own datasets only with experimental data.
- Explanation for the members of Data class for validation.

2.4 Algorithm

2.4.1 Defining a new algorithm

2.5 Control

to be updated…

2.6 Visualization

2.7 Simulation

2.7.1 Randomizing network structure

2.7.2 Randomizing link weights

2.7.3 Multiple algorithms vs multiple datasets

2.7.4 Parallel processing

2.8 Development

to be updated…

2.9 sfa

2.9.1 setup module

2.9.2 sfa package

Subpackages

sfa.algorithms package

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search