PyPhi Documentation

Release v0.8.1

Will Mayner

May 10, 2017

Contents

PyPhi is a Python library for computing integrated information.

To report issues, please use the issue tracker on the GitHub repository. Bug reports and pull requests are welcome.

CHAPTER 1

Getting started

The *Network* object is the main object on which computations are performed. It represents the network of interest.

The *Subsystem* object is the secondary object; it represents a subsystem of a network. Φ is defined on subsystems.

The *compute* module is the main entry-point for the library. It contains methods for calculating concepts, constellations, complexes, etc.

The best way to familiarize yourself with the software is to go through the examples. All the examples dicussed are available in the *examples* module, so you can follow along in a REPL. The relevant functions are listed at the beginning of each example.

Usage and Examples

All the examples discussed here are available from the *pyphi.examples* module, so you can follow along with a Python REPL.

The relevant functions are listed at the beginning of each example.

IIT 3.0 Paper (2014)

This section is meant to serve as a companion to the paper From the Phenomenology to the Mechanisms of Consciousness: Integrated Information Theory 3.0 by Oizumi, Albantakis, and Tononi, and as a demonstration of how to use PyPhi. Readers are encouraged to follow along and analyze the systems shown in the figures, hopefully becoming more familiar with both the theory and the software in the process.

First, start a Python 3 REPL by running `python3` on the command line. Then import PyPhi and NumPy:

```
>>> import pyphi
>>> import numpy as np
```

Figure 1

Existence: Mechanisms in a state having causal power.

For the first figure, we'll demonstrate how to set up a network and a candidate set. In PyPhi, networks are built by specifying a transition probability matrix and (optionally) a connectivity matrix. (If no connectivity matrix is given, full connectivity is assumed.) So, to set up the system shown in Figure 1, we'll start by defining its TPM.

Note: The TPM in the figure is given in **state-by-state** form; there is a row and a column for each state. However, in PyPhi, we use a more compact representation: **state-by-node** form, in which there is a row for each state, but a column for each node. The i, j^{th} entry gives the probability that the j^{th} node is on in the i^{th} state. For more information on how TPMs are represented in PyPhi, see the documentation for the *network* module and the explanation of *LOLI: Low-Order bits correspond to Low-Index nodes*.

In the figure, the TPM is shown only for the candidate set. We'll define the entire network's TPM. Also, nodes D, E and F are not assigned mechanisms; for the purposes of this example we will assume they are **OR** gates. With that assumption, we get the following TPM (before copying and pasting, see note below):

```
>>> tpm = np.array([
...     [0, 0, 0, 0, 0, 0],
...     [0, 0, 1, 0, 0, 0],
...     [1, 0, 1, 0, 1, 0],
...     [1, 0, 0, 0, 1, 0],
...     [1, 0, 0, 0, 0, 0],
...     [1, 1, 1, 0, 0, 0],
...     [1, 0, 1, 0, 1, 0],
...     [1, 1, 0, 0, 1, 0],
...     [1, 0, 0, 0, 0, 0],
...     [1, 0, 1, 0, 0, 0],
...     [1, 0, 1, 0, 1, 0],
...     [1, 0, 0, 0, 1, 0],
...     [1, 0, 0, 0, 0, 0],
...     [1, 1, 1, 0, 0, 0],
...     [1, 0, 1, 0, 1, 0],
...     [1, 1, 0, 0, 1, 0],
...     [0, 0, 0, 0, 0, 0],
...     [0, 0, 1, 0, 0, 0],
...     [1, 0, 1, 0, 1, 0],
...     [1, 0, 0, 0, 1, 0],
...     [1, 0, 0, 0, 0, 0],
...     [1, 1, 1, 0, 0, 0],
...     [1, 0, 1, 0, 1, 0],
...     [1, 1, 0, 0, 1, 0],
...     [1, 0, 0, 0, 0, 0],
...     [1, 0, 1, 0, 0, 0],
...     [1, 0, 1, 0, 1, 0],
...     [1, 0, 0, 0, 1, 0],
...     [1, 0, 0, 0, 0, 0],
...     [1, 1, 1, 0, 0, 0],
...     [1, 0, 1, 0, 1, 0],
...     [1, 1, 0, 0, 1, 0],
...     [0, 0, 0, 0, 0, 0],
...     [0, 0, 1, 0, 0, 0],
...     [1, 0, 1, 0, 1, 0],
...     [1, 0, 0, 0, 1, 0],
...     [1, 0, 0, 0, 0, 0],
...     [1, 1, 1, 0, 0, 0],
```

```
...      [1, 0, 1, 0, 1, 0],
...      [1, 1, 0, 0, 1, 0],
...      [1, 0, 0, 0, 0, 0],
...      [1, 0, 1, 0, 0, 0],
...      [1, 0, 1, 0, 1, 0],
...      [1, 0, 0, 0, 1, 0],
...      [1, 0, 0, 0, 0, 0],
...      [1, 1, 1, 0, 0, 0],
...      [1, 0, 1, 0, 1, 0],
...      [1, 1, 0, 0, 1, 0],
...      [0, 0, 0, 0, 0, 0],
...      [0, 0, 1, 0, 0, 0],
...      [1, 0, 1, 0, 1, 0],
...      [1, 0, 0, 0, 1, 0],
...      [1, 0, 0, 0, 0, 0],
...      [1, 1, 1, 0, 0, 0],
...      [1, 0, 1, 0, 1, 0],
...      [1, 1, 0, 0, 1, 0],
...      [1, 0, 0, 0, 0, 0],
...      [1, 0, 1, 0, 0, 0],
...      [1, 0, 1, 0, 1, 0],
...      [1, 0, 0, 0, 1, 0],
...      [1, 0, 0, 0, 0, 0],
...      [1, 1, 1, 0, 0, 0],
...      [1, 0, 1, 0, 1, 0],
...      [1, 1, 0, 0, 1, 0]
... ])
```

Note: This network is already built for you; you can get it from the *pyphi.examples* module with `network = pyphi.examples.fig1a()`. The TPM can then be accessed with `network.tpm`.

Next we'll define the connectivity matrix. In PyPhi, the i, j^{th} entry in a connectivity matrix indicates whether node i is connected to node j. Thus, this network's connectivity matrix is

```
>>> cm = np.array([
...      [0, 1, 1, 0, 0, 0],
...      [1, 0, 1, 0, 1, 0],
...      [1, 1, 0, 0, 0, 0],
...      [1, 0, 0, 0, 0, 0],
...      [0, 0, 0, 0, 0, 0],
...      [0, 0, 0, 0, 0, 0]
... ])
```

Now we can pass the TPM and connectivity matrix as arguments to the network constructor:

```
>>> network = pyphi.Network(tpm, connectivity_matrix=cm)
```

Now the network shown in the figure is stored in a variable called `network`. You can find more information about the network object we just created by running `help(network)` or by consulting the API documentation for *Network*.

The next step is to define the candidate set shown in the figure, consisting of nodes A, B and C. In PyPhi, a candidate set for Φ evaluation is represented by the *Subsystem* class. Subsystems are built by giving the network it is a part of, the state of the network, and indices of the nodes to be included in the subsystem. So, we define our candidate set like so:

```
>>> state = (1, 0, 0, 0, 1, 0)
>>> ABC = pyphi.Subsystem(network, state, [0, 1, 2])
```

For more information on the subsystem object, see the API documentation for *Subsystem*.

That covers the basic workflow with PyPhi and introduces the two types of objects we use to represent and analyze networks. First you define the network of interest with a TPM and connectivity matrix, then you define a candidate set you want to analyze.

Figure 3

Information requires selectivity.

(A)

We'll start by setting up the subsytem depicted in the figure and labeling the nodes. In this case, the subsystem is just the entire network.

```
>>> network = pyphi.examples.fig3a()
>>> state = (1, 0, 0, 0)
>>> subsystem = pyphi.Subsystem(network, state, range(network.size))
>>> A, B, C, D = subsystem.node_indices
```

Since the connections are noisy, we see that $A = 1$ is unselective; all past states are equally likely:

```
>>> subsystem.cause_repertoire((A,), (B, C, D))
array([[[[ 0.125,  0.125],
         [ 0.125,  0.125]],

        [[ 0.125,  0.125],
         [ 0.125,  0.125]]]])
```

And this gives us zero cause information:

```
>>> subsystem.cause_info((A,), (B, C, D))
0.0
```

(B)

The same as (A) but without noisy connections:

```
>>> network = pyphi.examples.fig3b()
>>> subsystem = pyphi.Subsystem(network, state, range(network.size))
>>> A, B, C, D = subsystem.node_indices
```

Now, A's cause repertoire is maximally selective.

```
>>> cr = subsystem.cause_repertoire((A,), (B, C, D))
>>> cr
array([[[[ 0.,  0.],
         [ 0.,  0.]],

        [[ 0.,  0.],
         [ 0.,  1.]]]])
```

Since the cause repertoire is over the purview BCD, the first dimension (which corresponds to A's states) is a singleton. We can squeeze out A's singleton dimension with

```
>>> cr = cr.squeeze()
```

and now we can see that the probability of B, C, and D having been all on is 1:

```
>>> cr[(1, 1, 1)]
1.0
```

Now the cause information specified by $A = 1$ is 1.5:

```
>>> subsystem.cause_info((A,), (B, C, D))
1.5
```

(C)

The same as (B) but with $A = 0$:

```
>>> state = (0, 0, 0, 0)
>>> subsystem = pyphi.Subsystem(network, state, range(network.size))
>>> A, B, C, D = subsystem.node_indices
```

And here the cause repertoire is minimally selective, only ruling out the state where B, C, and D were all on:

```
>>> subsystem.cause_repertoire((A,), (B, C, D))
array([[[[ 0.14285714,  0.14285714],
         [ 0.14285714,  0.14285714]],

        [[ 0.14285714,  0.14285714],
         [ 0.14285714,  0.        ]]]])
```

And so we have less cause information:

```
>>> subsystem.cause_info((A,), (B, C, D))
0.214284
```

Figure 4

Information: "Differences that make a difference to a system from its own intrinsic perspective."

First we'll get the network from the examples module, set up a subsystem, and label the nodes, as usual:

```
>>> network = pyphi.examples.fig4()
>>> state = (1, 0, 0)
>>> subsystem = pyphi.Subsystem(network, state, range(network.size))
>>> A, B, C = subsystem.node_indices
```

Then we'll compute the cause and effect repertoires of mechanism A over purview ABC:

```
>>> subsystem.cause_repertoire((A,), (A, B, C))
array([[[ 0.        ,  0.16666667],
        [ 0.16666667,  0.16666667]],

       [[ 0.        ,  0.16666667],
```

```
        [ 0.16666667,  0.16666667]]])
>>> subsystem.effect_repertoire((A,), (A, B, C))
array([[[ 0.0625,  0.0625],
        [ 0.0625,  0.0625]],

       [[ 0.1875,  0.1875],
        [ 0.1875,  0.1875]]])
```

And the unconstrained repertoires over the same (these functions don't take a mechanism; they only take a purview):

```
>>> subsystem.unconstrained_cause_repertoire((A, B, C))
array([[[ 0.125,  0.125],
        [ 0.125,  0.125]],

       [[ 0.125,  0.125],
        [ 0.125,  0.125]]])
>>> subsystem.unconstrained_effect_repertoire((A, B, C))
array([[[ 0.09375,  0.09375],
        [ 0.03125,  0.03125]],

       [[ 0.28125,  0.28125],
        [ 0.09375,  0.09375]]])
```

The Earth Mover's distance between them gives the cause and effect information:

```
>>> subsystem.cause_info((A,), (A, B, C))
0.333332
>>> subsystem.effect_info((A,), (A, B, C))
0.25
```

And the minimum of those gives the cause-effect information:

```
>>> subsystem.cause_effect_info((A,), (A, B, C))
0.25
```

Figure 5

A mechanism generates information only if it has both selective causes and selective effects within the system.

(A)

```
>>> network = pyphi.examples.fig5a()
>>> state = (1, 1, 1)
>>> subsystem = pyphi.Subsystem(network, state, range(network.size))
>>> A, B, C = subsystem.node_indices
```

A has inputs, so its cause repertoire is selective and it has cause information:

```
>>> subsystem.cause_repertoire((A,), (A, B, C))
array([[[ 0. ,  0. ],
        [ 0. ,  0.5]],

       [[ 0. ,  0. ],
        [ 0. ,  0.5]]])
```

```
>>> subsystem.cause_info((A,), (A, B, C))
1.0
```

But because it has no outputs, its effect repertoire no different from the unconstrained effect repertoire, so it has no effect information:

```
>>> np.array_equal(subsystem.effect_repertoire((A,), (A, B, C)),
...                subsystem.unconstrained_effect_repertoire((A, B, C)))
True
>>> subsystem.effect_info((A,), (A, B, C))
0.0
```

And thus its cause effect information is zero.

```
>>> subsystem.cause_effect_info((A,), (A, B, C))
0.0
```

(B)

```
>>> network = pyphi.examples.fig5b()
>>> state = (1, 0, 0)
>>> subsystem = pyphi.Subsystem(network, state, range(network.size))
>>> A, B, C = subsystem.node_indices
```

Symmetrically, A now has outputs, so its effect repertoire is selective and it has effect information:

```
>>> subsystem.effect_repertoire((A,), (A, B, C))
array([[[ 0.,  0.],
        [ 0.,  0.]],

       [[ 0.,  0.],
        [ 0.,  1.]]])
>>> subsystem.effect_info((A,), (A, B, C))
0.5
```

But because it now has no inputs, its cause repertoire is no different from the unconstrained effect repertoire, so it has no cause information:

```
>>> np.array_equal(subsystem.cause_repertoire((A,), (A, B, C)),
...                subsystem.unconstrained_cause_repertoire((A, B, C)))
True
>>> subsystem.cause_info((A,), (A, B, C))
0.0
```

And its cause effect information is again zero.

```
>>> subsystem.cause_effect_info((A,), (A, B, C))
0.0
```

Figure 6

Integrated information: The information generated by the whole that is irreducible to the information generated by its parts.

```
>>> network = pyphi.examples.fig6()
>>> state = (1, 0, 0)
>>> subsystem = pyphi.Subsystem(network, state, range(network.size))
>>> ABC = subsystem.node_indices
```

Here we demonstrate the functions that find the minimum information partition a mechanism over a purview:

```
>>> mip_c = subsystem.mip_past(ABC, ABC)
>>> mip_e = subsystem.mip_future(ABC, ABC)
```

These objects contain the $\varphi^{\text{MIP}}_{\text{cause}}$ and $\varphi^{\text{MIP}}_{\text{effect}}$ values in their respective `phi` attributes, and the minimal partitions in their `partition` attributes:

```
>>> mip_c.phi
0.499999
>>> mip_c.partition
0     1,2
-- X -----
[]   0,1,2
>>> mip_e.phi
0.25
>>> mip_e.partition
[]   0,1,2
-- X -----
1     0,2
```

For more information on these objects, see the API documentation for the *Mip* class, or use `help(mip_c)`.

Note that the minimal partition found for the past is

$$\frac{A^c}{[]} \times \frac{BC^c}{ABC^p},$$

rather than the one shown in the figure. However, both partitions result in a difference of 0.5 between the unpartitioned and partitioned cause repertoires. So we see that in small networks like this, there can be multiple choices of partition that yield the same, minimal φ^{MIP}. In these cases, which partition the software chooses is left undefined.

Figure 7

A mechanism generates integrated information only if it has both integrated causes and integrated effects.

It is left as an exercise for the reader to use the subsystem methods `mip_past` and `mip_future`, introduced in the previous section, to demonstrate the points made in Figure 7.

To avoid building TPMs and connectivity matrices by hand, one can use the graphical user interface for PyPhi available online at http://integratedinformationtheory.org/calculate.html. You can build the networks shown in the figure there, and then use the **Export** button to obtain a JSON file representing the network. You can then import the file into Python with the `json` module:

```
import json
with open('path/to/network.json') as json_file:
    network_dictionary = json.load(json_file)
```

The TPM and connectivity matrix can then be looked up with the keys `'tpm'` and `'cm'`:

```
tpm = network_dictionary['tpm']
cm = network_dictionary['cm']
```

For your convenience, there is a function that does this for you: *pyphi.network.from_json()* that takes a path to a JSON file and returns a PyPhi network object.

Figure 8

The maximally integrated cause repertoire over the power set of purviews is the "core cause" specified by a mechanism.

```
>>> network = pyphi.examples.fig8()
>>> state = (1, 0, 0)
>>> subsystem = pyphi.Subsystem(network, state, range(network.size))
>>> A, B, C = subsystem.node_indices
```

To find the core cause of a mechanism over all purviews, we just use the subsystem method of that name:

```
>>> core_cause = subsystem.core_cause((B, C))
>>> core_cause.phi
0.333334
```

For a detailed description of the objects returned by the *core_cause()* and *core_effect()* methods, see the API documentation for *Mice* or use `help(core_cause)`.

Figure 9

A mechanism that specifies a maximally irreducible cause-effect repertoire.

This figure and the next few use the same network as in Figure 8, so we don't need to reassign the `network` and `subsystem` variables.

Together, the core cause and core effect of a mechanism specify a "concept." In PyPhi, this is represented by the *Concept* object. Concepts are computed using the *concept()* method of a subsystem:

```
>>> concept_A = subsystem.concept((A,))
>>> concept_A.phi
0.166667
```

As usual, please consult the API documentation or use `help(concept_A)` for a detailed description of the *Concept* object.

Figure 10

Information: A conceptual structure C (constellation of concepts) is the set of all concepts generated by a set of elements in a state.

For functions of entire subsystems rather than mechanisms within them, we use the *pyphi.compute* module. In this figure, we see the constellation of concepts of the powerset of ABC's mechanisms. We can compute the constellation of the subsystem like so:

```
>>> constellation = pyphi.compute.constellation(subsystem)
```

And verify that the φ values match (rounding to two decimal places):

```
>>> [round(concept.phi, 2) for concept in constellation]
[0.17, 0.17, 0.25, 0.25, 0.33, 0.5]
```

The null concept (the small black cross shown in concept-space) is available as an attribute of the subsystem:

```
>>> subsystem.null_concept.phi
0
```

Figure 11

Assessing the conceptual information CI of a conceptual structure (constellation of concepts).

Conceptual information can be computed using the function named, as you might expect, *conceptual_information()*:

```
>>> pyphi.compute.conceptual_information(subsystem)
2.1111089999999999
```

Figure 12

Assessing the integrated conceptual information Φ of a constellation C.

To calculate Φ^{MIP} for a candidate set, we use the function *big_mip()*:

```
>>> big_mip = pyphi.compute.big_mip(subsystem)
```

The returned value is a large object containing the Φ^{MIP} value, the minimal cut, the constellation of concepts of the whole set and that of the partitioned set $C^{\text{MIP}}_{\rightarrow}$, the total calculation time, the calculation time for just the unpartitioned constellation, a reference to the subsystem that was analyzed, and a reference to the subsystem with the minimal unidirectional cut applied. For details see the API documentation for *BigMip* or use `help(big_mip)`.

We can verify that the Φ^{MIP} value and minimal cut are as shown in the figure:

```
>>> big_mip.phi
1.9166650000000001
>>> big_mip.cut
Cut (0, 1) --//--> (2,)
```

Note: A note on how to interpret the *Cut* object: it has two attributes, `severed` and `intact`. The connections going from the nodes in the `severed` set to those in the `intact` set are the connections removed by the cut.

Figure 13

A set of elements generates integrated conceptual information Φ only if each subset has both causes and effects in the rest of the set.

It is left as an exercise for the reader to demonstrate that of the networks shown, only **(B)** has $\Phi > 0$.

Figure 14

A complex: A local maximum of integrated conceptual information Φ.

```
>>> network = pyphi.examples.fig14()
>>> state = (1, 0, 0, 0, 1, 0)
```

To find the subsystem within a network that is the main complex, we use the function of that name, which returns a *BigMip* object:

```
>>> main_complex = pyphi.compute.main_complex(network, state)
```

And we see that the nodes in the complex are indeed A, B, and C:

```
>>> main_complex.subsystem.nodes
(n0, n1, n2)
```

Figure 15

A quale: The maximally irreducible conceptual structure (MICS) generated by a complex.

PyPhi does not provide any way to visualize a constellation out-of-the-box, but you can use the visual interface at http://integratedinformationtheory.org/calculate.html to view a constellation in a 3D projection of qualia space. The network in the figure is already built for you; click the **Load Example** button and select "IIT 3.0 Paper, Figure 1" (this network is the same as the candidate set in Figure 1).

Figure 16

A system can condense into a major complex and minor complexes that may or may not interact with it.

For this figure, we omit nodes H, I, J, K and L, since the TPM of the full 12-node network is very large, and the point can be illustrated without them.

```
>>> network = pyphi.examples.fig16()
>>> state = (1, 0, 0, 1, 1, 1, 0)
```

To find the maximal set of non-overlapping complexes that a network condenses into, use *condensed()*:

```
>>> condensed = pyphi.compute.condensed(network, state)
```

We find that there are 2 complexes: the major complex ABC with $\Phi \approx 1.92$, and a minor complex FG with $\Phi \approx 0.069$ (note that there is typo in the figure: FG's Φ value should be 0.069). Furthermore, the program has been updated to only consider background conditions of current states, not past states; as a result the minor complex DE shown in the paper no longer exists.

```
>>> len(condensed)
2
>>> ABC, FG = condensed
>>> (ABC.subsystem.nodes, ABC.phi)
((n0, n1, n2), 1.9166650000000001)
>>> (FG.subsystem.nodes, FG.phi)
((n5, n6), 0.069445)
```

There are several other functions available for working with complexes; see the documentation for *subsystems()*, *all_complexes()*, *possible_complexes()*, and *complexes()*.

Basic Usage

- *pyphi.examples.basic_network()*
- *pyphi.examples.basic_subsystem()*

Let's make a simple 3-node network and compute its Φ.

To make a network, we need a TPM and (optionally) a connectivity matrix. The TPM can be in more than one form; see the documentation for *pyphi.network*. Here we'll use the 2-dimensional state-by-node form.

```
>>> import pyphi
>>> import numpy as np
>>> tpm = np.array([
...     [0, 0, 0],
...     [0, 0, 1],
...     [1, 0, 1],
...     [1, 0, 0],
...     [1, 1, 0],
...     [1, 1, 1],
...     [1, 1, 1],
...     [1, 1, 0]
... ])
```

The connectivity matrix is a square matrix such that the i, j^{th} entry is 1 if there is a connection from node i to node j, and 0 otherwise.

```
>>> cm = np.array([
...     [0, 0, 1],
...     [1, 0, 1],
...     [1, 1, 0]
... ])
```

Now we construct the network itself with the arguments we just created:

```
>>> network = pyphi.Network(tpm, connectivity_matrix=cm)
```

The next step is to define a subsystem for which we want to evaluate Φ. To make a subsystem, we need the network that it belongs to, the state of that network, and the indices of the subset of nodes which should be included.

The state should be an n-tuple, where n is the number of nodes in the network, and where the i^{th} element is the state of the i^{th} node in the network.

```
>>> state = (1, 0, 0)
```

In this case, we want the Φ of the entire network, so we simply include every node in the network in our subsystem:

```
>>> subsystem = pyphi.Subsystem(network, state, range(network.size))
```

Now we use *pyphi.compute.big_phi()* function to compute the Φ of our subsystem:

```
>>> phi = pyphi.compute.big_phi(subsystem)
>>> phi
2.3125
```

If we want to take a deeper look at the integrated-information-theoretic properties of our network, we can access all the intermediate quantities and structures that are calculated in the course of arriving at a final Φ value by using *pyphi.compute.big_mip()*. This returns a deeply nested object, *BigMip*, that contains data about the subsystem's constellation of concepts, cause and effect repertoires, etc.

```
>>> mip = pyphi.compute.big_mip(subsystem)
```

For instance, we can see that this network has 4 concepts:

```
>>> len(mip.unpartitioned_constellation)
4
```

The documentation for *pyphi.models* contains description of these structures.

Note: Networks can be constructed with an optional set of textual labels for each node:

```
>>> labels = ('A', 'B', 'C')
>>> network = pyphi.Network(tpm, cm, node_labels=labels)
```

These labels must be unique. We can then use these labels when constructing a *Subsystem*:

```
>>> pyphi.Subsystem(network, state, ('B', 'C'))
Subsystem((B, C))
```

Note: The network and subsystem discussed here are returned by the *pyphi.examples.basic_network()* and *pyphi.examples.basic_subsystem()* functions.

Conditional Independence

- *pyphi.examples.cond_depend_tpm()*
- *pyphi.examples.cond_independ_tpm()*

This example explores the assumption of conditional independence, and the behaviour of the program when it is not satisfied.

Every state-by-node TPM corresponds to a unique state-by-state TPM which satisfies the conditional independence assumption. If a state-by-node TPM is given as input for a network, the program assumes that it is from a system with the corresponding conditionally independent state-by-state TPM.

When a state-by-state TPM is given as input for a network, the state-by-state TPM is first converted to a state-by-node TPM. The program then assumes that the system corresponds to the unique conditionally independent representation of the state-by-node TPM. If a non-conditionally independent TPM is given, the analyzed system will not correspond to the original TPM. Note that every deterministic state-by-state TPM will automatically satisfy the conditional independence assumption.

Consider a system of two binary nodes(A and B) which do not change if they have the same value, but flip with probability 50% if they have different values.

We'll load the state-by-state TPM for such a system from the examples module:

```
>>> import pyphi
>>> tpm = pyphi.examples.cond_depend_tpm()
>>> print(tpm)
[[ 1.   0.   0.   0. ]
 [ 0.   0.5  0.5  0. ]
 [ 0.   0.5  0.5  0. ]
 [ 0.   0.   0.   1. ]]
```

This system does not satisfy the conditional independence assumption; given a past state of `(1, 0)`, the current state of node A depends on whether or not B has flipped.

When creating a network, the program will convert this state-by-state TPM to a state-by-node form, and issue a warning if it does not satisfy the assumption:

```
>>> sbn_tpm = pyphi.convert.state_by_state2state_by_node(tpm)
```

"The TPM is not conditionally independent. See the conditional independence example in the documentation for more information on how this is handled."

```
>>> print(sbn_tpm)
[[[ 0.   0. ]
  [ 0.5  0.5]]

 [[ 0.5  0.5]
  [ 1.   1. ]]]
```

The program will continue with the state-by-node TPM, but since it assumes conditional independence, the network will not correspond to the original system.

To see the corresponding conditionally independent TPM, convert the state-by-node TPM back to state-by-state form:

```
>>> sbs_tpm = pyphi.convert.state_by_node2state_by_state(sbn_tpm)
>>> print(sbs_tpm)
[[ 1.    0.    0.    0.  ]
 [ 0.25  0.25  0.25  0.25]
 [ 0.25  0.25  0.25  0.25]
 [ 0.    0.    0.    1.  ]]
```

A system which does not satisfy the conditional independence assumption shows "instantaneous causality." In such situations, there must be additional exogenous variable(s) which explain the dependence.

Consider the above example, but with the addition of a third node (C) which is equally likely to be ON or OFF, and such that when nodes A and B are in different states, they will flip when C is ON, but stay the same when C is OFF.

```
>>> tpm2 = pyphi.examples.cond_independ_tpm()
>>> print(tpm2)
[[ 0.5  0.   0.   0.   0.5  0.   0.   0. ]
 [ 0.   0.5  0.   0.   0.   0.5  0.   0. ]
 [ 0.   0.   0.5  0.   0.   0.   0.5  0. ]
 [ 0.   0.   0.   0.5  0.   0.   0.   0.5]
 [ 0.5  0.   0.   0.   0.5  0.   0.   0. ]
 [ 0.   0.   0.5  0.   0.   0.   0.5  0. ]
 [ 0.   0.5  0.   0.   0.   0.5  0.   0. ]
 [ 0.   0.   0.   0.5  0.   0.   0.   0.5]]
```

The resulting state-by-state TPM now satisfies the conditional independence assumption.

```
>>> sbn_tpm2 = pyphi.convert.state_by_state2state_by_node(tpm2)
>>> print(sbn_tpm2)
[[[[ 0.   0.   0.5]
   [ 0.   0.   0.5]]

  [[ 0.   1.   0.5]
   [ 1.   0.   0.5]]]


 [[[ 1.   0.   0.5]
   [ 0.   1.   0.5]]
```

```
  [[ 1.   1.   0.5]
   [ 1.   1.   0.5]]]]
```

The node indices are `0` and `1` for A and B, and `2` for C:

```
>>> AB = [0, 1]
>>> C = [2]
```

From here, if we marginalize out the node C;

```
>>> tpm2_marginalizeC = pyphi.utils.marginalize_out(C, sbn_tpm2)
```

And then restrict the purview to only nodes A and B;

```
>>> import numpy as np
>>> tpm2_purviewAB = np.squeeze(tpm2_marginalizeC[:,:,:,AB])
```

We get back the original state-by-node TPM from the system with just A and B.

```
>>> np.all(tpm2_purviewAB == sbn_tpm)
True
```

Emergence (coarse-graining and blackboxing)

Coarse-graining

- *pyphi.examples.macro_network()*

We'll use the *macro* module to explore alternate spatial scales of a network. The network under consideration is a 4-node non-deterministic network, available from the *examples* module.

```
>>> import pyphi
>>> network = pyphi.examples.macro_network()
```

The connectivity matrix is all-to-all:

```
>>> network.connectivity_matrix
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

We'll set the state so that nodes are **OFF**.

```
>>> state = (0, 0, 0, 0)
```

At the "micro" spatial scale, we can compute the main complex, and determine the Φ value:

```
>>> main_complex = pyphi.compute.main_complex(network, state)
>>> main_complex.phi
0.113889
```

The question is whether there are other spatial scales which have greater values of Φ. This is accomplished by considering all possible coarse-graining of micro-elements to form macro-elements. A coarse-graining of nodes is any partition of the elements of the micro system. First we'll get a list of all possible coarse-grainings:

```
>>> grains = list(pyphi.macro.all_coarse_grains(network.node_indices))
```

We start by considering the first coarse grain:

```
>>> coarse_grain = grains[0]
>>> coarse_grain
CoarseGrain(partition=((0, 1, 2), (3,)), grouping=(((0, 1, 2), (3,)), ((0,), (1,))))
```

Each *CoarseGrain* specifies two fields: the `partition` of states into macro elements, and the `grouping` of micro-states into macro-states. Let's first look at the partition:

```
>>> coarse_grain.partition
((0, 1, 2), (3,))
```

There are two macro-elements in this partiion: one consists of micro-elements `(0, 1, 2)` and the other is simply micro-element `3`.

We must then determine the relationship between micro-elements and macro-elements. When coarse-graining the system we assume that the resulting macro-elements do not differentiate the different micro-elements. Thus any correspondence between states must be stated solely in terms of the number of micro-elements which are on, and not depend on which micro-elements are on.

For example, consider the macro-element `(0, 1, 2)`. We may say that the macro-element is **ON** if at least one micro-element is on, or if all micro-elements are on; however, we may not say that the macro-element is **ON** if micro-element `1` is on, because this relationship involves identifying specific micro-elements.

The `grouping` attribute of the *CoarseGrain* describes how the state of micro-elements describes the state of macro-elements:

```
>>> grouping = coarse_grain.grouping
>>> grouping
(((0, 1, 2), (3,)), ((0,), (1,)))
```

The grouping consists of two lists, one for each macro-element:

```
>>> grouping[0]
((0, 1, 2), (3,))
```

For the first macro-element, this grouping means that the element will be **OFF** if zero, one or two of its micro-elements are **ON**, and will be **ON** if all three micro-elements are **ON**.

```
>>> grouping[1]
((0,), (1,))
```

For the second macro-element, the grouping means that the element will be **OFF** if its micro-element is **OFF**, and **ON** if its micro-element is **ON**.

One we have selected a partition and grouping for analysis, we can create a mapping between micro-states and macro-states:

```
>>> mapping = coarse_grain.make_mapping()
>>> mapping
array([0, 0, 0, 0, 0, 0, 0, 1, 2, 2, 2, 2, 2, 2, 2, 3])
```

The interpretation of the mapping uses the **LOLI** convention of indexing (see *LOLI: Low-Order bits correspond to Low-Index nodes*).

```
>>> mapping[7]
1
```

This says that micro-state 7 corresponds to macro-state 1:

```
>>> pyphi.convert.loli_index2state(7, 4)
(1, 1, 1, 0)
```

```
>>> pyphi.convert.loli_index2state(1, 2)
(1, 0)
```

In micro-state 7, all three elements corresponding to the first macro-element are **ON**, so that macro-element is **ON**. The micro-element corresponding to the second macro-element is **OFF**, so that macro-element is **OFF**.

The *CoarseGrain* object uses the mapping internally to create a state-by-state TPM for the macro-system corresponding to the selected partition and grouping

```
>>> coarse_grain.macro_tpm(network.tpm)
Traceback (most recent call last):
    ...
pyphi.exceptions.ConditionallyDependentError...
```

However, this macro-TPM does not satisfy the conditional independence assumption, so this particular partition and grouping combination is not a valid coarse-graining of the system. Constructing a *MacroSubsystem* with this coarse-graining will also raise *ConditionallyDependentError*.

Let's consider a different coarse-graining instead.

```
>>> coarse_grain = grains[14]
>>> coarse_grain.partition
((0, 1), (2, 3))
>>> coarse_grain.grouping
(((0, 1), (2,)), ((0, 1), (2,)))
```

```
>>> mapping = coarse_grain.make_mapping()
>>> mapping
array([0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 2, 2, 2, 3])
```

```
>>> coarse_grain.macro_tpm(network.tpm)
array([[[ 0.09,  0.09],
        [ 1.  ,  0.09]],

       [[ 0.09,  1.  ],
        [ 1.  ,  1.  ]]])
```

We can now construct a *MacroSubsystem* using this coarse-graining:

```
>>> macro_subsystem = pyphi.macro.MacroSubsystem(network, state, network.node_indices,
↪ coarse_grain=coarse_grain)
>>> macro_subsystem
MacroSubsystem((n0, n1))
```

We can then consider the integrated information of this macro-network and compare it to the micro-network.

```
>>> macro_mip = pyphi.compute.big_mip(macro_subsystem)
>>> macro_mip.phi
0.597212
```

The integrated information of the macro subsystem ($\Phi = 0.597212$) is greater than the integrated information of the micro system ($\Phi = 0.113889$). We can conclude that a macro-scale is appropriate for this system, but to determine which one, we must check all possible partitions and all possible groupings to find the maximum of integrated information across all scales.

```
>>> M = pyphi.macro.emergence(network, state)
>>> M.emergence
0.483323
>>> M.system
(0, 1, 2, 3)
>>> M.coarse_grain.partition
((0, 1), (2, 3))
>>> M.coarse_grain.grouping
(((0, 1), (2,)), ((0, 1), (2,)))
```

The analysis determines the partition and grouping which results in the maximum value of integrated information, as well as the emergence (increase in Φ) from the micro-scale to the macro-scale.

Blackboxing

- *pyphi.examples.blackbox_network()*

The *macro* module also provides tools for studying the emergence of systems using blackboxing.

```
>>> import pyphi
>>> network = pyphi.examples.blackbox_network()
```

We consider the state where all nodes are off:

```
>>> state = (0, 0, 0, 0, 0, 0)
>>> all_nodes = (0, 1, 2, 3, 4, 5)
```

The system has minimal Φ without blackboxing:

```
>>> subsys = pyphi.Subsystem(network, state, all_nodes)
>>> pyphi.compute.big_phi(subsys)
0.215278
```

We will consider the blackbox system consisting of two blackbox elements, ABC and DEF, where C and F are output elements and AB and DE are hidden within their respective blackboxes.

Blackboxing is done with a *Blackbox* object. As with *CoarseGrain*, we pass it a partition of micro-elements:

```
>>> partition = ((0, 1, 2), (3, 4, 5))
>>> output_indices = (2, 5)
>>> blackbox = pyphi.macro.Blackbox(partition, output_indices)
```

Blackboxes have a few convenience methods. The `hidden_indices` property returns the elements which are hidden within blackboxes:

```
>>> blackbox.hidden_indices
(0, 1, 3, 4)
```

The `micro_indices` property lists all the micro-elements in the box:

```
>>> blackbox.micro_indices
(0, 1, 2, 3, 4, 5)
```

The `macro_indices` property generates a set of indices which index the blackbox macro-elements. Since there are two blackboxes in our example, and each has one output element, there are two macro-indices:

```
>>> blackbox.macro_indices
(0, 1)
```

The `macro_state` method converts a state of the micro elements to the state of the macro-elements. The macro-state of a blackbox system is simply the state of the system's output elements:

```
>>> micro_state = (0, 0, 0, 0, 0, 1)
>>> blackbox.macro_state(micro_state)
(0, 1)
```

Let us also define a time scale over which to perform our analysis:

```
>>> time_scale = 2
```

As in the coarse-graining example, the blackbox and time scale are passed to *MacroSubsystem*:

```
>>> macro_subsystem = pyphi.macro.MacroSubsystem(network, state, all_nodes,␣
↪blackbox=blackbox, time_scale=time_scale)
```

We can now compute Φ for this macro system:

```
>>> pyphi.compute.big_phi(macro_subsystem)
0.638888
```

We find that the macro subsystem has greater integrated information ($\Phi = 0.638888$) than the micro system ($\Phi = 0.215278$)—the system demonstrates emergence.

Magic Cuts

- *pyphi.examples.rule110_network()*
- *pyphi.examples.rule154_network()*

This example explores a system of three fully connected elements A, B and C, which follow the logic of the Rule 110 cellular automaton. The point of this example is to highlight an unexpected behaviour of system cuts: that the minimum information partition of a system can result in new concepts being created.

First let's create the the Rule 110 network, with all nodes **OFF** in the current state.

```
>>> import pyphi
>>> network = pyphi.examples.rule110_network()
>>> state = (0, 0, 0)
```

Next, we want to identify the spatial scale and main complex of the network:

```
>>> macro = pyphi.macro.emergence(network, state)
>>> macro.emergence
-1.35708
```

Since the emergence value is negative, there is no macro scale which has greater integrated information than the original micro scale. We can now analyze the micro scale to determine the main complex of the system:

```
>>> main_complex = pyphi.compute.main_complex(network, state)
>>> subsystem = main_complex.subsystem
```

```
>>> subsystem
Subsystem((n0, n1, n2))
>>> main_complex.phi
1.35708
```

The main complex of the system contains all three nodes of the system, and it has integrated information $\Phi = 1.35708$. Now that we have identified the main complex of the system, we can explore its conceptual structure and the effect of the **MIP**.

```
>>> constellation = main_complex.unpartitioned_constellation
```

There two equalivalent cuts for this system; for concreteness we sever all connections from elements A and B to C.

```
>>> cut = pyphi.models.Cut(severed = (0, 1), intact = (2,))
>>> cut_subsystem = pyphi.Subsystem(network, state, range(network.size), cut)
>>> cut_constellation = pyphi.compute.constellation(cut_subsystem)
```

Lets investigate the concepts in the unpartitioned constellation,

```
>>> [concept.mechanism for concept in constellation]
[(0,), (1,), (2,), (0, 1), (0, 2), (1, 2)]
>>> [concept.phi for concept in constellation]
[0.125, 0.125, 0.125, 0.499999, 0.499999, 0.499999]
>>> sum(_)
1.8749970000000002
```

and also the concepts of the partitioned constellation.

```
>>> [concept.mechanism for concept in cut_constellation]
[(0,), (1,), (2,), (0, 1), (1, 2), (0, 1, 2)]
>>> [concept.phi for concept in cut_constellation]
[0.125, 0.125, 0.125, 0.499999, 0.266666, 0.333333]
>>> sum(_)
1.4749980000000003
```

The unpartitioned constellation includes all possible first and second order concepts, but there is no third order concept. After applying the cut and severing the connections from A and B to C, the third order concept ABC is created and the second order concept AC is destroyed. The overall amount of φ in the system decreases from 1.875 to 1.475.

Lets explore the concept which was created to determine why it does not exist in the unpartitioned constellation and what changed in the partitioned constellation.

```
>>> subsystem = main_complex.subsystem
>>> ABC = subsystem.node_indices
>>> subsystem.cause_info(ABC, ABC)
0.749999
>>> subsystem.effect_info(ABC, ABC)
1.875
```

The mechanism has cause and effect power over the system, so it must be that this power is reducible.

```
>>> mice_cause = subsystem.core_cause(ABC)
>>> mice_cause.phi
0.0
>>> mice_effect = subsystem.core_effect(ABC)
>>> mice_effect.phi
0.625
```

The reason ABC does not exist as a concept is that its cause is reducible. Looking at the TPM of the system, there are no possible states with two of the elements set to **OFF**. This means that knowing two elements are **OFF** is enough to know that the third element must also be **OFF**, and thus the third element can always be cut from the concept without a loss of information. This will be true for any purview, so the cause information is reducible.

```
>>> BC = (1, 2)
>>> A = (0,)
>>> repertoire = subsystem.cause_repertoire(ABC, ABC)
>>> cut_repertoire = subsystem.cause_repertoire(BC, ABC) * subsystem.cause_
↪repertoire(A, ())
>>> pyphi.utils.hamming_emd(repertoire, cut_repertoire)
0.0
```

Next, lets look at the cut subsystem to understand how the new concept comes into existence.

```
>>> ABC = (0, 1, 2)
>>> C = (2,)
>>> AB = (0, 1)
```

The cut applied to the subsystem severs the connections from A and B to C. In this circumstance, knowing A and B do not tell us anything about the state of C, only the past state of C can tell us about the future state of C. Here, `past_tpm[1]` gives us the probability of C being **ON** in the next state, while `past_tpm[0]` would give us the probability of C being **OFF**.

```
>>> C_node = cut_subsystem.indices2nodes(C)[0]
>>> C_node.tpm[1].flatten()
array([ 0.5 ,  0.75])
```

This states that A has a 50% chance of being **ON** in the next state if it currently **OFF**, but a 75% chance of being **ON** in the next state if it is currently **ON**. Thus unlike the unpartitioned case, knowing the current state of C gives us additional information over and above knowing A and B.

```
>>> repertoire = cut_subsystem.cause_repertoire(ABC, ABC)
>>> cut_repertoire = cut_subsystem.cause_repertoire(AB, ABC) * cut_subsystem.cause_
↪repertoire(C, ())
>>> pyphi.utils.hamming_emd(repertoire, cut_repertoire)
0.500001
```

With this partition, the integrated information is $\varphi = 0.5$, but we must check all possible partitions to find the MIP.

```
>>> cut_subsystem.core_cause(ABC).purview
(0, 1, 2)
>>> cut_subsystem.core_cause(ABC).phi
0.333333
```

It turns out that the MIP is

$$\frac{AB}{[]} \times \frac{C}{ABC}$$

and the integrated information of ABC is $\varphi = 1/3$.

Note: In order for a new concept to be created by a cut, there must be a within mechanism connection severed by the cut.

In the previous example, the **MIP** created a new concept, but the amount of φ in the constellation still decreased. This is not always the case. Next we will look at an example of system whoes **MIP** increases the amount of φ. This example is based on a five node network which follows the logic of the Rule 154 cellular automaton. Lets first load the network,

```
>>> network = pyphi.examples.rule154_network()
>>> state = (1, 0, 0, 0, 0)
```

For this example, it is the subsystem consisting of $n_0 n_1 n_4$ that we explore. This is not the main concept of the system, but it serves as a proof of principle regardless.

```
>>> subsystem = pyphi.Subsystem(network, state, (0, 1, 4))
```

Calculating the **MIP** of the system,

```
>>> mip = pyphi.compute.big_mip(subsystem)
>>> mip.phi
0.217829
>>> mip.cut
Cut (0, 4) --//--> (1,)
```

This subsystem has a Φ value of 0.15533, and the **MIP** cuts the connections from $n_0 n_4$ to n_1. Investigating the concepts in both the partitioned and unpartitioned constellations,

```
>>> unpartitioned_constellation = mip.unpartitioned_constellation
>>> [concept.mechanism for concept in unpartitioned_constellation]
[(0,), (1,), (0, 1)]
>>> [concept.phi for concept in unpartitioned_constellation]
[0.25, 0.166667, 0.178572]
>>> sum([concept.phi for concept in unpartitioned_constellation])
0.5952390000000001
```

The unpartitioned constellation has mechanisms n_0, n_1 and $n_0 n_1$ with $\sum \varphi = 0.595239$.

```
>>> partitioned_constellation = mip.partitioned_constellation
>>> [concept.mechanism for concept in partitioned_constellation]
[(0, 1), (0,), (1,)]
>>> [concept.phi for concept in partitioned_constellation]
[0.214286, 0.25, 0.166667]
>>> sum([concept.phi for concept in partitioned_constellation])
0.630953
```

The unpartitioned constellation has mechanisms n_0, n_1 and $n_0 n_1$ with $\sum \varphi = 0.630953$. There are the same number of concepts in both constellations, over the same mechanisms; however, the partitioned constellation has a greater φ value for the concept $n_0 n_1$, resulting in an overall greater $\sum \varphi$ for the **MIP** constellation.

Although situations described above are rare, they do occur, so one must be careful when analyzing the integrated information of physical systems not to dismiss the possibility of partitions creating new concepts or increasing the amount of φ; otherwise, an incorrect main complex may be identified.

Residue

- *pyphi.examples.residue_network()*
- *pyphi.examples.residue_subsystem()*

This example describes a system containing two **AND** nodes, A and B, with a single overlapping input node.

First let's create the subsystem corresponding to the residue network, with all nodes off in the current and past states.

```
>>> import pyphi
>>> subsystem = pyphi.examples.residue_subsystem()
```

Next, we can define the mechanisms of interest. Mechanisms and purviews are represented by tuples of node indices in the network:

```
>>> A = (0,)
>>> B = (1,)
>>> AB = (0, 1)
```

And the possible past purviews that we're interested in:

```
>>> CD = (2, 3)
>>> DE = (3, 4)
>>> CDE = (2, 3, 4)
```

We can then evaluate the cause information for each of the mechanisms over the past purview CDE.

```
>>> subsystem.cause_info(A, CDE)
0.333332
```

```
>>> subsystem.cause_info(B, CDE)
0.333332
```

```
>>> subsystem.cause_info(AB, CDE)
0.5
```

The composite mechanism AB has greater cause information than either of the individual mechanisms. This contradicts the idea that AB should exist minimally in this system.

Instead, we can quantify existence as the irreducible cause information of a mechanism. The **MIP** of a mechanism is the partition of mechanism and purview which makes the least difference to the cause repertoire (see the documentation for the *Mip* object). The irreducible cause information is the distance between the unpartitioned and partitioned repertoires.

To calculate the MIP structure of mechanism AB:

```
>>> mip_AB = subsystem.mip_past(AB, CDE)
```

We can then determine what the specific partition is.

```
>>> mip_AB.partition
[]   0,1
-- X ---
2    3,4
```

The labels `(n0, n1, n2, n3, n4)` correspond to nodes A, B, C, D, E respectively. Thus the MIP is $\frac{AB}{DE} \times \frac{[]}{C}$, where $[]$ denotes the empty mechanism.

The partitioned repertoire of the MIP can also be retrieved:

```
>>> mip_AB.partitioned_repertoire
array([[[[[ 0.2,  0.2],
          [ 0.1,  0. ]],

         [[ 0.2,  0.2],
          [ 0.1,  0. ]]]]])
```

And we can then calculate the irreducible cause information as the difference between partitioned and unpartitioned repertoires.

```
>>> mip_AB.phi
0.1
```

One counterintuitive result that merits discussion is that since irreducible cause information is what defines existence, we must also evaluate the irreducible cause information of the mechanisms A and B.

The mechanism A over the purview CDE is completely reducible to $\frac{A}{CD} \times \frac{[]}{E}$ because E has no effect on A, so it has zero φ.

```
>>> subsystem.mip_past(A, CDE).phi
0.0
>>> subsystem.mip_past(A, CDE).partition
[]    0
-- X ---
4     2,3
```

Instead, we should evaluate A over the purview CD.

```
>>> mip_A = subsystem.mip_past(A, CD)
```

In this case, there is a well defined MIP

```
>>> mip_A.partition
[]   0
-- X -
2    3
```

which is $\frac{[]}{C} \times \frac{A}{D}$. It has partitioned repertoire

```
>>> mip_A.partitioned_repertoire
array([[[[[ 0.33333333],
          [ 0.16666667]],

         [[ 0.33333333],
          [ 0.16666667]]]]])
```

and irreducible cause information

```
>>> mip_A.phi
0.166667
```

A similar result holds for B. Thus the mechanisms A and B exist at levels of $\varphi = \frac{1}{6}$, while the higher-order mechanism AB exists only as the residual of causes, at a level of $\varphi = \frac{1}{10}$.

XOR Network

- *pyphi.examples.xor_network()*
- *pyphi.examples.xor_subsystem()*

This example describes a system of three fully connected **XOR** nodes, n_0, n_1 and n_2 (no self-connections).

First let's create the XOR network:

```
>>> import pyphi
>>> network = pyphi.examples.xor_network()
```

We'll consider the state with all nodes **OFF**.

```
>>> state = (0, 0, 0)
```

Existence is a top-down process; the whole is more important than its parts. The first step is to confirm the existence of the whole, by finding the main complex of the network:

```
>>> main_complex = pyphi.compute.main_complex(network, state)
```

The main complex exists ($\Phi > 0$),

```
>>> main_complex.phi
1.874999
```

and it consists of the entire network:

```
>>> main_complex.subsystem
Subsystem((n0, n1, n2))
```

Knowing what exists at the system level, we can now investigate the existence of concepts within the complex.

```
>>> constellation = main_complex.unpartitioned_constellation
>>> len(constellation)
3
>>> [concept.mechanism for concept in constellation]
[(0, 1), (0, 2), (1, 2)]
```

There are three concepts in the constellation. They are all the possible second order mechanisms: $n_0 n_1$, $n_0 n_2$ and $n_1 n_2$.

Focusing on the concept specified by mechanism $n_0 n_1$, we investigate existence, and the irreducible cause and effect. Based on the symmetry of the network, the results will be similar for the other second order mechanisms.

```
>>> concept = constellation[0]
>>> concept.mechanism
(0, 1)
>>> concept.phi
0.5
```

The concept has $\varphi = \frac{1}{2}$.

```
>>> concept.cause.purview
(0, 1, 2)
>>> concept.cause.repertoire
array([[[ 0.5,  0. ],
        [ 0. ,  0. ]],

       [[ 0. ,  0. ],
        [ 0. ,  0.5]]])
```

So we see that the cause purview of this mechanism is the whole system $n_0 n_1 n_2$, and that the repertoire shows a 0.5 of probability the past state being `(0, 0, 0)` and the same for `(1, 1, 1)`:

```
>>> concept.cause.repertoire[(0, 0, 0)]
0.5
>>> concept.cause.repertoire[(1, 1, 1)]
0.5
```

This tells us that knowing both n_0 and n_1 are currently **OFF** means that the past state of the system was either all **OFF** or all **ON** with equal probability.

For any reduced purview, we would still have the same information about the elements in the purview (either all **ON** or all **OFF**), but we would lose the information about the elements outside the purview.

```
>>> concept.effect.purview
(2,)
>>> concept.effect.repertoire
array([[[ 1.,  0.]]])
```

The effect purview of this concept is the node n_2. The mechanism $n_0 n_1$ is able to completely specify the next state of n_2. Since both nodes are **OFF**, the next state of n_2 will be **OFF**.

The mechanism $n_0 n_1$ does not provide any information about the next state of either n_0 or n_1, because the relationship depends on the value of n_2. That is, the next state of n_0 (or n_1) may be either **ON** or **OFF**, depending on the value of n_2. Any purview larger than n_2 would be reducible by pruning away the additional elements.

Main Complex: $n_0 n_1 n_2$ with $\Phi = 1.875$			
Mechanism	φ	**Cause Purview**	**Effect Purview**
$n_0 n_1$	0.5	$n_0 n_1 n_2$	n_2
$n_0 n_2$	0.5	$n_0 n_1 n_2$	n_1
$n_1 n_2$	0.5	$n_0 n_1 n_2$	n_0

An analysis of the *intrinsic existence* of this system reveals that the main complex of the system is the entire network of **XOR** nodes. Furthermore, the concepts which exist within the complex are those specified by the second-order mechanisms $n_0 n_1$, $n_0 n_2$, and $n_1 n_2$.

To understand the notion of intrinsic existence, in addition to determining what exists for the system, it is useful to consider also what does not exist.

Specifically, it may be surprising that none of the first order mechanisms n_0, n_1 or n_2 exist. This physical system of **XOR** gates is sitting on the table in front of me; I can touch the individual elements of the system, so how can it be that they do not exist?

That sort of existence is what we term *extrinsic existence*. The **XOR** gates exist for me as an observer, external to the system. I am able to manipulate them, and observe their causes and effects, but the question that matters for *intrinsic* existence is, do they have irreducible causes and effects within the system? There are two reasons a mechanism may have no irreducible cause-effect power: either the cause-effect power is completely reducible, or there was no cause-effect power to begin with. In the case of elementary mechanisms, it must be the latter.

To see this, again due to symmetry of the system, we will focus only on the mechanism n_0.

```
>>> subsystem = pyphi.examples.xor_subsystem()
>>> n0 = (0,)
>>> n0n1n2 = (0, 1, 2)
```

In order to exist, a mechanism must have irreducible cause and effect power within the system.

```
>>> subsystem.cause_info(n0, n0n1n2)
0.5
>>> subsystem.effect_info(n0, n0n1n2)
0.0
```

The mechanism has no effect power over the entire subsystem, so it cannot have effect power over any purview within the subsystem. Furthermore, if a mechanism has no effect power, it certainly has no irreducible effect power. The first-order mechanisms of this system do not exist intrinsically, because they have no effect power (having causal power is not enough).

To see why this is true, consider the effect of n_0. There is no self-loop, so n_0 can have no effect on itself. Without knowing the current state of n_0, in the next state n_1 could be either **ON** or **OFF**. If we know that the current state of n_0 is **ON**, then n_1 could still be either **ON** or **OFF**, depending on the state of n_2. Thus, on its own, the current state of

n_0 does not provide any information about the next state of n_1. A similar result holds for the effect of n_0 on n_2. Since n_0 has no effect power over any element of the system, it does not exist from the intrinsic perspective.

To complete the discussion, we can also investigate the potential third order mechanism $n_0 n_1 n_2$. Consider the cause information over the purview $n_0 n_1 n_2$:

```
>>> subsystem.cause_info(n0n1n2, n0n1n2)
0.749999
```

Since the mechanism has nonzero cause information, it has causal power over the system—but is it irreducible?

```
>>> mip = subsystem.mip_past(n0n1n2, n0n1n2)
>>> mip.phi
0.0
>>> mip.partition
0     1,2
-- X -----
[]   0,1,2
```

The mechanism has $ci = 0.75$, but it is completely reducible ($\varphi = 0$) to the partition

$$\frac{n_0}{[\,]} \times \frac{n_1 n_2}{n_0 n_1 n_2}$$

This result can be understood as follows: knowing that n_1 and n_2 are **OFF** in the current state is sufficient to know that n_0, n_1, and n_2 were all **OFF** in the past state; there is no additional information gained by knowing that n_0 is currently **OFF**.

Similarly for any other potential purview, the current state of n_1 and n_2 being `(0, 0)` is always enough to fully specify the previous state, so the mechanism is reducible for all possible purviews, and hence does not exist.

CHAPTER 2

Configuration

PyPhi can be configured in various important ways; see the *config* module for details.

Configuration

The configuration is loaded upon import from a YAML file in the directory where PyPhi is run: `pyphi_config.yml`. If no file is found, the default configuration is used.

The various options are listed here with their defaults

```
>>> import pyphi
>>> defaults = pyphi.config.DEFAULTS
```

It is also possible to manually load a YAML configuration file within your script:

```
>>> pyphi.config.load_config_file('pyphi_config.yml')
```

Or load a dictionary of configuration values:

```
>>> pyphi.config.load_config_dict({'SOME_CONFIG': 'value'})
```

Theoretical approximations

This section deals with assumptions that speed up computation at the cost of theoretical accuracy.

- `pyphi.config.ASSUME_CUTS_CANNOT_CREATE_NEW_CONCEPTS`: In certain cases, making a cut can actually cause a previously reducible concept to become a proper, irreducible concept. Assuming this can never happen can increase performance significantly, however the obtained results are not strictly accurate.

  ```
  >>> defaults['ASSUME_CUTS_CANNOT_CREATE_NEW_CONCEPTS']
  False
  ```

- `pyphi.config.CUT_ONE_APPROXIMATION`: When determining the MIP for Φ, this restricts the set of system cuts that are considered to only those that cut the inputs or outputs of a single node. This restricted set of cuts scales linearly with the size of the system; the full set of all possible bipartitions scales exponentially. This approximation is more likely to give theoretically accurate results with modular, sparsely-connected, or homogeneous networks.

```
>>> defaults['CUT_ONE_APPROXIMATION']
False
```

- `pyphi.config.MEASURE`: The measure to use when computing distances between repertoires and concepts. The default is `EMD`; the Earth Movers's Distance. `KLD` is the Kullback-Leibler Divergence. If `L1` is chosen, the `L1` distance is initially used instead of the EMD when computing MIPs but, if a mechanism and purview are found to be irreducible, the φ value of the MIP is recalculated using the EMD.

```
>>> defaults['MEASURE']
'EMD'
```

System resources

These settings control how much processing power and memory is available for PyPhi to use. The default values may not be appropriate for your use-case or machine, so **please check these settings before running anything**. Otherwise, there is a risk that simulations might crash (potentially after running for a long time!), resulting in data loss.

- `pyphi.config.PARALLEL_CONCEPT_EVALUATION`: Control whether concepts are evaluated in parallel when computing constellations.

```
>>> defaults['PARALLEL_CONCEPT_EVALUATION']
False
```

- `pyphi.config.PARALLEL_CUT_EVALUATION`: Control whether system cuts are evaluated in parallel, which requires more memory. If cuts are evaluated sequentially, only two *BigMip* instances need to be in memory at once.

```
>>> defaults['PARALLEL_CUT_EVALUATION']
True
```

> **Warning:** `PARALLEL_CONCEPT_EVALUATION` and `PARALLEL_CUT_EVALUATION` should not both be set to `True`. Enabling both parallelization modes will slow down computations. If you are doing Φ-computations (with `big_mip`, `main_complex`, etc.) `PARALLEL_CUT_EVALUATION` will be fastest. Use `PARALLEL_CONCEPT_EVALUATION` if you are only computing constellations.

- `pyphi.config.NUMBER_OF_CORES`: Control the number of CPU cores used to evaluate unidirectional cuts. Negative numbers count backwards from the total number of available cores, with `-1` meaning "use all available cores."

```
>>> defaults['NUMBER_OF_CORES']
-1
```

- `pyphi.config.MAXIMUM_CACHE_MEMORY_PERCENTAGE`: PyPhi employs several in-memory caches to speed up computation. However, these can quickly use a lot of memory for large networks or large numbers of them; to avoid thrashing, this options limits the percentage of a system's RAM that the caches can collectively use.

```
>>> defaults['MAXIMUM_CACHE_MEMORY_PERCENTAGE']
50
```

Caching

PyPhi is equipped with a transparent caching system for *BigMip* objects which stores them as they are computed to avoid having to recompute them later. This makes it easy to play around interactively with the program, or to accumulate results with minimal effort. For larger projects, however, it is recommended that you manage the results explicitly, rather than relying on the cache. For this reason it is disabled by default.

- pyphi.config.CACHE_BIGMIPS: Control whether *BigMip* objects are cached and automatically retreived.

```
>>> defaults['CACHE_BIGMIPS']
False
```

- pyphi.config.CACHE_POTENTIAL_PURVIEWS: Controls whether the potential purviews of mechanisms of a network are cached. Caching speeds up computations by not recomputing expensive reducibility checks, but uses additional memory.

```
>>> defaults['CACHE_POTENTIAL_PURVIEWS']
True
```

- pyphi.config.CACHING_BACKEND: Control whether precomputed results are stored and read from a database or from a local filesystem-based cache in the current directory. Set this to 'fs' for the filesystem, 'db' for the database. Caching results on the filesystem is the easiest to use but least robust caching system. Caching results in a database is more robust and allows for caching individual concepts, but requires installing MongoDB.

```
>>> defaults['CACHING_BACKEND']
'fs'
```

- pyphi.config.FS_CACHE_VERBOSITY: Control how much caching information is printed. Takes a value between 0 and 11. Note that printing during a loop iteration can slow down the loop considerably.

```
>>> defaults['FS_CACHE_VERBOSITY']
0
```

- pyphi.config.FS_CACHE_DIRECTORY: If the caching backend is set to use the filesystem, the cache will be stored in this directory. This directory can be copied and moved around if you want to reuse results _e.g._ on a another computer, but it must be in the same directory from which PyPhi is being run.

```
>>> defaults['FS_CACHE_DIRECTORY']
'__pyphi_cache__'
```

- pyphi.config.MONGODB_CONFIG: Set the configuration for the MongoDB database backend. This only has an effect if the caching backend is set to use the database.

```
>>> defaults['MONGODB_CONFIG']['host']
'localhost'
>>> defaults['MONGODB_CONFIG']['port']
27017
>>> defaults['MONGODB_CONFIG']['database_name']
'pyphi'
```

```
>>> defaults['MONGODB_CONFIG']['collection_name']
'cache'
```

- `pyphi.config.REDIS_CACHE`: Specifies whether to use Redis to cache Mice.

```
>>> defaults['REDIS_CACHE']
False
```

- **`pyphi.config.REDIS_CONFIG`: Configure the Redis database backend. These** are the defaults in the provided `redis.conf` file.

```
>>> defaults['REDIS_CONFIG']['host']
'localhost'
>>> defaults['REDIS_CONFIG']['port']
6379
```

Logging

These settings control how PyPhi handles log messages. Logs can be written to standard output, a file, both, or none. If these simple default controls are not flexible enough for you, you can override the entire logging configuration. See the documentation on Python's logger for more information.

- `pyphi.config.LOG_STDOUT_LEVEL`: Controls the level of log messages written to standard output. Can be one of `'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`, or `None`. `DEBUG` is the least restrictive level and will show the most log messages. `CRITICAL` is the most restrictive level and will only display information about unrecoverable errors. If set to `None`, logging to standard output will be disabled entirely.

```
>>> defaults['LOG_STDOUT_LEVEL']
'WARNING'
```

- `pyphi.config.LOG_FILE_LEVEL: Controls the level of log messages written to the log file. This option has the same possible values as ``LOG_STDOUT_LEVEL`.

```
>>> defaults['LOG_FILE_LEVEL']
'INFO'
```

- `pyphi.config.LOG_FILE`: Control the name of the logfile.

```
>>> defaults['LOG_FILE']
'pyphi.log'
```

- `pyphi.config.LOG_CONFIG_ON_IMPORT`: Controls whether the current configuration is printed when PyPhi is imported.

```
>>> defaults['LOG_CONFIG_ON_IMPORT']
True
```

Numerical precision

- `pyphi.config.PRECISION`: Computations in PyPhi rely on finding the Earth Mover's Distance. This is done via an external C++ library that uses flow-optimization to find a good approximation of the EMD. Consequently, systems with zero Φ will sometimes be computed to have a small but non-zero amount. This setting controls the number of decimal places to which PyPhi will consider EMD calculations accurate. Values

of Φ lower than `10e-PRECISION` will be considered insignificant and treated as zero. The default value is about as accurate as the EMD computations get.

```
>>> defaults['PRECISION']
6
```

Miscellaneous

- `pyphi.config.VALIDATE_SUBSYSTEM_STATES`: Control whether PyPhi checks if the subsystems's state is possible (reachable from some past state), given the subsystem's TPM (**which is conditioned on background conditions**). If this is turned off, then **calculated Φ values may not be valid**, since they may be associated with a subsystem that could never be in the given state.

```
>>> defaults['VALIDATE_SUBSYSTEM_STATES']
True
```

- `pyphi.config.SINGLE_NODES_WITH_SELFLOOPS_HAVE_PHI`: If set to `True`, this defines the Phi value of subsystems containing only a single node with a self-loop to be `0.5`. If set to False, their Φ will be actually be computed (to be zero, in this implementation).

```
>>> defaults['SINGLE_NODES_WITH_SELFLOOPS_HAVE_PHI']
False
```

- `pyphi.config.REPR_VERBOSITY`: Controls the verbosity of `__repr__` methods on PyPhi objects. Can be set to `0`, `1`, or `2`. If set to `1`, calling `repr` on PyPhi objects will return pretty-formatted and legible strings, excluding repertoires. If set to `2`, `repr` calls also include repertoires.

  Although this breaks the convention that `__repr__` methods should return a representation which can reconstruct the object, readable representations are convenient since the Python REPL calls `repr` to represent all objects in the shell and PyPhi is often used interactively with the REPL. If set to `0`, `repr` returns more traditional object representations.

```
>>> defaults['REPR_VERBOSITY']
2
```

- `pyphi.config.PARTITION_MECHANISMS`: If `True`, φ-MIP computations will only consider bipartitions that strictly partition the mechanism. That is, for the mechanism `(A, B)` and purview `(B, C, D)` the partition

```
AB   []
-- X --
B    CD
```

  is not considered, but

```
A    B
-- X --
B    CD
```

  is. The following is also valid:

```
AB   []
-- X ---
[]   BCD
```

  In addition, this option introduces wedge tripartitions of the form

```
A    B   []
-- X - X --
B    C   D
```

where the mechanism in the third part is always empty.

Finally, in the case of a φ-tie when computing MICE, this setting choses the MIP with smallest purview instead the largest (which is the default behavior.)

```
>>> defaults['PARTITION_MECHANISMS']
False
```

- `pyphi.config.PARTITION_MECHANISMS`: If `True`, φ-MIP computations will only consider bipartitions that strictly partition the mechanism. That is, for the mechanism `(A, B)` and purview `(B, C, D)` the partition

```
AB   []
-- X --
B    CD
```

is not considered, but

```
A    B
-- X --
B    CD
```

is. The following is also valid:

```
AB   []
-- X ---
[]   BCD
```

Additionally, in the case of a φ-tie when computing MICE, this setting choses the MIP with smallest purview instead the largest (which is the default behavior.)

```
>>> defaults['PARTITION_MECHANISMS']
False
```

`pyphi.config.`**`load_config_dict`**(*config*)
Load configuration values.

Parameters **`config`** (*dict*) – The dict of config to load.

`pyphi.config.`**`load_config_file`**(*filename*)
Load config from a YAML file.

`pyphi.config.`**`load_config_default`**()
Load default config values.

`pyphi.config.`**`get_config_string`**()
Return a string representation of the currently loaded configuration.

`pyphi.config.`**`print_config`**()
Print the current configuration.

`pyphi.config.`**`configure_logging`**()
Configure PyPhi logging based on the loaded configuration.

Note: if PyPhi config options that control logging are changed after they are loaded (eg. in testing), the Python logging configuration will stay the same unless you manually reconfigure the logging by calling this function.

TODO: call this in *config.override*?

class pyphi.config.**override**(***new_conf*)

Decorator and context manager to override config values.

The initial configuration values are reset after the decorated function returns or the context manager completes it block, even if the function or block raises an exception. This is intended to be used by testcases which require specific configuration values.

Example

```
>>> from pyphi import config
>>>
>>> @config.override(PRECISION=20000)
... def test_something():
...     assert config.PRECISION == 20000
...
>>> test_something()
>>> with config.override(PRECISION=100):
...     assert config.PRECISION == 100
...
```

__enter__()

Save original config values; override with new ones.

__exit__(**exc*)

Reset config to initial values; reraise any exceptions.

CHAPTER 3

Conventions

PyPhi uses some conventions for TPM and connectivity matrix formats. These are important to keep in mind when setting up networks.

Conventions

Connectivity Matrices

Throughout PyPhi, if `CM` is a connectivity matrix, then $CM_{i,j} = 1$ means that node i is connected to node j.

LOLI: Low-Order bits correspond to Low-Index nodes

There are several ways to write down a TPM. With both state-by-state and state-by-node TPMs, one is confronted with a choice about which rows correspond to which states. In state-by-state TPMs, this choice must also be made for the columns.

Either the first node changes state every other row (**LOLI**):

A, B	A	B
0, 0	0.1	0.2
1, 0	0.3	0.4
0, 1	0.5	0.6
1, 1	0.7	0.8

Or the last node does (**HOLI**):

A, B	A	B
0, 0	0.1	0.2
0, 1	0.5	0.6
1, 0	0.3	0.4
1, 1	0.7	0.8

Note that the index i of a row in a TPM encodes a network state: convert the index to binary, and each bit gives the state of a node. The question is, which node?

Throughout PyPhi, we always choose the first convention—the state of the first node (the one with the lowest index) varies the fastest. So, the lowest-order bit—the one's place—gives the state of the lowest-index node.

We call this convention the **LOLI convention**: Low Order bits correspond to Low Index nodes. The other convention, where the highest-index node varies the fastest, is similarly called **HOLI**.

Note: The rationale for this choice of convention is that the **LOLI** mapping is stable under changes in the number of nodes, in the sense that the same bit always corresponds to the same node index. The **HOLI** mapping does not have this property.

Note: This applies to only situations where decimal indices are encoding states. Whenever a network state is represented as a list or tuple, we use the only sensible convention: the i^{th} element gives the state of the i^{th} node.

Note: There are various conversion functions available for converting between TPMs, states, and indices using different conventions: see the *pyphi.convert* module.

CHAPTER 4

API Reference

API Reference

PyPhi API documentation, autogenerated from the source code.

`actual`

Methods for computing actual causation of subsystems and mechanisms.

class `pyphi.actual.`**`Context`**(*network*, *before_state*, *after_state*, *cause_indices*, *effect_indices*, *cut=None*)

A set of nodes in a network, with state transitions.

A *`Context`* contains two *`Subsystem`* objects - one representing the system at time $t - 1$ used to compute effect coefficients, and another representing the system at time t which is used to compute cause coefficients. These subsystems are accessed with the `effect_system` and `cause_system` attributes, and are mapped to the causal directions via the `system` attribute.

Parameters

- **`network`** (`Network`) – The network the subsystem belongs to.
- **`before_state`** (*`tuple[int]`*) – The state of the network at time $t - 1$.
- **`after_state`** (*`tuple[int]`*) – The state of the network at time t.
- **`cause_indices`** (*`tuple[int] or tuple[str]`*) – Indices of nodes in the cause system. (TODO: clarify)
- **`effect_indices`** (*`tuple[int] or tuple[str]`*) – Indices of nodes in the effect system. (TODO: clarify)

`node_indices`

tuple[int] – The indices of the nodes in the system.

`network`

Network – The network the system belongs to.

before_state
tuple[int] – The state of the network at time $t-1$.

after_state
tuple[int] – The state of the network at time t.

effect_system
Subsystem – The system in `before_state` used to compute effect repertoires and coefficients.

cause_system
Subsystem – The system in `after_state` used to compute cause repertoires and coefficients.

cause_system
Subsystem

system
dict – A dictionary mapping causal directions to the system used to compute repertoires in that direction.

cut
ActualCut – The cut that has been applied to this context.

Note: During initialization, both the cause and effect systems are conditioned on the `before_state` as the background state. After conditioning the `effect_system` is then properly reset to `after_state`.

---

to_json()

apply_cut(*cut*)
Return a cut version of this context.

cause_repertoire(*mechanism*, *purview*)

effect_repertoire(*mechanism*, *purview*)

unconstrained_cause_repertoire(*purview*)

unconstrained_effect_repertoire(*purview*)

state_probability(*direction*, *repertoire*, *purview*)
The dimensions of the repertoire that correspond to the fixed nodes are collapsed onto their state. All other dimension should be singular already (repertoire size and fixed_nodes need to match), and thus should receive 0 as the conditioning index. A single probability is returned.

probability(*direction*, *mechanism*, *purview*)
Probability that the purview is in it's current state given the state of the mechanism.

unconstrained_probability(*direction*, *purview*)
Unconstrained probability of the purview.

purview_state(*direction*)
The state of the purview when we are computing coefficients in `direction`.

For example, if we are computing the cause coefficient of a mechanism in `after_state`, the direction is‘‘PAST‘‘ and the `purview_state` is ‘‘before_state.

mechanism_state(*direction*)
The state of the mechanism when we are computing coefficients in `direction`.

cause_coefficient(*mechanism*, *purview*, *norm=True*)
Return the cause coefficient for a mechanism in a state over a purview in the actual past state

effect_coefficient(*mechanism*, *purview*, *norm=True*)
Return the effect coefficient for a mechanism in a state over a purview in the actual future state

partitioned_repertoire(*direction*, *partition*)
Compute the repertoire over the partition in the given direction.

partitioned_probability(*direction*, *partition*)
Compute the probability of the mechanism over the purview in the partition.

find_mip(*direction*, *mechanism*, *purview*, *norm=True*, *allow_neg=False*)
Find the coefficient minimum information partition for a mechanism over a purview.

Parameters

- **direction** (*str*) – DIRECTIONS[PAST] or DIRECTIONS[FUTURE]
- **mechanism** (*tuple[int]*) – A mechanism.
- **purview** (*tuple[int]*) – A purview.

Keyword Arguments

- **norm** (*boolean*) – If true, probabilities will be normalized.
- **allow_neg** (*boolean*) – If true, alpha is allowed to be negative. Otherwise, negative values of alpha will be treated as if they were 0.

Returns *AcMip* – The found MIP.

find_occurence(*direction*, *mechanism*, *purviews=False*, *norm=True*, *allow_neg=False*)
Return the maximally irreducible cause or effect coefficient for a mechanism.

Parameters

- **direction** (*str*) – The temporal direction, specifying cause or effect.
- **mechanism** (*tuple[int]*) – The mechanism to be tested for irreducibility.

Keyword Arguments **purviews** (*tuple[int]*) – Optionally restrict the possible purviews to a subset of the subsystem. This may be useful for _e.g._ finding only concepts that are "about" a certain subset of nodes.

Returns *Occurence* – The maximally-irreducible actual cause or effect.

Note: Strictly speaking, the Occurence is a pair of coefficients: the actual cause and actual effect of a mechanism. Here, we return only information corresponding to one direction, DIRECTIONS[PAST] or DIRECTIONS[FUTURE], i.e., we return an actual cause or actual effect coefficient, not the pair of them.

---

find_mice(**args*, ***kwargs*)
Backwards-compatible alias for *find_occurence*.

pyphi.actual.**nice_ac_composition**(*account*)

pyphi.actual.**multiple_states_nice_ac_composition**(*network*, *transitions*, *cause_indices*, *effect_indices*, *mechanisms=False*, *purviews=False*, *norm=True*, *allow_neg=False*)
Print a nice composition for multiple pairs of states Args: As above

transitions (list(2 state tuples)): The first is past the second current. For 'past' current belongs to subsystem and past is the second state. Vice versa for "future"

pyphi.actual.**directed_account**(*context*, *direction*, *mechanisms=False*, *purviews=False*, *norm=True*, *allow_neg=False*)
Set of all Occurence of the specified direction

pyphi.actual.**account**(*context*, *direction*)

pyphi.actual.**account_distance**(*A1*, *A2*)
Return the distance between two accounts. Here that is just the difference in sum(alpha)

Parameters

- **A1** (Account) – The first account.
- **A2** (Account) – The second account

Returns *float* – The distance between the two accounts.

pyphi.actual.**big_acmip**(*context*, *direction=None*)
Return the minimal information partition of a context in a specific direction.

Parameters **context** (Context) – The candidate system.

Returns

AcBigMip –

A nested structure containing all the data from the intermediate calculations. The top level contains the basic MIP information for the given subsystem.

pyphi.actual.**contexts**(*network*, *before_state*, *after_state*)
Return a generator of all **possible** contexts of a network.

pyphi.actual.**nexus**(*network*, *before_state*, *after_state*, *direction=None*)
Return a generator for all irreducible nexus of the network. Direction options are past, future, bidirectional.

pyphi.actual.**causal_nexus**(*network*, *before_state*, *after_state*, *direction=None*)
Return the causal nexus of the network.

pyphi.actual.**nice_true_constellation**(*true_constellation*)

pyphi.actual.**events**(*network*, *past_state*, *current_state*, *future_state*, *nodes*, *mechanisms=False*)
Find all events (mechanisms with actual causes and actual effects.

pyphi.actual.**true_constellation**(*subsystem*, *past_state*, *future_state*)
Set of all sets of elements that have true causes and true effects. Note: Since the true constellation is always about the full system, the background conditions don't matter and the subsystem should be conditioned on the current state.

pyphi.actual.**true_events**(*network*, *past_state*, *current_state*, *future_state*, *indices=None*, *main_complex=None*)
Set of all mechanisms that have true causes and true effects within the complex.

Parameters

- **network** (Network) –
- **past_state** (*tuple[int]*) – The state of the network at t-1
- **current_state** (*tuple[int]*) – The state of the network at t
- **future_state** (*tuple[int]*) – The state of the network at t+1

Optional Args: indices (tuple[int]): The indices of the main complex main_complex (AcBigMip): The main complex. If `main_complex` is given

then `indices` is ignored.

Returns *tuple[Event]* – List of true events in the main complex

pyphi.actual.**extrinsic_events**(*network*, *past_state*, *current_state*, *future_state*, *indices=None*, *main_complex=None*)

Set of all mechanisms that are in the main complex but which have true causes and effects within the entire network.

Parameters

- **network** (Network) –
- **past_state** (*tuple[int]*) – The state of the network at t-1
- **current_state** (*tuple[int]*) – The state of the network at t
- **future_state** (*tuple[int]*) – The state of the network at t+1

Optional Args: indices (tuple[int]): The indices of the main complex main_complex (AcBigMip): The main complex. If `main_complex` is given

then `indices` is ignored.

Returns *tuple(actions)* – List of true events in the main complex

compute

Maintains backwards compatability with the old `compute` API.

See *compute.concept* and *compute.big_phi* for documentation.

pyphi.compute.**concept**
Alias for *concept.concept()*.

pyphi.compute.**conceptual_information**
Alias for *concept.conceptual_information()*.

pyphi.compute.**constellation**
Alias for *concept.constellation()*.

pyphi.compute.**concept_distance**
Alias for *distance.concept_distance()*.

pyphi.compute.**constellation_distance**
Alias for *distance.constellation_distance()*.

pyphi.compute.**all_complexes**
Alias for *big_phi.all_complexes()*.

pyphi.compute.**big_mip**
Alias for *big_phi.big_mip()*.

pyphi.compute.**big_phi**
Alias for *big_phi.big_phi()*.

pyphi.compute.**complexes**
Alias for *big_phi.complexes()*.

pyphi.compute.**condensed**
Alias for *big_phi.condensed()*.

pyphi.compute.**evaluate_cut**
Alias for *big_phi.evaluate_cut()*.

pyphi.compute.**main_complex**
Alias for *big_phi.main_complex()*.

pyphi.compute.**possible_complexes**
Alias for *big_phi.possible_complexes()*.

pyphi.compute.**subsystems**
Alias for *big_phi.subsystems()*.

compute.big_phi

Methods for computing concepts, constellations, and integrated information of subsystems.

pyphi.compute.big_phi.**evaluate_cut**(*uncut_subsystem*, *cut*, *unpartitioned_constellation*)
Find the *BigMip* for a given cut.

Parameters

- **uncut_subsystem** (*Subsystem*) – The subsystem without the cut applied.
- **cut** (*Cut*) – The cut to evaluate.
- **unpartitioned_constellation** (*Constellation*) – The constellation of the uncut subsystem.

Returns |*BigMip*| – The *BigMip* for that cut.

pyphi.compute.big_phi.**big_mip_bipartitions**(*nodes*)
Return all Φ cuts for the given nodes.

This value changes based on *config.CUT_ONE_APPROXIMATION*.

Parameters **nodes** (*tuple[int]*) – The node indices to partition.

Returns *list[|Cut|]* – All unidirectional partitions.

pyphi.compute.big_phi.**big_mip**(*cache_key*, *subsystem*)
Return the minimal information partition of a subsystem.

Parameters **subsystem** (Subsystem) – The candidate set of nodes.

Returns |*BigMip*| – A nested structure containing all the data from the intermediate calculations. The top level contains the basic MIP information for the given subsystem.

pyphi.compute.big_phi.**big_phi**(*subsystem*)
Return the Φ value of a subsystem.

pyphi.compute.big_phi.**subsystems**(*network*, *state*)
Return a generator of all **possible** subsystems of a network.

Does not return subsystems that are in an impossible state.

pyphi.compute.big_phi.**all_complexes**(*network*, *state*)
Return a generator for all complexes of the network.

Includes reducible, zero-phi complexes (which are not, strictly speaking, complexes at all).

pyphi.compute.big_phi.**possible_complexes**(*network*, *state*)
Return a generator of subsystems of a network that could be a complex.

This is the just powerset of the nodes that have at least one input and output (nodes with no inputs or no outputs cannot be part of a main complex, because they do not have a causal link with the rest of the subsystem in the past or future, respectively).

Does not include subsystems in an impossible state.

Parameters

- **network** (Network) – The network for which to return possible complexes.
- **state** (*tuple[int]*) – The state of the network.

Yields *Subsystem* – The next subsystem which could be a complex.

pyphi.compute.big_phi.**complexes**(*network*, *state*)
Return all irreducible complexes of the network.

pyphi.compute.big_phi.**main_complex**(*network*, *state*)
Return the main complex of the network.

pyphi.compute.big_phi.**condensed**(*network*, *state*)
Return the set of maximal non-overlapping complexes.

compute.concept

pyphi.compute.concept.**concept**(*subsystem*, *mechanism*, *purviews=False*, *past_purviews=False*, *future_purviews=False*)
Return the concept specified by a mechanism within a subsytem.

Parameters

- **subsystem** (*Subsytem*) – The context in which the mechanism should be considered.
- **mechanism** (*tuple[int]*) – The candidate set of nodes.

Keyword Arguments

- **purviews** (*tuple[tuple[int]]*) – Restrict the possible purviews to those in this list.
- **past_purviews** (*tuple[tuple[int]]*) – Restrict the possible cause purviews to those in this list. Takes precedence over purviews.
- **future_purviews** (*tuple[tuple[int]]*) – Restrict the possible effect purviews to those in this list. Takes precedence over purviews.

Returns |*Concept*| – The pair of maximally irreducible cause/effect repertoires that constitute the concept specified by the given mechanism.

pyphi.compute.concept.**constellation**(*subsystem*, *mechanisms=False*, *purviews=False*, *past_purviews=False*, *future_purviews=False*)
Return the conceptual structure of this subsystem, optionally restricted to concepts with the mechanisms and purviews given in keyword arguments.

If you will not be using the full constellation, restricting the possible mechanisms and purviews can make this function much faster.

Parameters **subsystem** (Subsystem) – The subsystem for which to determine the constellation.

Keyword Arguments

- **mechanisms** (*tuple[tuple[int]]*) – A list of mechanisms, as node indices, to be considered as possible mechanisms for the concepts in the constellation.
- **purviews** (*tuple[tuple[int]]*) – A list of purviews, as node indices, to be considered as possible purviews for the concepts in the constellation.
- **past_purviews** (*tuple[tuple[int]]*) – A list of purviews, as node indices, to be considered as possible *cause* purviews for the concepts in the constellation. This takes precedence over the more general purviews option.

- **future_purviews** (*tuple[tuple[int]]*) – A list of purviews, as node indices, to be considered as possible *effect* purviews for the concepts in the constellation. This takes precedence over the more general `purviews` option.

Returns |*Constellation*| – A tuple of every *Concept* in the constellation.

`pyphi.compute.concept.`**conceptual_information**(*subsystem*)
Return the conceptual information for a subsystem.

This is the distance from the subsystem's constellation to the null concept.

compute.distance

`pyphi.compute.distance.`**measure**(*d1*, *d2*)
Compute the distance between two repertoires.

Parameters

- **d1** (*np.ndarray*) – The first repertoire.
- **d2** (*np.ndarray*) – The second repertoire.

Returns *float* – The distance between `d1` and `d2`.

`pyphi.compute.distance.`**concept_distance**(*c1*, *c2*)
Return the distance between two concepts in concept-space.

Parameters

- **c1** (*Concept*) – The first concept.
- **c2** (*Concept*) – The second concept.

Returns *float* – The distance between the two concepts in concept-space.

`pyphi.compute.distance.`**constellation_distance**(*C1*, *C2*)
Return the distance between two constellations in concept-space.

Parameters

- **C1** (*Constellation*) – The first constellation.
- **C2** (*Constellation*) – The second constellation.

Returns *float* – The distance between the two constellations in concept-space.

config

The configuration is loaded upon import from a YAML file in the directory where PyPhi is run: `pyphi_config.yml`. If no file is found, the default configuration is used.

The various options are listed here with their defaults

```
>>> import pyphi
>>> defaults = pyphi.config.DEFAULTS
```

It is also possible to manually load a YAML configuration file within your script:

```
>>> pyphi.config.load_config_file('pyphi_config.yml')
```

Or load a dictionary of configuration values:

```
>>> pyphi.config.load_config_dict({'SOME_CONFIG': 'value'})
```

Theoretical approximations

This section deals with assumptions that speed up computation at the cost of theoretical accuracy.

- `pyphi.config.ASSUME_CUTS_CANNOT_CREATE_NEW_CONCEPTS`: In certain cases, making a cut can actually cause a previously reducible concept to become a proper, irreducible concept. Assuming this can never happen can increase performance significantly, however the obtained results are not strictly accurate.

```
>>> defaults['ASSUME_CUTS_CANNOT_CREATE_NEW_CONCEPTS']
False
```

- `pyphi.config.CUT_ONE_APPROXIMATION`: When determining the MIP for Φ, this restricts the set of system cuts that are considered to only those that cut the inputs or outputs of a single node. This restricted set of cuts scales linearly with the size of the system; the full set of all possible bipartitions scales exponentially. This approximation is more likely to give theoretically accurate results with modular, sparsely-connected, or homogeneous networks.

```
>>> defaults['CUT_ONE_APPROXIMATION']
False
```

- `pyphi.config.MEASURE`: The measure to use when computing distances between repertoires and concepts. The default is `EMD`; the Earth Movers's Distance. `KLD` is the Kullback-Leibler Divergence. If `L1` is chosen, the `L1` distance is initially used instead of the EMD when computing MIPs but, if a mechanism and purview are found to be irreducible, the φ value of the MIP is recalculated using the EMD.

```
>>> defaults['MEASURE']
'EMD'
```

System resources

These settings control how much processing power and memory is available for PyPhi to use. The default values may not be appropriate for your use-case or machine, so **please check these settings before running anything**. Otherwise, there is a risk that simulations might crash (potentially after running for a long time!), resulting in data loss.

- `pyphi.config.PARALLEL_CONCEPT_EVALUATION`: Control whether concepts are evaluated in parallel when computing constellations.

```
>>> defaults['PARALLEL_CONCEPT_EVALUATION']
False
```

- `pyphi.config.PARALLEL_CUT_EVALUATION`: Control whether system cuts are evaluated in parallel, which requires more memory. If cuts are evaluated sequentially, only two *BigMip* instances need to be in memory at once.

```
>>> defaults['PARALLEL_CUT_EVALUATION']
True
```

Warning: `PARALLEL_CONCEPT_EVALUATION` and `PARALLEL_CUT_EVALUATION` should not both be set to `True`. Enabling both parallelization modes will slow down computations. If you are doing Φ-computations (with `big_mip`, `main_complex`, etc.) `PARALLEL_CUT_EVALUATION` will be fastest. Use `PARALLEL_CONCEPT_EVALUATION` if you are only computing constellations.

- `pyphi.config.NUMBER_OF_CORES`: Control the number of CPU cores used to evaluate unidirectional cuts. Negative numbers count backwards from the total number of available cores, with `-1` meaning "use all available cores."

```
>>> defaults['NUMBER_OF_CORES']
-1
```

- `pyphi.config.MAXIMUM_CACHE_MEMORY_PERCENTAGE`: PyPhi employs several in-memory caches to speed up computation. However, these can quickly use a lot of memory for large networks or large numbers of them; to avoid thrashing, this options limits the percentage of a system's RAM that the caches can collectively use.

```
>>> defaults['MAXIMUM_CACHE_MEMORY_PERCENTAGE']
50
```

Caching

PyPhi is equipped with a transparent caching system for *BigMip* objects which stores them as they are computed to avoid having to recompute them later. This makes it easy to play around interactively with the program, or to accumulate results with minimal effort. For larger projects, however, it is recommended that you manage the results explicitly, rather than relying on the cache. For this reason it is disabled by default.

- `pyphi.config.CACHE_BIGMIPS`: Control whether *BigMip* objects are cached and automatically retreived.

```
>>> defaults['CACHE_BIGMIPS']
False
```

- `pyphi.config.CACHE_POTENTIAL_PURVIEWS`: Controls whether the potential purviews of mechanisms of a network are cached. Caching speeds up computations by not recomputing expensive reducibility checks, but uses additional memory.

```
>>> defaults['CACHE_POTENTIAL_PURVIEWS']
True
```

- `pyphi.config.CACHING_BACKEND`: Control whether precomputed results are stored and read from a database or from a local filesystem-based cache in the current directory. Set this to 'fs' for the filesystem, 'db' for the database. Caching results on the filesystem is the easiest to use but least robust caching system. Caching results in a database is more robust and allows for caching individual concepts, but requires installing MongoDB.

```
>>> defaults['CACHING_BACKEND']
'fs'
```

- `pyphi.config.FS_CACHE_VERBOSITY`: Control how much caching information is printed. Takes a value between 0 and 11. Note that printing during a loop iteration can slow down the loop considerably.

```
>>> defaults['FS_CACHE_VERBOSITY']
0
```

- `pyphi.config.FS_CACHE_DIRECTORY`: If the caching backend is set to use the filesystem, the cache will be stored in this directory. This directory can be copied and moved around if you want to reuse results _e.g._ on a another computer, but it must be in the same directory from which PyPhi is being run.

```
>>> defaults['FS_CACHE_DIRECTORY']
'__pyphi_cache__'
```

- `pyphi.config.MONGODB_CONFIG`: Set the configuration for the MongoDB database backend. This only has an effect if the caching backend is set to use the database.

```
>>> defaults['MONGODB_CONFIG']['host']
'localhost'
>>> defaults['MONGODB_CONFIG']['port']
27017
>>> defaults['MONGODB_CONFIG']['database_name']
'pyphi'
>>> defaults['MONGODB_CONFIG']['collection_name']
'cache'
```

- `pyphi.config.REDIS_CACHE`: Specifies whether to use Redis to cache Mice.

```
>>> defaults['REDIS_CACHE']
False
```

- **`pyphi.config.REDIS_CONFIG`: Configure the Redis database backend. These** are the defaults in the provided `redis.conf` file.

```
>>> defaults['REDIS_CONFIG']['host']
'localhost'
>>> defaults['REDIS_CONFIG']['port']
6379
```

Logging

These settings control how PyPhi handles log messages. Logs can be written to standard output, a file, both, or none. If these simple default controls are not flexible enough for you, you can override the entire logging configuration. See the documentation on Python's logger for more information.

- `pyphi.config.LOG_STDOUT_LEVEL`: Controls the level of log messages written to standard output. Can be one of `'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`, or `None`. `DEBUG` is the least restrictive level and will show the most log messages. `CRITICAL` is the most restrictive level and will only display information about unrecoverable errors. If set to `None`, logging to standard output will be disabled entirely.

```
>>> defaults['LOG_STDOUT_LEVEL']
'WARNING'
```

- `pyphi.config.LOG_FILE_LEVEL: Controls the level of log messages written to the log file. This option has the same possible values as ``LOG_STDOUT_LEVEL`.

```
>>> defaults['LOG_FILE_LEVEL']
'INFO'
```

- `pyphi.config.LOG_FILE`: Control the name of the logfile.

```
>>> defaults['LOG_FILE']
'pyphi.log'
```

- `pyphi.config.LOG_CONFIG_ON_IMPORT`: Controls whether the current configuration is printed when PyPhi is imported.

```
>>> defaults['LOG_CONFIG_ON_IMPORT']
True
```

Numerical precision

- `pyphi.config.PRECISION`: Computations in PyPhi rely on finding the Earth Mover's Distance. This is done via an external C++ library that uses flow-optimization to find a good approximation of the EMD. Consequently, systems with zero Φ will sometimes be computed to have a small but non-zero amount. This setting controls the number of decimal places to which PyPhi will consider EMD calculations accurate. Values of Φ lower than `10e-PRECISION` will be considered insignificant and treated as zero. The default value is about as accurate as the EMD computations get.

```
>>> defaults['PRECISION']
6
```

Miscellaneous

- `pyphi.config.VALIDATE_SUBSYSTEM_STATES`: Control whether PyPhi checks if the subsystems's state is possible (reachable from some past state), given the subsystem's TPM (**which is conditioned on background conditions**). If this is turned off, then **calculated Φ values may not be valid**, since they may be associated with a subsystem that could never be in the given state.

```
>>> defaults['VALIDATE_SUBSYSTEM_STATES']
True
```

- `pyphi.config.SINGLE_NODES_WITH_SELFLOOPS_HAVE_PHI`: If set to `True`, this defines the Phi value of subsystems containing only a single node with a self-loop to be `0.5`. If set to False, their Φ will be actually be computed (to be zero, in this implementation).

```
>>> defaults['SINGLE_NODES_WITH_SELFLOOPS_HAVE_PHI']
False
```

- `pyphi.config.REPR_VERBOSITY`: Controls the verbosity of `__repr__` methods on PyPhi objects. Can be set to `0`, `1`, or `2`. If set to `1`, calling `repr` on PyPhi objects will return pretty-formatted and legible strings, excluding repertoires. If set to `2`, `repr` calls also include repertoires.

  Although this breaks the convention that `__repr__` methods should return a representation which can reconstruct the object, readable representations are convenient since the Python REPL calls `repr` to represent all objects in the shell and PyPhi is often used interactively with the REPL. If set to `0`, `repr` returns more traditional object representations.

```
>>> defaults['REPR_VERBOSITY']
2
```

- `pyphi.config.PARTITION_MECHANISMS`: If `True`, φ-MIP computations will only consider bipartitions that strictly partition the mechanism. That is, for the mechanism `(A, B)` and purview `(B, C, D)` the partition

```
AB    []
-- X --
B     CD
```

is not considered, but

```
A     B
-- X --
B     CD
```

is. The following is also valid:

```
AB    []
-- X ---
[]    BCD
```

In addition, this option introduces wedge tripartitions of the form

```
A     B    []
-- X - X --
B     C    D
```

where the mechanism in the third part is always empty.

Finally, in the case of a φ-tie when computing MICE, this setting choses the MIP with smallest purview instead the largest (which is the default behavior.)

```
>>> defaults['PARTITION_MECHANISMS']
False
```

- `pyphi.config.PARTITION_MECHANISMS`: If `True`, φ-MIP computations will only consider bipartitions that strictly partition the mechanism. That is, for the mechanism `(A, B)` and purview `(B, C, D)` the partition

```
AB    []
-- X --
B     CD
```

is not considered, but

```
A     B
-- X --
B     CD
```

is. The following is also valid:

```
AB    []
-- X ---
[]    BCD
```

Additionally, in the case of a φ-tie when computing MICE, this setting choses the MIP with smallest purview instead the largest (which is the default behavior.)

```
>>> defaults['PARTITION_MECHANISMS']
False
```

pyphi.config.**load_config_dict**(*config*)
Load configuration values.

Parameters **config** (*dict*) – The dict of config to load.

pyphi.config.**load_config_file**(*filename*)
Load config from a YAML file.

pyphi.config.**load_config_default**()
Load default config values.

pyphi.config.**get_config_string**()
Return a string representation of the currently loaded configuration.

pyphi.config.**print_config**()
Print the current configuration.

pyphi.config.**configure_logging**()
Configure PyPhi logging based on the loaded configuration.

Note: if PyPhi config options that control logging are changed after they are loaded (eg. in testing), the Python logging configuration will stay the same unless you manually reconfigure the logging by calling this function.

TODO: call this in *config.override*?

class pyphi.config.**override**(***new_conf*)
Decorator and context manager to override config values.

The initial configuration values are reset after the decorated function returns or the context manager completes it block, even if the function or block raises an exception. This is intended to be used by testcases which require specific configuration values.

Example

```
>>> from pyphi import config
>>>
>>> @config.override(PRECISION=20000)
... def test_something():
...     assert config.PRECISION == 20000
...
>>> test_something()
>>> with config.override(PRECISION=100):
...     assert config.PRECISION == 100
...
```

__enter__()
Save original config values; override with new ones.

__exit__(**exc*)
Reset config to initial values; reraise any exceptions.

constants

Package-wide constants.

class pyphi.constants.**Direction**
Constants that parametrize cause and effect methods.

Accessed using `Direction.PAST` and `Direction.FUTURE`.

PAST = 0

FUTURE = 1

BIDIRECTIONAL = 2

pyphi.constants.**EPSILON = 1e-06**
The threshold below which we consider differences in phi values to be zero.

pyphi.constants.**FILESYSTEM = 'fs'**
Label for the filesystem cache backend.

pyphi.constants.**DATABASE = 'db'**
Label for the MongoDB cache backed.

pyphi.constants.**PICKLE_PROTOCOL = 4**
The protocol used for pickling objects.

pyphi.constants.**joblib_memory = Memory(cachedir='__pyphi_cache__/joblib')**
The joblib Memory object for persistent caching without a database.

pyphi.constants.**EMD = 'EMD'**
Earth Movers Distance

pyphi.constants.**KLD = 'KLD'**
Kullback-Leibler Divergence

pyphi.constants.**L1 = 'L1'**
L1 distance

pyphi.constants.**MEASURES = ['EMD', 'KLD', 'L1']**
All available measures

convert

Conversion functions.

pyphi.convert.**nodes2indices**(*nodes*)

pyphi.convert.**nodes2state**(*nodes*)

pyphi.convert.**state2holi_index**(*state*)
Convert a PyPhi state-tuple to a decimal index according to the **HOLI** convention.

Parameters **state** (*tuple[int]*) – A state-tuple where the i^{th} element of the tuple gives the state of the i^{th} node.

Returns *int* – A decimal integer corresponding to a network state under the **HOLI** convention.

Examples

```
>>> from pyphi.convert import state2loli_index
>>> state2holi_index((1, 0, 0, 0, 0))
16
>>> state2holi_index((1, 1, 1, 0, 0, 0, 0, 0))
224
```

pyphi.convert.**state2loli_index**(*state*)
Convert a PyPhi state-tuple to a decimal index according to the **LOLI** convention.

Parameters **state** (*tuple[int]*) – A state-tuple where the i^{th} element of the tuple gives the state of the i^{th} node.

Returns *int* – A decimal integer corresponding to a network state under the **LOLI** convention.

Examples

```
>>> from pyphi.convert import state2loli_index
>>> state2loli_index((1, 0, 0, 0, 0))
1
>>> state2loli_index((1, 1, 1, 0, 0, 0, 0, 0))
7
```

pyphi.convert.**loli_index2state**(*i*, *number_of_nodes*)

Convert a decimal integer to a PyPhi state tuple with the **LOLI** convention.

The output is the reverse of *holi_index2state*.

Parameters **i** (*int*) – A decimal integer corresponding to a network state under the **LOLI** convention.

Returns *tuple[int]* – A state-tuple where the i^{th} element of the tuple gives the state of the i^{th} node.

Examples

```
>>> from pyphi.convert import loli_index2state
>>> number_of_nodes = 5
>>> loli_index2state(1, number_of_nodes)
(1, 0, 0, 0, 0)
>>> number_of_nodes = 8
>>> loli_index2state(7, number_of_nodes)
(1, 1, 1, 0, 0, 0, 0, 0)
```

pyphi.convert.**holi_index2state**(*i*, *number_of_nodes*)

Convert a decimal integer to a PyPhi state tuple using the **HOLI** convention that high-order bits correspond to low-index nodes.

The output is the reverse of *loli_index2state*.

Parameters **i** (*int*) – A decimal integer corresponding to a network state under the **HOLI** convention.

Returns *tuple[int]* – A state-tuple where the i^{th} element of the tuple gives the state of the i^{th} node.

Examples

```
>>> from pyphi.convert import holi_index2state
>>> number_of_nodes = 5
>>> holi_index2state(1, number_of_nodes)
(0, 0, 0, 0, 1)
>>> number_of_nodes = 8
>>> holi_index2state(7, number_of_nodes)
(0, 0, 0, 0, 0, 1, 1, 1)
```

pyphi.convert.**to_n_dimensional**(*tpm*)
Reshape a state-by-node TPM to the N-D form.

See documentation for the *Network* object for more information on TPM formats.

pyphi.convert.**state_by_state2state_by_node**(*tpm*)
Convert a state-by-state TPM to a state-by-node TPM.

Note: The indices of the rows and columns of the state-by-state TPM are assumed to follow the **LOLI** convention. The indices of the rows of the resulting state-by-node TPM also follow the **LOLI** convention. See the documentation for the *examples* module for more info on these conventions.

Parameters **tpm** (*list[list] or np.ndarray*) – A square state-by-state TPM with row and column indices following the **LOLI** convention.

Returns *np.ndarray* – A state-by-node TPM, with row indices following the **LOLI** convention.

Examples

```
>>> from pyphi.convert import state_by_state2state_by_node
>>> tpm = np.array([[0.5, 0.5, 0.0, 0.0],
...                 [0.0, 1.0, 0.0, 0.0],
...                 [0.0, 0.2, 0.0, 0.8],
...                 [0.0, 0.3, 0.7, 0.0]])
>>> state_by_state2state_by_node(tpm)
array([[[ 0.5,  0. ],
        [ 1. ,  0.8]],

       [[ 1. ,  0. ],
        [ 0.3,  0.7]]])
```

pyphi.convert.**state_by_node2state_by_state**(*tpm*)
Convert a state-by-node TPM to a state-by-state TPM.

Note: **A nondeterministic state-by-node TPM can have more than one representation as a state-by-state TPM.** However, the mapping can be made to be one-to-one if we assume the TPMs to be conditionally independent. Therefore, given a nondeterministic state-by-node TPM, this function returns the corresponding conditionally independent state-by-state.

Note: The indices of the rows of the state-by-node TPM are assumed to follow the **LOLI** convention, while the indices of the columns follow the **HOLI** convention. The indices of the rows and columns of the resulting state-by-state TPM both follow the **HOLI** convention.

Parameters **tpm** (*list[list] or np.ndarray*) – A state-by-node TPM with row indices following the **LOLI** convention and column indices following the **HOLI** convention.

Returns *np.ndarray* – A state-by-state TPM, with both row and column indices following the **HOLI** convention.

```
>>> from pyphi.convert import state_by_node2state_by_state
>>> tpm = np.array([[1, 1, 0],
...                 [0, 0, 1],
...                 [0, 1, 1],
...                 [1, 0, 0],
...                 [0, 0, 1],
...                 [1, 0, 0],
...                 [1, 1, 1],
...                 [1, 0, 1]])
>>> state_by_node2state_by_state(tpm)
array([[ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.]])
```

db

Interface to MongoDB that exposes it as a key-value store.

pyphi.db.**find**(*key*)
Return the value associated with a key.

If there is no value with the given key, returns `None`.

pyphi.db.**insert**(*key*, *value*)
Store a value with a key.

If the key is already present in the database, this does nothing.

pyphi.db.**generate_key**(*filtered_args*)
Get a key from some input.

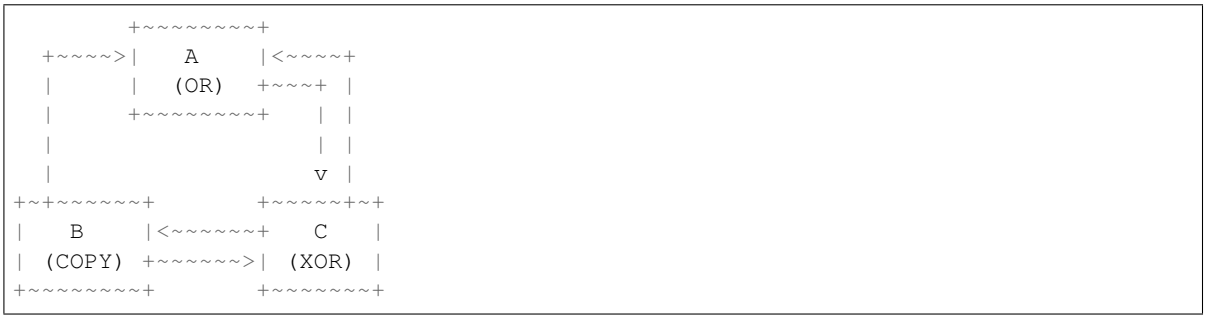
This function should be used whenever a key is needed, to keep keys consistent.

examples

Example networks and subsystems to go along with the documentation.

pyphi.examples.**basic_network**(*cm=False*)
A simple 3-node network with roughly two bits of Φ.

Diagram:



TPM:

Past state	Current state
A, B, C	A, B, C
0, 0, 0	0, 0, 0
1, 0, 0	0, 0, 1
0, 1, 0	1, 0, 1
1, 1, 0	1, 0, 0
0, 0, 1	1, 1, 0
1, 0, 1	1, 1, 1
0, 1, 1	1, 1, 1
1, 1, 1	1, 1, 0

Connectivity matrix:

.	A	B	C
A	0	0	1
B	1	0	1
C	1	1	0

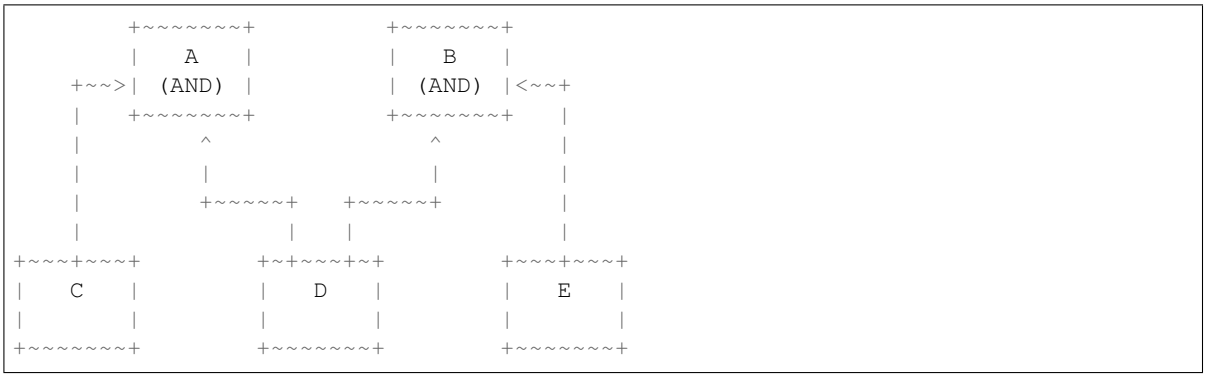
Note: $CM_{i,j} = 1$ means that node i is connected to node j.

pyphi.examples.**basic_subsystem**()
A subsystem containing all the nodes of the *basic_network()*.

pyphi.examples.**residue_network**()
The network for the residue example.

Current and past state are all nodes off.

Diagram:



Connectivity matrix:

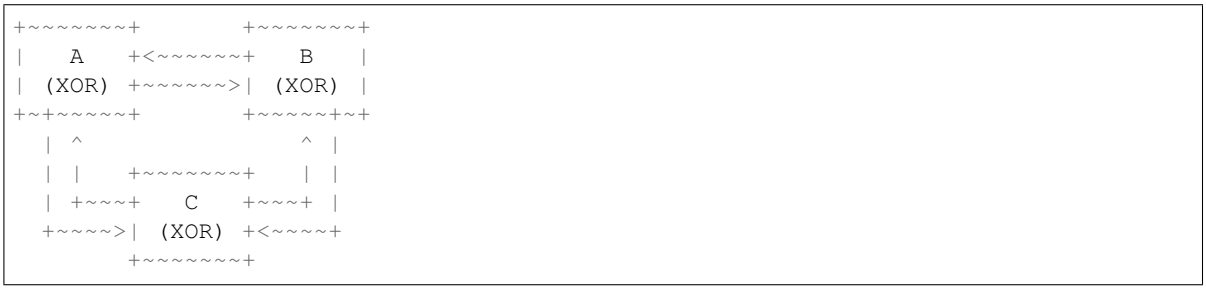
.	A	B	C	D	E
A	0	0	0	0	0
B	0	0	0	0	0
C	1	0	0	0	0
D	1	1	0	0	0
E	0	1	0	0	0

pyphi.examples.**residue_subsystem**()
The subsystem containing all the nodes of the *residue_network()*.

pyphi.examples.**xor_network**()
A fully connected system of three XOR gates. In the state `(0, 0, 0)`, none of the elementary mechanisms

exist.

Diagram:



Connectivity matrix:

.	A	B	C
A	0	1	1
B	1	0	1
C	1	1	0

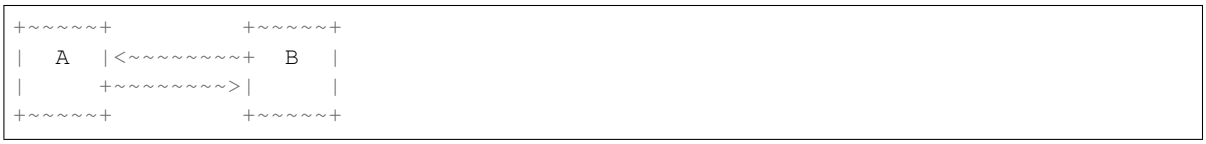
pyphi.examples.**xor_subsystem**()

The subsystem containing all the nodes of the *xor_network()*.

pyphi.examples.**cond_depend_tpm**()

A system of two general logic gates A and B such if they are in the same state they stay the same, but if they are in different states, they flip with probability 50%.

Diagram:



TPM:

	(0, 0)	(1, 0)	(0, 1)	(1, 1)
(0, 0)	1.0	0.0	0.0	0.0
(1, 0)	0.0	0.5	0.5	0.0
(0, 1)	0.0	0.5	0.5	0.0
(1, 1)	0.0	0.0	0.0	1.0

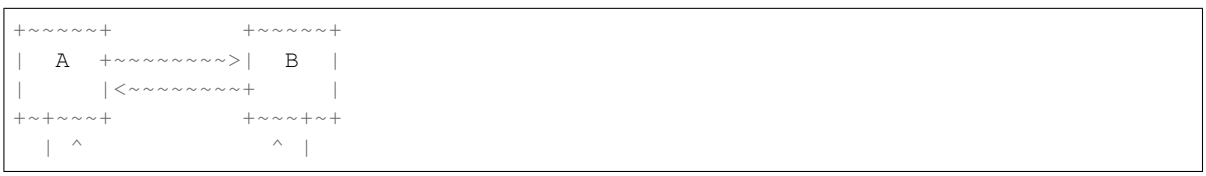
Connectivity matrix:

.	A	B
A	0	1
B	1	0

pyphi.examples.**cond_independ_tpm**()

A system of three general logic gates A, B and C such that: if A and B are in the same state then they stay the same; if they are in different states, they flip if C is **ON** and stay the same if C is **OFF**; and C is **ON** 50% of the time, independent of the previous state.

Diagram:



TPM:

	(0, 0, 0)	(1, 0, 0)	(0, 1, 0)	(1, 1, 0)	(0, 0, 1)	(1, 0, 1)	(0, 1, 1)	(1, 1, 1)
(0, 0, 0)	0.5	0.0	0.0	0.0	0.5	0.0	0.0	0.0
(1, 0, 0)	0.0	0.5	0.0	0.0	0.0	0.5	0.0	0.0
(0, 1, 0)	0.0	0.0	0.5	0.0	0.0	0.0	0.5	0.0
(1, 1, 0)	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.5
(0, 0, 1)	0.5	0.0	0.0	0.0	0.5	0.0	0.0	0.0
(1, 0, 1)	0.0	0.0	0.5	0.0	0.0	0.0	0.5	0.0
(0, 1, 1)	0.0	0.5	0.0	0.0	0.0	0.5	0.0	0.0
(1, 1, 1)	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.5

Connectivity matrix:

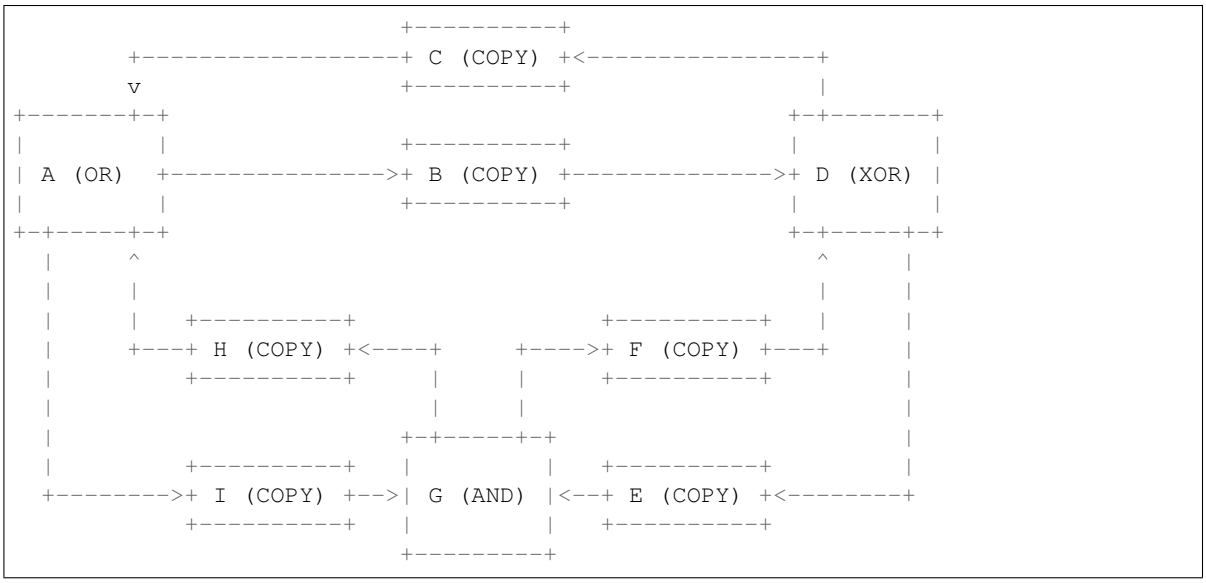
.	A	B	C
A	0	1	0
B	1	0	0
C	1	1	0

pyphi.examples.**propagation_delay_network**()

A version of the primary example from the IIT 3.0 paper with deterministic COPY gates on each connection. These copy gates essentially function as propagation delays on the signal between OR, AND and XOR gates from the original system.

The current and past states of the network are also selected to mimic the corresponding states from the IIT 3.0 paper.

Diagram:



Connectivity matrix:

.	A	B	C	D	E	F	G	H	I
A	0	1	0	0	0	0	0	0	1
B	0	0	0	1	0	0	0	0	0
C	1	0	0	0	0	0	0	0	0
D	0	0	1	0	1	0	0	0	0
E	0	0	0	0	0	0	1	0	0
F	0	0	0	1	0	0	0	0	0
G	0	0	0	0	0	1	0	1	0
H	1	0	0	0	0	0	0	0	0
I	0	0	0	0	0	0	1	0	0

States:

In the IIT 3.0 paper example, the past state of the system has only the XOR gate on. For the propagation delay network, this corresponds to a state of `(0, 0, 0, 1, 0, 0, 0, 0, 0)`.

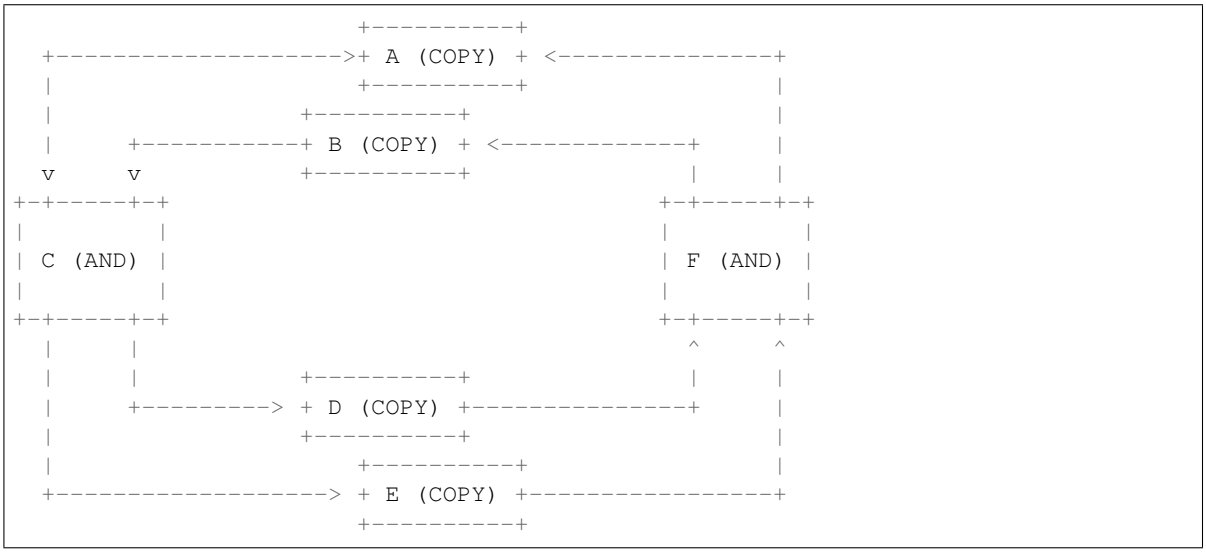
The current state of the IIT 3.0 example has only the OR gate on. By advancing the propagation delay system two time steps, the current state `(1, 0, 0, 0, 0, 0, 0, 0, 0)` is achieved, with corresponding past state `(0, 0, 1, 0, 1, 0, 0, 0, 0)`.

pyphi.examples.**macro_network**()
A network of micro elements which has greater integrated information after coarse graining to a macro scale.

pyphi.examples.**macro_subsystem**()

pyphi.examples.**blackbox_network**()
A micro-network to demonstrate blackboxing.

Diagram:



Connectivity Matrix:

.	A	B	C	D	E	F
A	0	0	1	0	0	0
B	0	0	1	0	0	0
C	0	0	0	1	1	0

```
| D | 0 | 0 | 0 | 0 | 0 | 1 |
+---+---+---+---+---+---+---+
| E | 0 | 0 | 0 | 0 | 0 | 1 |
+---+---+---+---+---+---+---+
| F | 1 | 1 | 0 | 0 | 0 | 0 |
+---+---+---+---+---+---+---+
```

In the documentation example, the state is (0, 0, 0, 0, 0, 0).

pyphi.examples.**rule110_network**()
A network of three elements which follows the logic of the Rule 110 cellular automaton with current and past state (0, 0, 0).

pyphi.examples.**rule154_network**()
A network of three elements which follows the logic of the Rule 154 cellular automaton.

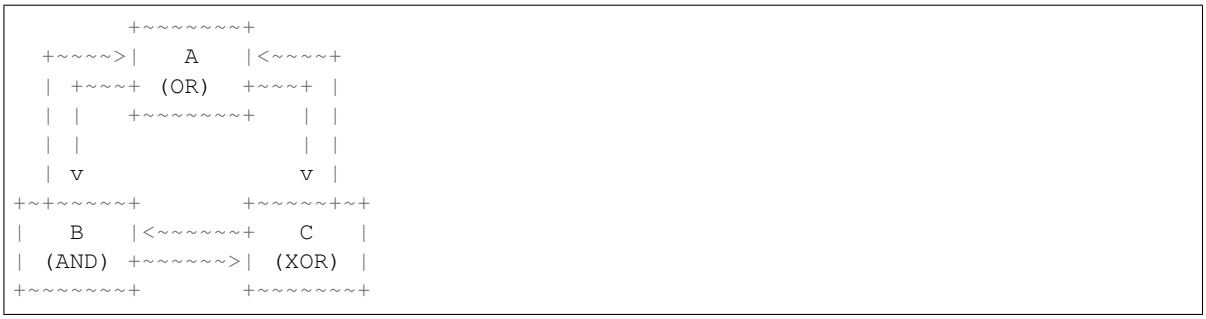
pyphi.examples.**fig1a**()
The network shown in Figure 1A of the 2014 IIT 3.0 paper.

pyphi.examples.**fig3a**()
The network shown in Figure 3A of the 2014 IIT 3.0 paper.

pyphi.examples.**fig3b**()
The network shown in Figure 3B of the 2014 IIT 3.0 paper.

pyphi.examples.**fig4**()
The network shown in Figure 4 of the 2014 IIT 3.0 paper.

Diagram:



pyphi.examples.**fig5a**()
The network shown in Figure 5A of the 2014 IIT 3.0 paper.

Diagram:

pyphi.examples.**fig5b**()
The network shown in Figure 5B of the 2014 IIT 3.0 paper.

Diagram:

pyphi.examples.**fig14**()
The network shown in Figure 1A of the 2014 IIT 3.0 paper.

pyphi.examples.**fig16**()
The network shown in Figure 5B of the 2014 IIT 3.0 paper.

pyphi.examples.**ac_ex1_network**()
A network of three elements, an OR gate with two inputs.

pyphi.examples.**ac_ex1_context**()
The OR gate is ON, others are OFF just so they conform to the tpm

pyphi.examples.**ac_ex2_network**()
A network of four elements, one 'output' with three 'inputs' (A B C). The output turns ON if A AND B are ON or if C is ON.

pyphi.examples.**ac_ex2_context**()
The output is ON, others are OFF just so they conform to the tpm

pyphi.examples.**ac_ex3_network**()
A network of three elements, an output that only turns ON for a specific pattern of its input.

pyphi.examples.**ac_ex3_context**()
The output is OFF, the input are OFF just so they conform to the tpm

pyphi.examples.**fig10**()
The network shown in Figure 4 of the 2014 IIT 3.0 paper.

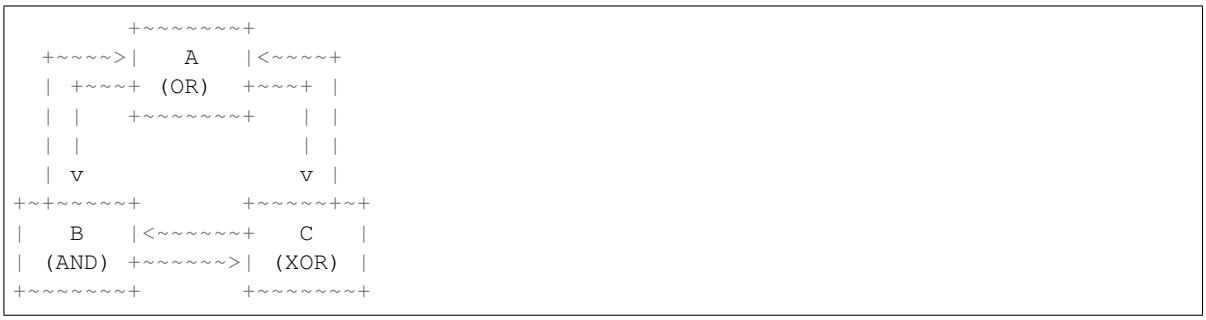
Diagram:

pyphi.examples.**fig6**()
The network shown in Figure 4 of the 2014 IIT 3.0 paper.

Diagram:

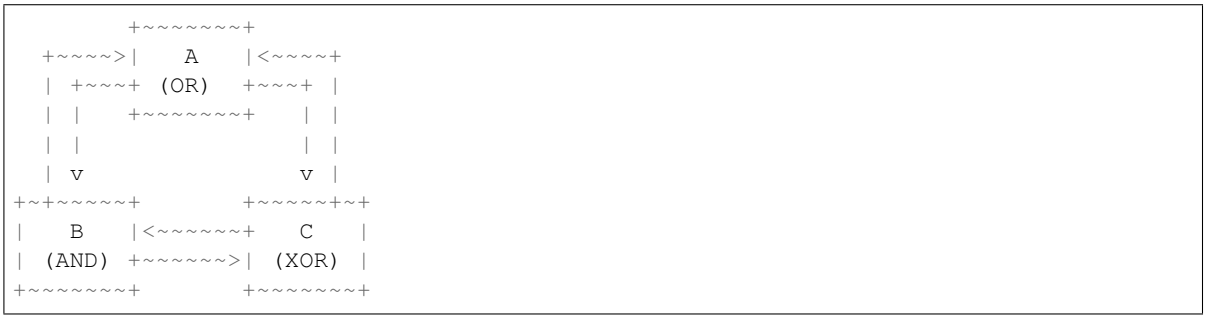
pyphi.examples.**fig8**()
The network shown in Figure 4 of the 2014 IIT 3.0 paper.

Diagram:



pyphi.examples.**fig9**()
The network shown in Figure 4 of the 2014 IIT 3.0 paper.

Diagram:



exceptions

PyPhi exceptions.

exception pyphi.exceptions.**StateUnreachableError**(*state*)
The current state cannot be reached from any past state.

exception pyphi.exceptions.**ConditionallyDependentError**
The TPM is conditionally dependent.

exception pyphi.exceptions.**JSONVersionError**
JSON was serialized with a different version of PyPhi.

jsonify

PyPhi- and NumPy-aware JSON serialization.

To be properly serialized and deserialized, PyPhi models must implement a `to_json` method which returns a dictionary of attribute names and attribute values. These attributes should be the names of arguments passed to the model constructor. If the constructor takes additional, fewer, or differnt arguments, the model needs to implement a custom `from_json` classmethod which takes a Python dictionary as an argument and returns a PyPhi object. For example:

```
class Phi:
    def __init__(self, phi):
        self.phi = phi

    def to_json(self):
        return {'phi': self.phi, 'twice_phi': 2 * self.phi}

    @classmethod
    def from_json(cls, json):
        return Phi(json['phi'])
```

The model must also be added to `jsonify._loadable_models`.

The JSON encoder adds the name of the model and the current PyPhi version to the JSON stream. The JSON decoder uses this metadata to recursively deserialize the stream to a nested PyPhi model structure. The decoder will raise an exception if the version of the JSON does not match the current version of PyPhi.

pyphi.jsonify.**jsonify**(*obj*)

Return a JSON-encodable representation of an object, recursively using any available `to_json` methods, converting NumPy arrays and datatypes to native lists and types along the way.

class pyphi.jsonify.**PyPhiJSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Extension of the default JSONEncoder that allows for serializing PyPhi objects with `jsonify`.

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

encode(*obj*)

Encode the output of `jsonify` with the default encoder.

iterencode(*obj*, ***kwargs*)
Analog to *encode* used by json.dump.

pyphi.jsonify.**dumps**(*obj*, ***user_kwargs*)
Serialize `obj` as JSON-formatted stream.

pyphi.jsonify.**dump**(*obj*, *fp*, ***user_kwargs*)
Serialize `obj` as a JSON-formatted stream and write to `fp` (a `.write()`-supporting file-like object.

class pyphi.jsonify.**PyPhiJSONDecoder**(**args*, ***kwargs*)
Extension of the default encoder which automatically deserializes PyPhi JSON to the appropriate model classes.

pyphi.jsonify.**loads**(*string*)
Deserialize a JSON string to a Python object.

pyphi.jsonify.**load**(*fp*)
Deserialize a JSON stream to a Python object.

macro

Methods for coarse-graining systems to different levels of spatial analysis.

pyphi.macro.**reindex**(*indices*)
Generate a new set of node indices, the size of indices.

pyphi.macro.**rebuild_system_tpm**(*node_tpms*)
Reconstruct the network TPM from a collection of node TPMs.

class pyphi.macro.**MacroSubsystem**(*network*, *state*, *nodes*, *cut=None*, *mice_cache=None*, *time_scale=1*, *blackbox=None*, *coarse_grain=None*)
A subclass of *Subsystem* implementing macro computations.

This subsystem performs blackboxing and coarse-graining of elements.

Unlike *Subsystem*, whose TPM has dimensionality equal to that of the subsystem's network and represents nodes external to the system using singleton dimensions, *MacroSubsystem* squeezes the TPM to remove these singletons. As a result, the node indices of the system are also squeezed to `0..n` so they properly index the TPM, and the state-tuple is reduced to the size of the system.

After each macro update (temporal blackboxing, spatial blackboxing, and spatial coarse-graining) the TPM, CM, nodes, and state are updated so that they correctly represent the updated system.

cut_indices
The indices of this system to be cut for Φ computations.

For macro computations the cut is applied to the underlying micro-system.

apply_cut(*cut*)
Return a cut version of this *MacroSubsystem*.

Parameters **cut** (Cut) – The cut to apply to this *MacroSubsystem*.

Returns *MacroSubsystem* – The cut version of this *MacroSubsystem*.

macro2micro(*macro_indices*)
Returns all micro indices which compose the elements specified by `macro_indices`.

__eq__(*other*)
Two macro systems are equal if each underlying *Subsystem* is equal and all macro attributes are equal.

TODO: handle cases where a MacroSubsystem is identical to a micro Subsystem, e.g. the macro has no timescale, hidden indices, etc.

class pyphi.macro.**CoarseGrain**

Represents a coarse graining of a collection of nodes.

partition

tuple[tuple] – The partition of micro-elements into macro-elements.

grouping

tuple[tuple[tuple]] – The grouping of micro-states into macro-states.

Create new instance of CoarseGrain(partition, grouping)

micro_indices

Indices of micro elements represented in this coarse-graining.

macro_indices

Indices of macro elements of this coarse-graining.

reindex()

Re-index this coarse graining to use squeezed indices.

The output grouping is translated to use indices `0..n`, where `n` is the number of micro indices in the coarse-graining. Re-indexing does not effect the state grouping, which is already index-independent.

Returns *CoarseGrain* – A new *CoarseGrain* object, indexed from `0..n`.

Example

```
>>> partition = ((1, 2),)
>>> grouping = (((0,), (1, 2)),)
>>> coarse_grain = CoarseGrain(partition, grouping)
>>> coarse_grain.reindex()
CoarseGrain(partition=((0, 1),), grouping=(((0,), (1, 2)),))
```

macro_state(*micro_state*)

Translate a micro state to a macro state

Parameters **micro_state** (*tuple[int]*) – The state of the micro nodes in this coarse-graining.

Returns *tuple[int]* – The state of the macro system, translated as specified by this coarse-graining.

Example

```
>>> coarse_grain = CoarseGrain(((1, 2),), (((0,), (1, 2)),))
>>> coarse_grain.macro_state((0, 0))
(0,)
>>> coarse_grain.macro_state((1, 0))
(1,)
>>> coarse_grain.macro_state((1, 1))
(1,)
```

make_mapping()

Return a mapping from micro-state to the macro-states based on the partition and state grouping of this coarse-grain.

Returns *(nd.ndarray)* – A mapping from micro-states to macro-states. The i^{th} entry in the mapping is the macro-state corresponding to the i^{th} micro-state.

macro_tpm(*micro_tpm*, *check_independence=True*)
Create a coarse-grained macro TPM.

Parameters

- **micro_tpm** (*nd.array*) – The TPM of the micro-system.
- **check_independence** (*bool*) – If `True`, the method will raise a `ConditionallyDependentError` if the macro TPM is not conditionally independent.

Returns *np.ndarray* – The state-by-node TPM of the macro-system.

class pyphi.macro.**Blackbox**
Class representing a blackboxing of a system.

partition
tuple[tuple[int]] – The partition of nodes into boxes.

output_indices
tuple[int] – Outputs of the blackboxes.

Create new instance of Blackbox(partition, output_indices)

hidden_indices
All elements hidden inside the blackboxes.

micro_indices
Indices of micro-elements in this blackboxing.

macro_indices
Fresh indices of macro-elements of the blackboxing.

reindex()
Squeeze the indices of this blackboxing to `0..n`.

Returns *Blackbox* – a new, reindexed *Blackbox*.

Example

```
>>> partition = ((3,), (2, 4))
>>> output_indices = (2, 3)
>>> blackbox = Blackbox(partition, output_indices)
>>> blackbox.reindex()
Blackbox(partition=((1,), (0, 2)), output_indices=(0, 1))
```

macro_state(*micro_state*)
Compute the macro-state of this blackbox.

This is just the state of the blackbox's output indices.

Parameters **micro_state** (*tuple[int]*) – The state of the micro-elements in the blackbox.

Returns *tuple[int]* – The state of the output indices.

in_same_box(*a*, *b*)
Returns `True` if nodes `a` and `b`` are in the same box.

pyphi.macro.**all_partitions**(*indices*)
Return a list of all possible coarse grains of a network.

Parameters **indices** (*tuple[int]*) – The micro indices to partition.

Yields *tuple[tuple]* – A possible partition. Each element of the tuple is a tuple of micro-elements which correspond to macro-elements.

pyphi.macro.**all_groupings**(*partition*)
Return all possible groupings of states for a particular coarse graining (partition) of a network.

Parameters **partition** (*tuple[tuple]*) – A partition of micro-elements into macro elements.

Yields *tuple[tuple[tuple]]* –

A grouping of micro-states into macro states of system.

TODO: document exactly how to interpret the grouping.

pyphi.macro.**all_coarse_grains**(*indices*)
Generator over all possible CoarseGrains of these indices.

Parameters **indices** (*tuple[int]*) – Node indices to coarse grain.

Yields *CoarseGrain* – The next *CoarseGrain* for indices.

pyphi.macro.**all_coarse_grains_for_blackbox**(*blackbox*)
Generator over all CoarseGrains for the given blackbox.

If a box has multiple outputs, those outputs are partitioned into the same coarse-grain macro-element.

pyphi.macro.**all_blackboxes**(*indices*)
Generator over all possible blackboxings of these indices.

Parameters **indices** (*tuple[int]*) – Nodes to blackbox.

Yields *Blackbox* – The next *Blackbox* of indices.

class pyphi.macro.**MacroNetwork**(*network*, *system*, *macro_phi*, *micro_phi*, *coarse_grain*, *time_scale=1*, *blackbox=None*)
A coarse-grained network of nodes.

See the *Emergence (coarse-graining and blackboxing)* example in the documentation for more information.

network
Network – The network object of the macro-system.

phi
float – The Φ of the network's main complex.

micro_network
Network – The network object of the corresponding micro system.

micro_phi
float – The Φ of the main complex of the corresponding micro-system.

coarse_grain
CoarseGrain – The coarse-graining of micro-elements into macro-elements.

time_scale
int – The time scale the macro-network run over.

blackbox
Blackbox – The blackboxing of micro elements in the network.

emergence
float – The difference between the Φ of the macro- and the micro-system.

emergence
Difference between the Φ of the macro and micro systems

pyphi.macro.**coarse_grain**(*network*, *state*, *internal_indices*)
Find the maximal coarse-graining of a micro-system.

Parameters

- **network** (Network) – The network in question.
- **state** (*tuple[int]*) – The state of the network.
- **internal_indices** (*tuple[int]*) – Nodes in the micro-system.

Returns *tuple[int, CoarseGrain]* – The phi-value of the maximal *CoarseGrain*.

pyphi.macro.**all_macro_systems**(*network*, *state*, *blackbox*, *coarse_grain*, *time_scales*)
Generator over all possible macro-systems for the network.

pyphi.macro.**emergence**(*network*, *state*, *blackbox=False*, *coarse_grain=True*, *time_scales=None*)
Check for the emergence of a micro-system into a macro-system.

Checks all possible blackboxings and coarse-grainings of a system to find the spatial scale with maximum integrated information.

Use the `blackbox` and `coarse_grain` args to specifiy whether to use blackboxing, coarse-graining, or both. The default is to just coarse-grain the system.

Parameters

- **network** (Network) – The network of the micro-system under investigation.
- **state** (*tuple[int]*) – The state of the network.
- **blackbox** (*bool*) – Set to `True` to enable blackboxing. Defaults to `False`.
- **coarse_grain** (*bool*) – Set to `True` to enable coarse-graining. Defaults to `True`.
- **time_scales** (*list[int]*) – List of all time steps over which to check for emergence.

Returns *MacroNetwork* – The maximal macro-system generated from the micro-system.

pyphi.macro.**phi_by_grain**(*network*, *state*)

pyphi.macro.**effective_info**(*network*)
Return the effective information of the given network.

Note: For details, see:

Hoel, Erik P., Larissa Albantakis, and Giulio Tononi. "Quantifying causal emergence shows that macro can beat micro." Proceedings of the National Academy of Sciences 110.49 (2013): 19790-19795.

Available online: doi: 10.1073/pnas.1314922110.

---

models

See *big_phi*, *concept*, and *cuts* for documentation.

pyphi.models.**BigMip**
Alias for *big_phi.BigMip*

pyphi.models.**Mip**
Alias for *concept.Mip*

pyphi.models.**Mice**
Alias for *concept.Mice*

pyphi.models.**Concept**
Alias for *concept.Concept*

pyphi.models.**Constellation**
Alias for *concept.Constellation*

pyphi.models.**Cut**
Alias for *cuts.Cut*

pyphi.models.**Part**
Alias for *cuts.Part*

pyphi.models.**Bipartition**
Alias for *cuts.Bipartition*

pyphi.models.**ActualCut**
Alias for *cuts.ActualCut*

pyphi.models.**AcMip**
Alias for `actual_causation.AcMip`

pyphi.models.**Occurence**
Alias for `actual_causation.Occurence`

pyphi.models.**AcBigMip**
Alias for `actual_causation.AcBigMip`

pyphi.models.**Account**
Alias for `actual_causation.Account`

pyphi.models.**DirectedAccount**
Alias for `actual_causation.DirectedAccount`

models.big_phi

class pyphi.models.big_phi.**BigMip**(*phi=None*, *unpartitioned_constellation=None*, *partitioned_constellation=None*, *subsystem=None*, *cut_subsystem=None*, *time=None*, *small_phi_time=None*)
A minimum information partition for Φ calculation.

BigMips may be compared with the built-in Python comparison operators (<, >, etc.). First, `phi` values are compared. Then, if these are equal up to `constants.PRECISION`, the size of the subsystem is compared (exclusion principle).

phi
float – The Φ value for the subsystem when taken against this MIP, *i.e.* the difference between the unpartitioned constellation and this MIP's partitioned constellation.

unpartitioned_constellation
Constellation – The constellation of the whole subsystem.

partitioned_constellation
Constellation – The constellation when the subsystem is cut.

subsystem
Subsystem – The subsystem this MIP was calculated for.

cut_subsystem
Subsystem – The subsystem with the minimal cut applied.

time
float – The number of seconds it took to calculate.

small_phi_time
float – The number of seconds it took to calculate the unpartitioned constellation.

cut
The unidirectional cut that makes the least difference to the subsystem.

network
The network this *BigMip* belongs to.

__bool__()
A BigMip is truthy if it is not reducible.

(That is, if it has a significant amount of Φ.)

to_json()

models.concept

class pyphi.models.concept.**Mip**(*phi*, *direction*, *mechanism*, *purview*, *partition*, *unpartitioned_repertoire*, *partitioned_repertoire*, *subsystem=None*)
A minimum information partition for φ calculation.

MIPs may be compared with the built-in Python comparison operators (<, >, etc.). First, `phi` values are compared. Then, if these are equal up to `constants.PRECISION`, the size of the mechanism is compared (exclusion principle).

phi
float – This is the difference between the mechanism's unpartitioned and partitioned repertoires.

direction
str –

direction (Direction): *PAST* or *FUTURE*.

mechanism
tuple[int] – The mechanism over which to evaluate the MIP.

purview
tuple[int] – The purview over which the unpartitioned repertoire differs the least from the partitioned repertoire.

partition
Bipartition – The partition that makes the least difference to the mechanism's repertoire.

unpartitioned_repertoire
np.ndarray – The unpartitioned repertoire of the mechanism.

partitioned_repertoire
np.ndarray – The partitioned repertoire of the mechanism. This is the product of the repertoires of each part of the partition.

phi

direction

mechanism

purview

partition

unpartitioned_repertoire

partitioned_repertoire

subsystem

__bool__()
A Mip is truthy if it is not reducible.

(That is, if it has a significant amount of φ.)

to_json()

class pyphi.models.concept.**Mice**(*mip*)
A maximally irreducible cause or effect (i.e., "core cause" or "core effect").

MICEs may be compared with the built-in Python comparison operators (<, >, etc.). First, phi values are compared. Then, if these are equal up to constants.PRECISION, the size of the mechanism is compared (exclusion principle).

phi
float – The difference between the mechanism's unpartitioned and partitioned repertoires.

direction
str – Either DIRECTIONS[PAST] or DIRECTIONS[FUTURE]. If DIRECTIONS[PAST] (DIRECTIONS[FUTURE]), this represents a maximally irreducible cause (effect).

mechanism
list(int) – The mechanism for which the MICE is evaluated.

purview
list(int) – The purview over which this mechanism's φ is maximal.

repertoire
np.ndarray – The unpartitioned repertoire of the mechanism over the purview.

partitioned_repertoire
np.ndarray – The partitioned repertoire of the mechanism over the purview.

mip
Mip – The minimum information partition for this mechanism.

to_json()

damaged_by_cut(*subsystem*)
Return True if this *Mice* is affected by the subsystem's cut.

The cut affects the *Mice* if it either splits the *Mice*'s mechanism or splits the connections between the purview and mechanism.

class pyphi.models.concept.**Concept**(*phi=None*, *mechanism=None*, *cause=None*, *effect=None*, *subsystem=None*, *normalized=False*, *time=None*)
A star in concept-space.

The phi attribute is the $\varphi^{\max}$ value. cause and effect are the MICE objects for the past and future, respectively.

Concepts may be compared with the built-in Python comparison operators (<, >, etc.). First, phi values are compared. Then, if these are equal up to constants.PRECISION, the size of the mechanism is compared.

phi
float – The size of the concept. This is the minimum of the φ values of the concept's core cause and core effect.

mechanism
tuple(int) – The mechanism that the concept consists of.

cause
|*Mice*| – The *Mice* representing the core cause of this concept.

effect
|Mice| – The `Mice` representing the core effect of this concept.

subsystem
Subsystem – This concept's parent subsystem.

time
float – The number of seconds it took to calculate.

location
`tuple(np.ndarray)` – The concept's location in concept space. The two elements of the tuple are the cause and effect repertoires.

cause_purview

effect_purview

cause_repertoire

effect_repertoire

__bool__()
A concept is truthy if it is not reducible.

(That is, if it has a significant amount of Φ.)

eq_repertoires(*other*)
Return whether this concept has the same cause and effect repertoires as another.

> **Warning:** This only checks if the cause and effect repertoires are equal as arrays; mechanisms, purviews, or even the nodes that node indices refer to, might be different.

emd_eq(*other*)
Return whether this concept is equal to another in the context of an EMD calculation.

expand_cause_repertoire(*new_purview=None*)
Expand a cause repertoire into a distribution over an entire network.

expand_effect_repertoire(*new_purview=None*)
Expand an effect repertoire into a distribution over an entire network.

expand_partitioned_cause_repertoire()
Expand a partitioned cause repertoire into a distribution over an entire network.

expand_partitioned_effect_repertoire()
Expand a partitioned effect repertoire into a distribution over an entire network.

to_json()

classmethod from_json(*dct*)

class pyphi.models.concept.**Constellation**
A constellation of concepts.

This is a wrapper around a tuple to provide a nice string representation and place to put constellation methods. Previously, constellations were represented as `tuple(|Concept|)`; this usage still works in all functions.

to_json()

classmethod from_json(*json*)

pyphi.models.concept.**normalize_constellation**(*constellation*)
Deterministically reorder the concepts in a constellation.

Parameters **constellation** (*Constellation*) – The constellation in question.

Returns

***Constellation*: The constellation, ordered lexicographically by** mechanism.

models.cuts

class pyphi.models.cuts.**Cut**

Represents a unidirectional cut.

severed

tuple[int] – Connections from this group of nodes to those in `intact` are severed.

intact

tuple[int] – Connections to this group of nodes from those in `severed` are severed.

Create new instance of Cut(severed, intact)

indices

Returns the indices of this cut.

splits_mechanism(*mechanism*)

Check if this cut splits a mechanism.

Parameters **mechanism** (*tuple[int]*) – The mechanism in question

Returns *bool* – `True` if *mechanism* has elements on both sides of the cut, otherwise `False`.

cuts_connections(*a*, *b*)

Check if this cut severs any connections from nodes *a* to *b*.

all_cut_mechanisms()

Return all mechanisms with elements on both sides of this cut.

Returns tuple[tuple[int]]

apply_cut(*cm*)

Return a modified connectivity matrix where the connections from one set of nodes to the other are destroyed.

cut_matrix()

Compute the cut matrix for this cut.

The cut matrix is a square matrix which represents connections severed by the cut. The matrix is shrunk to the size of the cut subsystem–not necessarily the size of the entire network.

Example

```
>>> cut = Cut((1,), (2,))
>>> cut.cut_matrix()
array([[ 0.,  1.],
       [ 0.,  0.]])
```

to_json()

class pyphi.models.cuts.**ActualCut**

Represents an actual cut for a context.

cause_part1
tuple(int) – Connections from this group to those in `effect_part2` are cut

cause_part2
tuple(int) – Connections from this group to those in `effect_part1` are cut

effect_part1
tuple(int) – Connections to this group from `cause_part2` are cut

effect_part2
tuple(int) – Connections to this group from `cause_part1` are cut

Create new instance of ActualCut(cause_part1, cause_part2, effect_part1, effect_part2)

indices
tuple[int] – The indices in this cut.

apply_cut(*cm*)
Cut a connectivity matrix.

Parameters **cm** (*np.ndarray*) – A connectivity matrix

Returns

np.ndarray –

A copy of the connectivity matrix with connections cut across the cause and effect indices.

cut_matrix()

class pyphi.models.cuts.**Part**
Represents one part of a bipartition.

mechanism
tuple[int] – The nodes in the mechanism for this part.

purview
tuple[int] – The nodes in the mechanism for this part.

Example

When calculating φ of a 3-node subsystem, we partition the system in the following way:

```
mechanism:   A C         B
            -----   X   -----
  purview:    B         A C
```

This class represents one term in the above product.

Create new instance of Part(mechanism, purview)

to_json()

class pyphi.models.cuts.**Bipartition**
A bipartition of a mechanism and purview.

part0
Part – The first part of the partition.

part1
Part – The second part of the partition.

Create new instance of Bipartition(part0, part1)

to_json()

class pyphi.models.cuts.**Tripartition**
Create new instance of Tripartition(part0, part1, part2)

to_json()

network

Represents the network of interest. This is the primary object of PyPhi and the context of all φ and Φ computation.

pyphi.network.**immutable**(*array*)
Make a numpy array immutable.

class pyphi.network.**Network**(*tpm*, *connectivity_matrix=None*, *node_labels=None*, *purview_cache=None*)
A network of nodes.

Represents the network we're analyzing and holds auxilary data about it.

Parameters **tpm** (*np.ndarray*) – The transition probability matrix of the network.

The TPM can be provided in either state-by-node (either 2-D or N-D) or state-by-state form. In either form, row indices must follow the **LOLI** convention (see discussion in the *examples* module.) In state-by-state form column indices must also follow **LOLI** convention.

If given in state-by-node form, the TPM can be either 2-dimensional, so that `tpm[i]` gives the probabilities of each node being on if the past state is encoded by i according to **LOLI**, or in N-D form, so that `tpm[(0, 0, 1)]` gives the probabilities of each node being on if the past state is $\{N_0 = 0, N_1 = 0, N_2 = 1\}$.

The shape of the 2-dimensional form of a state-by-node TPM must be `(S, N)`, and the shape of the N-D form of the TPM must be `[2] * N + [N]`, where `S` is the number of states and N is the number of nodes in the network.

Keyword Arguments

- **connectivity_matrix** (*np.ndarray*) – A square binary adjacency matrix indicating the connections between nodes in the network. `connectivity_matrix[i][j] == 1` means that node i is connected to node j. If no connectivity matrix is given, every node is connected to every node **(including itself)**.
- **node_labels** (*tuple[str]*) – Human readable labels for each node in the network.

Example

In a 3-node network, `a_network.tpm[(0, 0, 1)]` gives the transition probabilities for each node at t_0 given that state at t_{-1} was $\{N_0 = 0, N_1 = 0, N_2 = 1\}$.

tpm
np.ndarray – The network's transition probability matrix, in N-D form.

cm
np.ndarray – The network's connectivity matrix.

A square binary adjacency matrix indicating the connections between nodes in the network.

connectivity_matrix
np.ndarray – Alias for *Network.cm*.

size
int – The number of nodes in the network.

num_states
int – The number of possible states of the network.

node_indices
tuple[int] – The indices of nodes in the network.

This is `0..network.size`.

node_labels
tuple[str] – The labels of nodes in the network.

labels2indices(*labels*)
Convert a tuple of node labels to node indices.

indices2labels(*indices*)
Convert a tuple of node indices to node labels.

parse_node_indices(*nodes*)
Returns the nodes indices for nodes, where `nodes` is either already integer indices or node labels.

__eq__(*other*)
Return whether this network equals the other object.

Two networks are equal if they have the same TPM and CM.

to_json()

classmethod from_json(*json*)

pyphi.network.**irreducible_purviews**(*cm*, *direction*, *mechanism*, *purviews*)
Returns all purview which are irreducible for the mechanism.

Parameters

- **cm** (*np.ndarray*) – A $N \times N$ connectivity matrix.
- **direction** (*Direction*) – *PAST* or *FUTURE*.
- **purviews** (*list[tuple[int]]*) – The purviews to check.
- **mechanism** (*tuple[int]*) – The mechanism in question.

Returns

list[tuple[int]] –

All purviews in purviews which are not reducible over `mechanism`.

Raises `ValueError` – If `direction` is invalid.

pyphi.network.**from_json**(*filename*)
Convert a JSON representation of a network to a PyPhi network.

Parameters **filename** (*str*) – A path to a JSON file representing a network.

Returns |*Network*| – The corresponding PyPhi network object.

node

Represents a node in a subsystem. Each node has a unique index, its position in the network's list of nodes.

class pyphi.node.**Node**(*subsystem*, *index*, *indices=None*, *label=None*)
A node in a subsystem.

subsystem
Subsystem – The subsystem the node belongs to.

index
int – The node's index in the network.

network
Network – The network the node belongs to.

label
str – An optional label for the node.

state
int – The state of this node.

input_indices
The indices of nodes which connect to this node.

output_indices
The indices of nodes that this node connects to.

inputs
The set of nodes with connections to this node.

outputs
The set of nodes this node has connections to.

__eq__(*other*)
Return whether this node equals the other object.

Two nodes are equal if they belong to the same subsystem and have the same index (their TPMs must be the same in that case, so this method doesn't need to check TPM equality).

Labels are for display only, so two equal nodes may have different labels.

to_json()

pyphi.node.**default_label**(*index*)
Default label for a node.

pyphi.node.**default_labels**(*indices*)
Default labels for serveral nodes.

pyphi.node.**generate_nodes**(*subsystem*, *indices=None*, *labels=False*)
Generate the *Node* objects for these indices.

Parameters **subsystem** (Subsystem) – The subsystem for which nodes are being generated.

Keyword Arguments

- **indices** (*tuple[int]*) – Used by *MacroSubsystem* to force generation to use certain indices.
- **labels** (*bool*) – If True, nodes will be labeled with the labels of the network. (This is also used by macro systems to keep labels from being mixed up when many micro elements are combined into one macro element.)

Returns *tuple[|Node|]* – The nodes of the *Subsystem*.

pyphi.node.**expand_node_tpm**(*tpm*)
Broadcast a node TPM over the full network.

This is different from broadcasting the TPM of a full system since the last dimension (containing the state of the node) is unitary – not a state- tuple.

subsystem

Represents a candidate system for φ and Φ evaluation.

class pyphi.subsystem.**Subsystem**(*network*, *state*, *nodes*, *cut=None*, *mice_cache=None*, *repertoire_cache=None*)

A set of nodes in a network.

Parameters

- **network** (Network) – The network the subsystem belongs to.
- **state** (*tuple[int]*) – The state of the network.
- **nodes** (*tuple[int] or tuple[str]*) – The nodes of the network which are in this subsystem. Nodes can be specified either as indices or as labels if the *Network* was passed node_labels.

Keyword Arguments **cut** (Cut) – The unidirectional *Cut* to apply to this subsystem.

network
Network – The network the subsystem belongs to.

tpm
np.ndarray – The TPM conditioned on the state of the external nodes.

cm
np.ndarray – The connectivity matrix after applying the cut.

state
tuple[int] – The state of the network.

nodes
tuple[Node] – The nodes of the subsystem.

node_indices
tuple[int] – The indices of the nodes in the subsystem.

cut
Cut – The cut that has been applied to this subsystem.

cut_matrix
np.ndarray – A matrix of connections which have been severed by the cut.

null_cut
Cut – The cut object representing no cut.

proper_state
tuple[int]) – The state of the subsystem.

proper_state[i] gives the state of the i^{th} node **in the subsystem**. Note that this is **not** the state of nodes[i].

connectivity_matrix
np.ndarray – Alias for Subsystem.cm.

size
int – The number of nodes in the subsystem.

is_cut
bool – True if this Subsystem has a cut applied to it.

cut_indices
tuple[int] – The nodes of this subsystem cut for Φ computations.

This was added to support `MacroSubsystem`, which cuts indices other than `node_indices`.

tpm_size
int – The number of nodes in the TPM.

repertoire_cache_info()
Report repertoire cache statistics.

clear_caches()
Clear the mice and repertoire caches.

__repr__()
Return a representation of this Subsystem.

__str__()
Return this Subsystem as a string.

__eq__(*other*)
Return whether this Subsystem is equal to the other object.

Two Subsystems are equal if their sets of nodes, networks, and cuts are equal.

__bool__()
Return false if the Subsystem has no nodes, true otherwise.

__ne__(*other*)
Return whether this Subsystem is not equal to the other object.

__ge__(*other*)
Return whether this Subsystem >= the other object.

__le__(*other*)
Return whether this Subsystem <= the other object.

__gt__(*other*)
Return whether this Subsystem > the other object.

__lt__(*other*)
Return whether this Subsystem < the other object.

__len__()
Return the number of nodes in this Subsystem.

__hash__()
Return the hash value of this Subsystem.

to_json()
Return this Subsystem as a JSON object.

apply_cut(*cut*)
Return a cut version of this *Subsystem*.

Parameters **cut** (*Cut*) – The cut to apply to this *Subsystem*.

Returns *Subsystem*

indices2nodes(*indices*)
Return nodes for these indices.

Parameters **indices** (*tuple[int]*) – The indices in question.

Returns *tuple[Node]* – The *Node* objects corresponding to these indices.

Raises `ValueError` – If requested indices are not in the subsystem.

indices2labels(*indices*)
Returns the node labels for these indices.

cause_repertoire(*mechanism*, *purview*)
Return the cause repertoire of a mechanism over a purview.

Parameters

- **mechanism** (*tuple[int]*) – The mechanism for which to calculate the cause repertoire.
- **purview** (*tuple[int]*) – The purview over which to calculate the cause repertoire.

Returns *np.ndarray* – The cause repertoire of the mechanism over the purview.

Note: The returned repertoire is a distribution over the nodes in the purview, not the whole network. This is because we never actually need to compare proper cause/effect repertoires, which are distributions over the whole network; we need only compare the purview-repertoires with each other, since cut vs. whole comparisons are only ever done over the same purview.

---

effect_repertoire(*mechanism*, *purview*)
Return the effect repertoire of a mechanism over a purview.

Parameters

- **mechanism** (*tuple[int]*) – The mechanism for which to calculate the effect repertoire.
- **purview** (*tuple[int]*) – The purview over which to calculate the effect repertoire.

Returns *np.ndarray* – The effect repertoire of the mechanism over the purview.

Note: The returned repertoire is a distribution over the nodes in the purview, not the whole network. This is because we never actually need to compare proper cause/effect repertoires, which are distributions over the whole network; we need only compare the purview-repertoires with each other, since cut vs. whole comparisons are only ever done over the same purview.

---

unconstrained_cause_repertoire(*purview*)
Return the unconstrained cause repertoire for a purview.

This is just the cause repertoire in the absence of any mechanism.

unconstrained_effect_repertoire(*purview*)
Return the unconstrained effect repertoire for a purview.

This is just the effect repertoire in the absence of any mechanism.

partitioned_repertoire(*direction*, *partition*)
Compute the repertoire of a partitioned mechanism and purview.

expand_repertoire(*direction*, *repertoire*, *new_purview=None*)
Expand a partial repertoire over a purview to a distribution over a new state space.

Parameters

- **direction** (*Direction*) – *PAST* or *FUTURE*.
- **repertoire** (*np.ndarray*) – A repertoire.

Keyword Arguments **new_purview** (*tuple[int]*) – The purview to expand the repertoire over. Defaults to the entire subsystem.

Returns *np.ndarray* – The expanded repertoire.

Raises `ValueError` – If the expanded purview doesn't contain the original purview.

expand_cause_repertoire(*repertoire*, *new_purview=None*)
Expand a partial cause repertoire over a purview to a distribution over the entire subsystem's state space.

expand_effect_repertoire(*repertoire*, *new_purview=None*)
Expand a partial effect repertoire over a purview to a distribution over the entire subsystem's state space.

cause_info(*mechanism*, *purview*)
Return the cause information for a mechanism over a purview.

effect_info(*mechanism*, *purview*)
Return the effect information for a mechanism over a purview.

cause_effect_info(*mechanism*, *purview*)
Return the cause-effect information for a mechanism over a purview.

This is the minimum of the cause and effect information.

evaluate_partition(*direction*, *mechanism*, *purview*, *partition*, *unpartitioned_repertoire=None*)
Return the φ of a mechanism over a purview for the given partition.

Parameters

- **direction** (`Direction`) – *PAST* or *FUTURE*.
- **mechanism** (*tuple[int]*) – The nodes in the mechanism.
- **purview** (*tuple[int]*) – The nodes in the purview.
- **partition** (`Bipartition`) – The partition to evaluate.

Keyword Arguments **unpartitioned_repertoire** (*np.array*) – The unpartitioned repertoire. If not supplied, it will be computed.

Returns *tuple[phi, partitioned_repertoire]* – The distance between the unpartitioned and partitioned repertoires, and the partitioned repertoire.

find_mip(*direction*, *mechanism*, *purview*)
Return the minimum information partition for a mechanism over a purview.

Parameters

- **direction** (`Direction`) – *PAST* or *FUTURE*.
- **mechanism** (*tuple[int]*) – The nodes in the mechanism.
- **purview** (*tuple[int]*) – The nodes in the purview.

Returns *|Mip|* – The mininum-information partition in one temporal direction.

mip_past(*mechanism*, *purview*)
Return the past minimum information partition.

Alias for *find_mip()* with `direction` set to `Direction.FUTURE`.

mip_future(*mechanism*, *purview*)
Return the future minimum information partition.

Alias for *find_mip()* with `direction` set to `DIRECTIONS[FUTURE]`.

phi_mip_past(*mechanism*, *purview*)
Return the φ of the past minimum information partition.

This is the distance between the unpartitioned cause repertoire and the MIP cause repertoire.

phi_mip_future (*mechanism*, *purview*)
Return the φ of the future minimum information partition.

This is the distance between the unpartitioned effect repertoire and the MIP cause repertoire.

phi (*mechanism*, *purview*)
Return the φ of a mechanism over a purview.

find_mice (*direction*, *mechanism*, *purviews=False*)
Return the maximally irreducible cause or effect for a mechanism.

Parameters

- **direction** (Direction) – *PAST* or *FUTURE*.
- **mechanism** (*tuple[int]*) – The mechanism to be tested for irreducibility.

Keyword Arguments **purviews** (*tuple[int]*) – Optionally restrict the possible purviews to a subset of the subsystem. This may be useful for _e.g._ finding only concepts that are "about" a certain subset of nodes.

Returns |*Mice*| – The maximally-irreducible cause or effect in one temporal direction.

Note: Strictly speaking, the MICE is a pair of repertoires: the core cause repertoire and core effect repertoire of a mechanism, which are maximally different than the unconstrained cause/effect repertoires (*i.e.*, those that maximize φ). Here, we return only information corresponding to one direction, `Direction.PAST` or `Direction.FUTURE`, i.e., we return a core cause or core effect, not the pair of them.

---

core_cause (*mechanism*, *purviews=False*)
Return the core cause repertoire of a mechanism.

Alias for *find_mice()* with `direction` set to *PAST*.

core_effect (*mechanism*, *purviews=False*)
Return the core effect repertoire of a mechanism.

Alias for *find_mice()* with `direction` set to *FUTURE*.

phi_max (*mechanism*)
Return the $\varphi^{\max}$ of a mechanism.

This is the maximum of φ taken over all possible purviews.

null_concept
Return the null concept of this subsystem.

The null concept is a point in concept space identified with the unconstrained cause and effect repertoire of this subsystem.

concept (*mechanism*, *purviews=False*, *past_purviews=False*, *future_purviews=False*)
Calculate a concept.

See *pyphi.compute.concept()* for more information.

pyphi.subsystem.**mip_bipartitions** (*mechanism*, *purview*)
Return an generator of all φ bipartitions of a mechanism over a purview.

Excludes all bipartitions where one half is entirely empty, e.g:

```
 A     []
--- X --
 B     []
```

is not valid, but

```
A     []
-- X --
[]    B
```

is.

Parameters

- **mechanism** (*tuple[int]*) – The mechanism to partition
- **purview** (*tuple[int]*) – The purview to partition

Yields *Bipartition* – Where each bipartition is

```
bipart[0].mechanism   bipart[1].mechanism
------------------- X -------------------
bipart[0].purview     bipart[1].purview
```

Example

```
>>> mechanism = (0,)
>>> purview = (2, 3)
>>> for partition in mip_bipartitions(mechanism, purview):
...     print(partition, "\n")
[]   0
-- X -
2    3

[]   0
-- X -
3    2

[]    0
--- X --
2,3   []
```

pyphi.subsystem.**wedge_partitions**(*mechanism*, *purview*)

Return an iterator over all wedge partitions.

These are partitions which strictly split the mechanism and allow a subset of the purview to be split into a third partition, eg:

```
A     B    []
-- X - X --
B     C    D
```

See pyphi.config.PARTITION_MECHANISMS for more information.

Parameters

- **mechanism** (*tuple[int]*) – A mechanism.
- **purview** (*tuple[int]*) – A purview.

Yields *Tripartition* – all unique tripartitions of this mechanism and purview.

pyphi.subsystem.**effect_emd**(*d1*, *d2*)

Compute the EMD between two effect repertoires.

Billy's synopsis: Because the nodes are independent, the EMD between effect repertoires is equal to the sum of the EMDs between the marginal distributions of each node, and the EMD between marginal distribution for a node is the absolute difference in the probabilities that the node is off.

Parameters

- **d1** (*np.ndarray*) – The first repertoire.
- **d2** (*np.ndarray*) – The second repertoire.

Returns *float* – The EMD between `d1` and `d2`.

pyphi.subsystem.**emd**(*direction*, *d1*, *d2*)
Compute the EMD between two repertoires for a given direction.

The full EMD computation is used for cause repertoires. A fast analytic solution is used for effect repertoires.

Parameters

- **direction** (`Direction`) – *PAST* or *FUTURE*.
- **d1** (*np.ndarray*) – The first repertoire.
- **d2** (*np.ndarray*) – The second repertoire.

Returns *float* – The EMD between `d1` and `d2`, rounded to `constants.PRECISION`.

Raises `ValueError` – If `direction` is invalid.

pyphi.subsystem.**measure**(*direction*, *d1*, *d2*)
Compute the distance between two repertoires for the given direction.

Parameters

- **direction** (`Direction`) – *PAST* or *FUTURE*.
- **d1** (*np.ndarray*) – The first repertoire.
- **d2** (*np.ndarray*) – The second repertoire.

Returns *float* – The distance between `d1` and `d2`, rounded to `constants.PRECISION`.

utils

Functions used by more than one PyPhi module or class, or that might be of external use.

pyphi.utils.**state_of**(*nodes*, *network_state*)
Return the state-tuple of the given nodes.

pyphi.utils.**all_states**(*n*)
Return all binary states for a system.

Parameters **n** (*int*) – The number of elements in the system.

Yields *tuple[int]* – The next state of an `n`-element system, in LOLI order.

pyphi.utils.**sparse**(*matrix*, *threshold=0.1*)

pyphi.utils.**sparse_time**(*tpm*, *time_scale*)

pyphi.utils.**dense_time**(*tpm*, *time_scale*)

pyphi.utils.**run_tpm**(*tpm*, *time_scale*)
Iterate a TPM by the specified number of time steps.

Parameters

- **tpm** (*np.ndarray*) – A state-by-node tpm.
- **time_scale** (*int*) – The number of steps to run the tpm.

Returns np.ndarray

pyphi.utils.**run_cm**(*cm*, *time_scale*)
Iterate a connectivity matrix the specified number of steps.

Parameters

- **cm** (*np.ndarray*) – A $N \times N$ connectivity matrix
- **time_scale** (*int*) – The number of steps to run.

Returns np.ndarray

pyphi.utils.**state_by_state**(*tpm*)
Return `True` if `tpm` is in state-by-state form, otherwise `False`.

pyphi.utils.**condition_tpm**(*tpm*, *fixed_nodes*, *state*)
Return a TPM conditioned on the given fixed node indices, whose states are fixed according to the given state-tuple.

The dimensions of the new TPM that correspond to the fixed nodes are collapsed onto their state, making those dimensions singletons suitable for broadcasting. The number of dimensions of the conditioned TPM will be the same as the unconditioned TPM.

pyphi.utils.**expand_tpm**(*tpm*)
Broadcast a state-by-node TPM so that singleton dimensions are expanded over the full network.

pyphi.utils.**fully_connected**(*cm*, *nodes1*, *nodes2*)
Test connectivity of one set of nodes to another.

Parameters

- **cm** (`np.ndarrray`) – The connectivity matrix
- **nodes1** (*tuple[int]*) – The nodes whose outputs to `nodes2` will be tested.
- **nodes2** (*tuple[int]*) – The nodes whose inputs from `nodes1` will be tested.

Returns *bool* – Returns `True` if all elements in `nodes1` output to some element in `nodes2` AND all elements in `nodes2` have an input from some element in `nodes1`. Otherwise return `False`. Return `True` if either set of nodes is empty.

pyphi.utils.**apply_boundary_conditions_to_cm**(*external_indices*, *cm*)
Return a connectivity matrix with all connections to or from external nodes removed.

pyphi.utils.**get_inputs_from_cm**(*index*, *cm*)
Return a tuple of node indices that have connections to the node with the given index.

pyphi.utils.**get_outputs_from_cm**(*index*, *cm*)
Return a tuple of node indices that the node with the given index has connections to.

pyphi.utils.**causally_significant_nodes**(*cm*)
Return a tuple of all nodes indices in the connectivity matrix which are causally significant (have inputs and outputs).

pyphi.utils.**np_hash**(*a*)
Return a hash of a NumPy array.

pyphi.utils.**phi_eq**(*x*, *y*)
Compare two phi values up to `constants.PRECISION`.

pyphi.utils.**normalize**(*a*)
Normalize a distribution.

Parameters **a** (*np.ndarray*) – The array to normalize.

Returns *np.ndarray* – `a` normalized so that the sum of its entries is 1.

pyphi.utils.**combs**(*a*, *r*)
NumPy implementation of itertools.combinations.

Return successive r-length combinations of elements in the array `a`.

Parameters

- **a** (*np.ndarray*) – The array from which to get combinations.
- **r** (*int*) – The length of the combinations.

Returns *np.ndarray* – An array of combinations.

pyphi.utils.**comb_indices**(*n*, *k*)
N-D version of itertools.combinations.

Parameters

- **a** (*np.ndarray*) – The array from which to get combinations.
- **k** (*int*) – The desired length of the combinations.

Returns *np.ndarray* – Indices that give the k-combinations of n elements.

Example

```
>>> n, k = 3, 2
>>> data = np.arange(6).reshape(2, 3)
>>> data[:, comb_indices(n, k)]
array([[[0, 1],
        [0, 2],
        [1, 2]],

       [[3, 4],
        [3, 5],
        [4, 5]]])
```

pyphi.utils.**powerset**(*iterable*)
Return the power set of an iterable (see itertools recipes).

Parameters **iterable** (*Iterable*) – The iterable from which to generate the power set.

Returns *generator* – An chained generator over the power set.

Example

```
>>> ps = powerset(np.arange(2))
>>> print(list(ps))
[(), (0,), (1,), (0, 1)]
```

pyphi.utils.**uniform_distribution**(*number_of_nodes*)
Return the uniform distribution for a set of binary nodes, indexed by state (so there is one dimension per node, the size of which is the number of possible states for that node).

Parameters **nodes** (*np.ndarray*) – A set of indices of binary nodes.

Returns *np.ndarray* – The uniform distribution over the set of nodes.

pyphi.utils.**marginalize_out**(*indices*, *tpm*)
Marginalize out a node from a TPM.

Parameters

- **indices** (*list[int]*) – The indices of nodes to be marginalized out.
- **tpm** (*np.ndarray*) – The TPM to marginalize the node out of.

Returns *np.ndarray* – A TPM with the same number of dimensions, with the nodes marginalized out.

pyphi.utils.**marginal_zero**(*repertoire*, *node_index*)
Return the marginal probability that the node is off.

pyphi.utils.**marginal**(*repertoire*, *node_index*)
Get the marginal distribution for a node.

pyphi.utils.**independent**(*repertoire*)
Check whether the repertoire is independent.

pyphi.utils.**purview**(*repertoire*)
The purview of the repertoire.

Parameters **repertoire** (*np.ndarray*) – A repertoire

Returns *tuple[int]* – The purview that the repertoire was computed over.

pyphi.utils.**purview_size**(*repertoire*)
Return the size of the purview of the repertoire.

Parameters **repertoire** (*np.ndarray*) – A repertoire

Returns *int* – The size of purview that the repertoire was computed over.

pyphi.utils.**repertoire_shape**(*purview*, *N*)
Return the shape a repertoire.

Parameters

- **purview** (*tuple[int]*) – The purview over which the repertoire is computed.
- **N** (*int*) – The number of elements in the system.

Returns *list[int]* – The shape of the repertoire. Purview nodes have two dimensions and non-purview nodes are collapsed to a unitary dimension.

Example

```
>>> purview = (0, 2)
>>> N = 3
>>> repertoire_shape(purview, N)
[2, 1, 2]
```

pyphi.utils.**max_entropy_distribution**(*node_indices*, *number_of_nodes*)
Return the maximum entropy distribution over a set of nodes.

This is different from the network's uniform distribution because nodes outside `node_indices` are fixed and treated as if they have only 1 state.

Parameters

- **node_indices** (*tuple[int]*) – The set of node indices over which to take the distribution.
- **number_of_nodes** (*int*) – The total number of nodes in the network.

Returns *np.ndarray* – The maximum entropy distribution over the set of nodes.

pyphi.utils.**hamming_emd**(*d1*, *d2*)

Return the Earth Mover's Distance between two distributions (indexed by state, one dimension per node).

Singleton dimensions are sqeezed out.

pyphi.utils.**l1**(*d1*, *d2*)

Return the L1 distance between two distributions.

Parameters

- **d1** (*np.ndarray*) – The first distribution.
- **d2** (*np.ndarray*) – The second distribution.

Returns *float* – The sum of absolute differences of `d1` and `d2`.

pyphi.utils.**kld**(*d1*, *d2*)

Return the Kullback-Leibler Divergence (KLD) between two distributions.

Parameters

- **d1** (*np.ndarray*) – The first distribution.
- **d2** (*np.ndarray*) – The second distribution.

Returns *float* – The KLD of `d1` from `d2`.

pyphi.utils.**bipartition**(*a*)

Return a list of bipartitions for a sequence.

Parameters **a** (*Iterable*) – The iterable to partition.

Returns *list[tuple[tuple]]* – A list of tuples containing each of the two partitions.

Example

```
>>> bipartition((1,2,3))
[((), (1, 2, 3)), ((1,), (2, 3)), ((2,), (1, 3)), ((1, 2), (3,))]
```

pyphi.utils.**directed_bipartition**(*a*)

Return a list of directed bipartitions for a sequence.

Parameters **a** (*Iterable*) – The iterable to partition.

Returns *list[tuple[tuple]]* – A list of tuples containing each of the two partitions.

Example

```
>>> directed_bipartition((1, 2, 3))
[((), (1, 2, 3)),
 ((1,), (2, 3)),
 ((2,), (1, 3)),
 ((1, 2), (3,)),
```

```
 ((3,), (1, 2)),
 ((1, 3), (2,)),
 ((2, 3), (1,)),
 ((1, 2, 3), ())]
```

pyphi.utils.**directed_bipartition_of_one**(*a*)

Return a list of directed bipartitions for a sequence where each bipartitions includes a set of size 1.

Parameters **a** (*Iterable*) – The iterable to partition.

Returns *list[tuple[tuple]]* – A list of tuples containing each of the two partitions.

Example

```
>>> directed_bipartition_of_one((1,2,3))
[((1,), (2, 3)),
 ((2,), (1, 3)),
 ((1, 2), (3,)),
 ((3,), (1, 2)),
 ((1, 3), (2,)),
 ((2, 3), (1,))]
```

pyphi.utils.**directed_bipartition_indices**(*N*)

Return indices for directed bipartitions of a sequence.

Parameters **N** (*int*) – The length of the sequence.

Returns *list* – A list of tuples containing the indices for each of the two partitions.

Example

```
>>> N = 3
>>> directed_bipartition_indices(N)
[((), (0, 1, 2)),
 ((0,), (1, 2)),
 ((1,), (0, 2)),
 ((0, 1), (2,)),
 ((2,), (0, 1)),
 ((0, 2), (1,)),
 ((1, 2), (0,)),
 ((0, 1, 2), ())]
```

pyphi.utils.**bipartition_indices**(*N*)

Return indices for undirected bipartitions of a sequence.

Parameters **N** (*int*) – The length of the sequence.

Returns *list* – A list of tuples containing the indices for each of the two partitions.

Example

```
>>> N = 3
>>> bipartition_indices(N)
[((), (0, 1, 2)), ((0,), (1, 2)), ((1,), (0, 2)), ((0, 1), (2,))]
```

pyphi.utils.**directed_tripartition_indices**(*N*)
Return indices for directed tripartitions of a sequence.

Parameters **N** (*int*) – The length of the sequence.

Returns *list[tuple]* – A list of tuples containing the indices for each partition.

Example

```
>>> N = 1
>>> directed_tripartition_indices(N)
[((0,), (), ()), ((), (0,), ()), ((), (), (0,))]
```

pyphi.utils.**directed_tripartition**(*seq*)
Generator over all directed tripartitions of a sequence.

Parameters **seq** (*Iterable*) – a sequence.

Yields *tuple[tuple]* – A tripartition of `seq`.

Example

```
>>> seq = (2, 5)
>>> list(directed_tripartition(seq))
[((2, 5), (), ()),
 ((2,), (5,), ()),
 ((2,), (), (5,)),
 ((5,), (2,), ()),
 ((), (2, 5), ()),
 ((), (2,), (5,)),
 ((5,), (), (2,)),
 ((), (5,), (2,)),
 ((), (), (2, 5))]
```

pyphi.utils.**load_data**(*dir*, *num*)
Load numpy data from the data directory.

The files should stored in `data/{dir}` and named `0.npy, 1.npy, ... {num - 1}.npy`.

Returns *list* – A list of loaded data, such that `list[i]` contains the the contents of `i.npy`.

pyphi.utils.**relevant_connections**(*n*, *_from*, *to*)
Construct a connectivity matrix.

Parameters

- **n** (*int*) – The dimensions of the matrix
- **_from** (*tuple[int]*) – Nodes with outgoing connections to `to`
- **to** (*tuple[int]*) – Nodes with incoming connections from `_from`

Returns *np.ndarray* – An $N \times N$ connectivity matrix with the i, j^{th} entry set to `1` if i is in `_from` and j is in `to`.

pyphi.utils.**block_cm**(*cm*)
Return whether `cm` can be arranged as a block connectivity matrix.

If so, the corresponding mechanism/purview is trivially reducible. Technically, only square matrices are "block diagonal", but the notion of connectivity carries over.

We test for block connectivity by trying to grow a block of nodes such that:

- 'source' nodes only input to nodes in the block
- 'sink' nodes only receive inputs from source nodes in the block

For example, the following connectivity matrix represents connections from `nodes1 = A, B, C` to `nodes2 = D, E, F, G` (without loss of generality—note that `nodes1` and `nodes2` may share elements):

```
   D  E  F  G
A [1, 1, 0, 0]
B [1, 1, 0, 0]
C [0, 0, 1, 1]
```

Since nodes AB only connect to nodes DE, and node C only connects to nodes FG, the subgraph is reducible; the cut

```
AB   C
-- X --
DE   FG
```

does not change the structure of the graph.

pyphi.utils.**block_reducible**(*cm*, *nodes1*, *nodes2*)
Return whether connections from `nodes1` to `nodes2` are reducible.

Parameters

- **cm** (*np.ndarray*) – The network's connectivity matrix.
- **nodes1** (*tuple[int]*) – Source nodes
- **nodes2** (*tuple[int]*) – Sink nodes

pyphi.utils.**strongly_connected**(*cm*, *nodes=None*)
Return whether the connectivity matrix is strongly connected.

Parameters **cm** (*np.ndarray*) – A square connectivity matrix.

Keyword Arguments **nodes** (*tuple[int]*) – An optional subset of node indices to test strong connectivity over.

pyphi.utils.**weakly_connected**(*cm*, *nodes=None*)
Return whether the connectivity matrix is weakly connected.

Parameters **cm** (*np.ndarray*) – A square connectivity matrix.

Keyword Arguments **nodes** (*tuple[int]*) – An optional subset of node indices to test weak connectivity over.

pyphi.utils.**print_repertoire**(*r*)
Print a vertical, human-readable cause/effect repertoire.

pyphi.utils.**print_repertoire_horiz**(*r*)
Print a horizontal, human-readable cause/effect repertoire.

validate

Methods for validating common types of input.

pyphi.validate.**direction**(*direction*)
Validate that the given direction is one of the allowed constants.

pyphi.validate.**tpm**(*tpm*)
Validate a TPM.

The TPM can be in

- 2-D state-by-state form,
- 2-D state-by-node form, or
- N-D state-by-node form.

pyphi.validate.**conditionally_independent**(*tpm*)
Validate that the TPM is conditionally independent.

pyphi.validate.**connectivity_matrix**(*cm*)
Validate the given connectivity matrix.

pyphi.validate.**node_labels**(*node_labels*, *node_indices*)
Validate that there is a label for each node.

pyphi.validate.**network**(*n*)
Validate a *Network*.

Checks the TPM and connectivity matrix.

pyphi.validate.**is_network**(*network*)
Validate that the argument is a *Network*.

pyphi.validate.**node_states**(*state*)
Check that the state contains only zeros and ones.

pyphi.validate.**state_length**(*state*, *size*)
Check that the state is the given size.

pyphi.validate.**state_reachable**(*subsystem*)
Return whether a state can be reached according to the network's TPM.

pyphi.validate.**cut**(*cut*, *node_indices*)
Check that the cut is for only the given nodes.

pyphi.validate.**subsystem**(*s*)
Validate a *Subsystem*.

Checks its state and cut.

pyphi.validate.**time_scale**(*time_scale*)
Validate a macro temporal time scale.

pyphi.validate.**partition**(*partition*)
Validate a partition - used by blackboxes and coarse grains.

pyphi.validate.**coarse_grain**(*coarse_grain*)
Validate a macro coarse-graining.

pyphi.validate.**blackbox**(*blackbox*)
Validate a macro blackboxing.

pyphi.validate.**blackbox_and_coarse_grain**(*blackbox*, *coarse_grain*)
Validate that a coarse-graining properly combines the outputs of a blackboxing.

pyphi.validate.**measure**(*value*)
Validate a distance measure.

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W

X