PyPhi Documentation

Release v0.8.1

Will Mayner

June 02, 2016

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.

1.1 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.

1.1.1 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
(Part(mechanism=(0,), purview=()), Part(mechanism=(1, 2), purview=(0, 1, 2)))
>>> mip_e.phi
0.25
>>> mip_e.partition
(Part(mechanism=(), purview=(1,)), Part(mechanism=(0, 1, 2), purview=(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 the 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)
2.111109
```

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
1.916665
>>> 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), 1.916665)
>>> (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()`.

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

1.1.3 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
```

1.1.4 Emergence (Macro/Micro)

- *pyphi.examples.macro_network()*

For this example, we will use the `pprint` module to display lists in a way which makes them easer to read.

```
>>> from pprint import pprint
```

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 partitions:

```
>>> partitions = pyphi.macro.list_all_partitions(network)
>>> pprint(partitions)
[[[0, 1, 2], [3]],
 [[0, 1, 3], [2]],
 [[0, 1], [2, 3]],
 [[0, 1], [2], [3]],
 [[0, 2, 3], [1]],
 [[0, 2], [1, 3]],
 [[0, 2], [1], [3]],
 [[0, 3], [1, 2]],
 [[0], [1, 2, 3]],
 [[0], [1, 2], [3]],
 [[0, 3], [1], [2]],
 [[0], [1, 3], [2]],
 [[0], [1], [2, 3]],
 [[0, 1, 2, 3]]]
```

Lets start by considering the partition `[[0, 1, 2], [3]]`:

```
>>> partition = partitions[0]
>>> partition
[[0, 1, 2], [3]]
```

For this partition there are two macro-elements, one consisting 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. An assumption when coarse-graining the system, is that the resulting macro-elements do not differentiate the different micro-elements. Thus any correspondence between states must be stated soley in terms of the number of micro-elements which are on, and not depend on which micro-element 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.

To see a list of all possible groupings of micro-states into macro-states:

```
>>> groupings = pyphi.macro.list_all_groupings(partition)
>>> pprint(groupings)
[[[[0, 1, 2], [3]], [[0], [1]]],
 [[[0, 1, 3], [2]], [[0], [1]]],
 [[[0, 1], [2, 3]], [[0], [1]]],
 [[[0, 2, 3], [1]], [[0], [1]]],
 [[[0, 2], [1, 3]], [[0], [1]]],
 [[[0, 3], [1, 2]], [[0], [1]]],
 [[[0], [1, 2, 3]], [[0], [1]]]]
```

We will focus on the first grouping in the list.

```
>>> grouping = groupings[0]
>>> grouping
[[[0, 1, 2], [3]], [[0], [1]]]
```

The grouping contains 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 = pyphi.macro.make_mapping(partition, grouping)
>>> 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]
1.0
```

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**.

Using the mapping, we can then create a state-by-state TPM for the macro-system corresponding to the selected partition and grouping:

```
>>> macro_tpm = pyphi.macro.make_macro_tpm(network.tpm, mapping)
>>> macro_tpm
array([[ 0.5838,  0.0162,  0.3802,  0.0198],
       [ 0.    ,  0.    ,  0.91  ,  0.09  ],
       [ 0.5019,  0.0981,  0.3451,  0.0549],
       [ 0.    ,  0.    ,  0.    ,  1.    ]])
```

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:

```
>>> pyphi.validate.conditionally_independent(macro_tpm)
False
```

In these cases, the object returned *make_macro_network()* function will have a boolean value of `False`:

```
>>> (macro_network, macro_state) = pyphi.macro.make_macro_network(network, state, mapping)
>>> bool(macro_network)
False
```

Lets consider a different partition instead.

```
>>> partition = partitions[2]
>>> partition
[[0, 1], [2, 3]]
```

```
>>> groupings = pyphi.macro.list_all_groupings(partition)
>>> grouping = groupings[0]
>>> grouping
[[[0, 1], [2]], [[0, 1], [2]]]
```

```
>>> mapping = pyphi.macro.make_mapping(partition, grouping)
>>> mapping
array([ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.,  2.,
        2.,  2.,  3.])
```

```
>>> (macro_network, macro_state) = pyphi.macro.make_macro_network(network, state, mapping)
>>> bool(macro_network)
True
```

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

```
>>> macro_main_complex = pyphi.compute.main_complex(macro_network, macro_state)
>>> macro_main_complex.phi
0.86905
```

The integrated information of the macro system ($\Phi = 0.86905$) 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.partition
[[0, 1], [2, 3]]
>>> M.grouping
[[[0, 1], [2]], [[0, 1], [2]]]
>>> M.emergence
0.755161
```

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.

1.1.5 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, a third order concept is created in the partitioned constellation. A new concept is created as a result of the cut, but there is also a concept destroyed **|A,B|** and 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
0.624999
```

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 **|n0, n1, n4|** 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 **|n0, n4|** 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.

1.1.6 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
(Part(mechanism=(), purview=(2,)), Part(mechanism=(0, 1), purview=(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
(Part(mechanism=(), purview=(4,)), Part(mechanism=(0,), purview=(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
(Part(mechanism=(), purview=(2,)), Part(mechanism=(0,), purview=(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 $\varphi = \frac{1}{10}$.

1.1.7 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
(Part(mechanism=(0,), purview=()), Part(mechanism=(1, 2), purview=(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.

2.1 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'})
```

2.1.1 Theoretical approximations

This section with deals 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
```

2.1.2 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
```

2.1.3 Caching

PyPhi is equipped with a transparent caching system for the `BigMip` and `Concept` 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
```

Note: Concept caching only has an effect when a database is used as the the caching backend.

- `pyphi.config.NORMALIZE_TPMS`: Control whether TPMs should be normalized as part of concept normalization. TPM normalization increases the chances that a precomputed concept can be used again, but is expensive.

```
>>> defaults['NORMALIZE_TPMS']
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
```

2.1.4 Logging

These are the settings for PyPhi logging. You can control the format of the logs and the name of the log file. Logs can be written to standard output, a file, both, or none. See the documentation on Python's logger for more information.

- `pyphi.config.LOGGING_CONFIG['file']['enabled']`: Control whether logs are written to a file.

```
>>> defaults['LOGGING_CONFIG']['file']['enabled']
True
```

- `pyphi.config.LOGGING_CONFIG['file']['filename']`: Control the name of the logfile.

```
>>> defaults['LOGGING_CONFIG']['file']['filename']
'pyphi.log'
```

- `pyphi.config.LOGGING_CONFIG['file']['level']`: Control the concern level of file logging. Can be one of `'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, or `'CRITICAL'`.

```
>>> defaults['LOGGING_CONFIG']['file']['level']
'INFO'
```

- `pyphi.config.LOGGING_CONFIG['stdout']['enabled']`: Control whether logs are written to standard output.

```
>>> defaults['LOGGING_CONFIG']['stdout']['enabled']
True
```

- `pyphi.config.LOGGING_CONFIG['stdout']['level']`: Control the concern level of standard output logging. Same possible values as file logging.

```
>>> defaults['LOGGING_CONFIG']['stdout']['level']
'WARNING'
```

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

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

2.1.5 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
```

2.1.6 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.READABLE_REPRS`: If set to `True`, this causes `repr` calls on PyPhi objects to return pretty-formatted and legible strings. 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 `False`, `repr` returns more traditional object representations.

```
>>> defaults['READABLE_REPRS']
True
```

---

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.

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.

3.1 Conventions

3.1.1 Connectivity Matrices

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

3.1.2 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.

API Reference

4.1 API Reference

PyPhi API documentation, autogenerated from the source code.

4.1.1 compute

Maintains backwards compatability with the old compute API.

4.1.2 concept_caching

Objects and functions for managing the normalization, caching, and retrieval of concepts.

> **Warning:** Concept caching is disabled and likely broken. Use at your own risk!

class pyphi.concept_caching.**NormalizedMechanism**(*mechanism*, *subsystem*, *normalize_tpms=True*)

A mechanism rendered into a normal form, suitable for use as a cache key in concept memoization.

The broad outline for the normalization procedure is as follows:

- Get the set of all nodes that input to (output from) at least one mechanism node.
- Sort the Marbls in a stable way (this is done by marbl-python, when the MarblSet is initialized).
- Iterate over the sorted Marbls; for each one, iterate over its corresponding mechanism node's inputs (outputs).
- Label each input (output) with a unique integer. These are the "normalized indices" of the inputs (outputs).
- Record the inverse mapping, which sends a normalized index to a real index.
- Record the state of the mechanism and all input/ouptut nodes.

Then two normalized mechanisms are the same if they have the same MarblSets, inputs, outputs, state, and input/output state.

marblset

MarblSet – A dictionary where keys are directions, and values are MarblSets containing Marbls generated from the TPMs of the mechanism nodes' corresponding to the direction.

normalized_indices
dict – A dictionary where keys are directions, and values are dictionaries mapping mechanism node indices to their normalized indices for that direction.

unnormalized_indices
dict – The inverse of `normalized_indices`.

inputs
tuple(tuple(int – A tuple where the i^{th} element contains a tuple of the normalized indices of the i^{th} node, where i is a normalized index.

outputs
tuple(tuple(int – The same as `inputs`, but the elements contain normalized indices of the outputs.

permutation
dict(– A dictionary where the keys are directions and the values are the permutations that maps mechanism nodes to the position of their marbl in the marblset for that direction.

class `pyphi.concept_caching.`**NormalizedMice**
A lightweight container for MICE data.

See documentation for `Mice` for its unnormalized counterpart.

phi
float – The difference between the mechanism's unpartitioned and partitioned repertoires.

direction
str – Either 'past' or 'future'. If 'past' ('future'), this represents a maximally irreducible cause (effect).

mechanism
tuple(int – The normalized indices of the MICE's mechanism.

purview
tuple(int – A normalized purview. This is a tuple of the normalized indices of its nodes.

repertoire
np.ndarray – The normalized unpartitioned repertoire of the mechanism over the purview. A repertoire is normalized by squeezing and then reordering its dimensions so they correspond to the normalized purview.

class `pyphi.concept_caching.`**NormalizedConcept**(*normalized_mechanism*, *concept*)
A precomputed concept in a form suitable for memoization.

Upon initialization, the normal form of the concept to be cached is computed, and data relating its cause and effect purviews are stored, which the concept to be reconstituted in a different subsystem.

mechanism
NormalizedMechanism – The mechanism the concept consists of.

phi
float – The φ value of the concept.

cause
NormalizedMice – The concept's normalized core cause.

effect
NormalizedMice – The concept's normalized core effect.

`pyphi.concept_caching.`**concept**(*subsystem*, *mechanism*)
Find the concept specified by a mechanism, returning a cached value if one is found and computing and caching it otherwise.

4.1.3 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 with deals 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
```

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 the `BigMip` and `Concept` 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
```

Note: Concept caching only has an effect when a database is used as the the caching backend.

- `pyphi.config.NORMALIZE_TPMS`: Control whether TPMs should be normalized as part of concept normalization. TPM normalization increases the chances that a precomputed concept can be used again, but is expensive.

```
>>> defaults['NORMALIZE_TPMS']
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 are the settings for PyPhi logging. You can control the format of the logs and the name of the log file. Logs can be written to standard output, a file, both, or none. See the documentation on Python's logger for more information.

- `pyphi.config.LOGGING_CONFIG['file']['enabled']`: Control whether logs are written to a file.

```
>>> defaults['LOGGING_CONFIG']['file']['enabled']
True
```

- `pyphi.config.LOGGING_CONFIG['file']['filename']`: Control the name of the logfile.

```
>>> defaults['LOGGING_CONFIG']['file']['filename']
'pyphi.log'
```

- `pyphi.config.LOGGING_CONFIG['file']['level']`: Control the concern level of file logging. Can be one of `'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, or `'CRITICAL'`.

```
>>> defaults['LOGGING_CONFIG']['file']['level']
'INFO'
```

- `pyphi.config.LOGGING_CONFIG['stdout']['enabled']`: Control whether logs are written to standard output.

```
>>> defaults['LOGGING_CONFIG']['stdout']['enabled']
True
```

- `pyphi.config.LOGGING_CONFIG['stdout']['level']`: Control the concern level of standard output logging. Same possible values as file logging.

```
>>> defaults['LOGGING_CONFIG']['stdout']['level']
'WARNING'
```

- `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.READABLE_REPRS`: If set to `True`, this causes `repr` calls on PyPhi objects to return pretty-formatted and legible strings. 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 `False`, `repr` returns more traditional object representations.

```
>>> defaults['READABLE_REPRS']
True
```

`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.

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.

4.1.4 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

holi_index –

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

Return type int

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

loli_index –

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

Return type int

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

state –

A state-tuple where the i^{th} element of the tuple gives the state of the i^{th} node.

Return type ‘‘tuple(int

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

state –

A state-tuple where the i^{th} element of the tuple gives the state of the i^{th} node.

Return type ‘‘tuple(int

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)*) – A square state-by-state TPM with row and column indices following the **LOLI** convention.

Returns

state_by_node_tpm –

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

Return type np.ndarray

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)*) – A state-by-node TPM with row indices following the **LOLI** convention and column indices following the **HOLI** convention.

Returns

state_by_state_tpm –

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

Return type np.ndarray

```
>>> 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.]])
```

4.1.5 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.

4.1.6 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**()

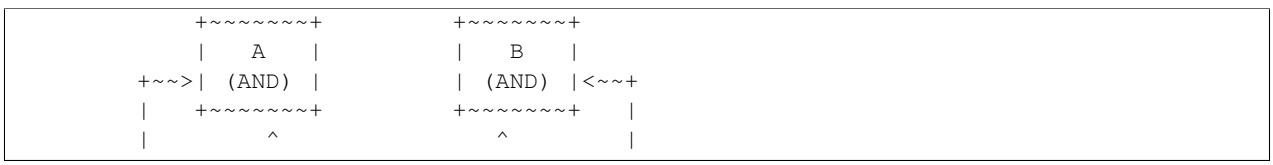
A subsystem containing all the nodes of the *basic_network()*.

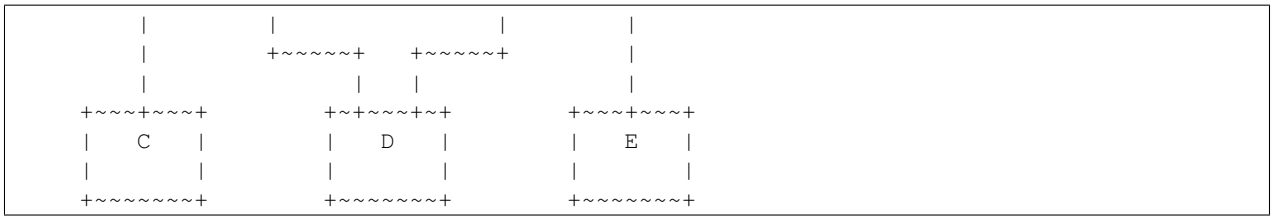
pyphi.examples.**residue_network**()

The network for the residue example.

Current and past state are all nodes off.

Diagram:





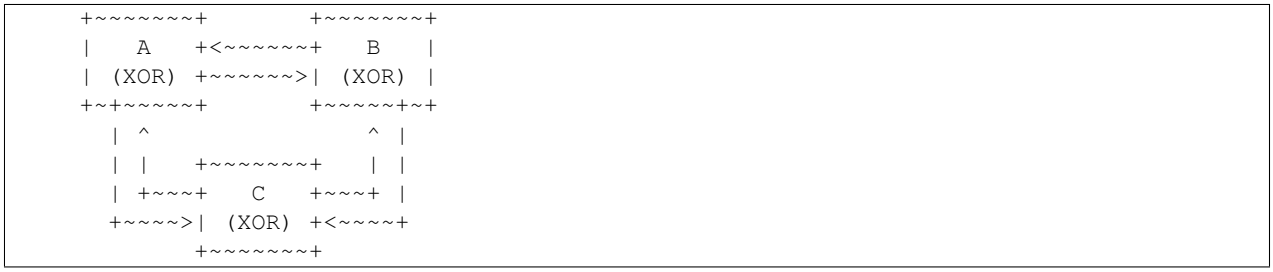
Connectivity matrix:

.	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

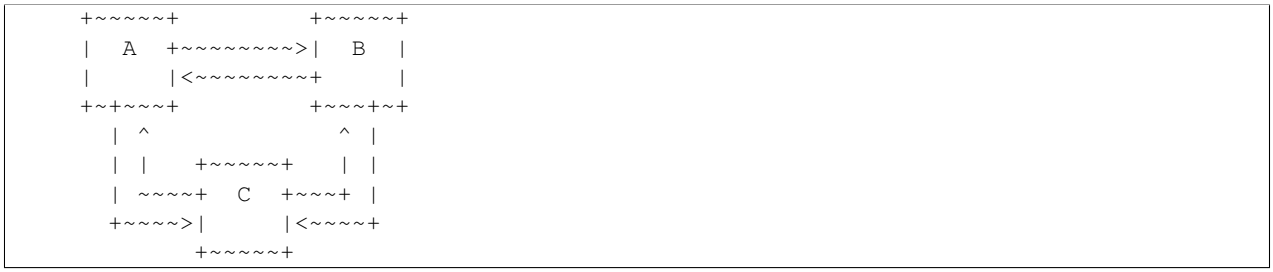
Connectivity matrix:

.	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:

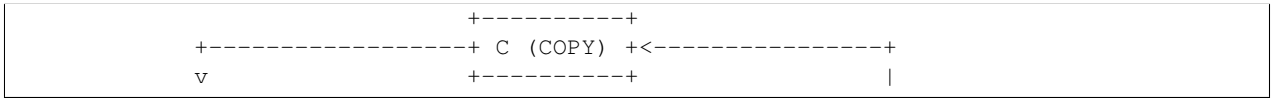
.	A	B	C
A	0	1	0
B	1	0	0
C	1	1	0

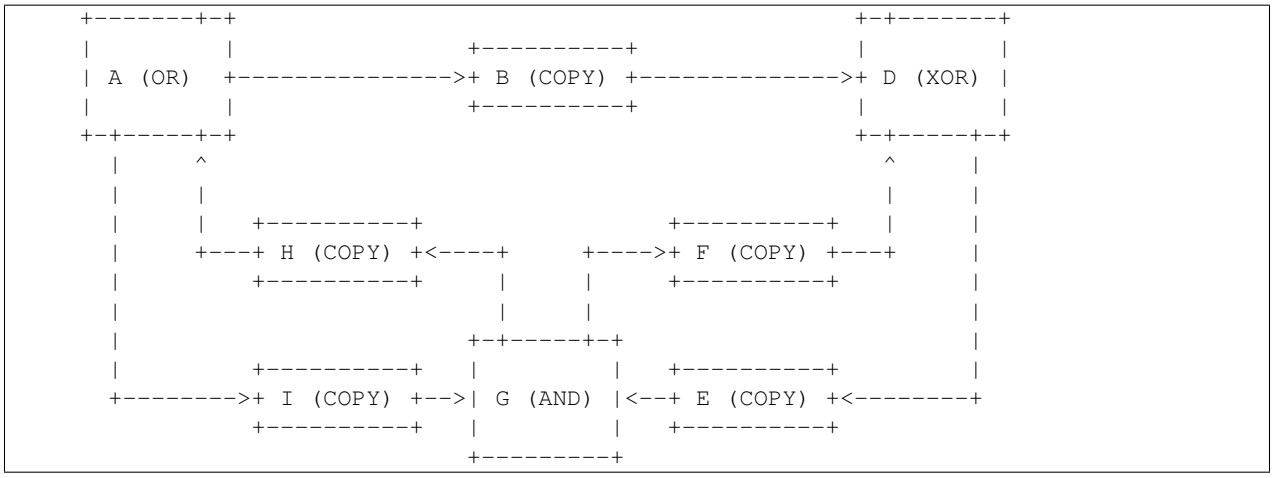
pyphi.examples.**propagation_delay_network**()

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.**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.

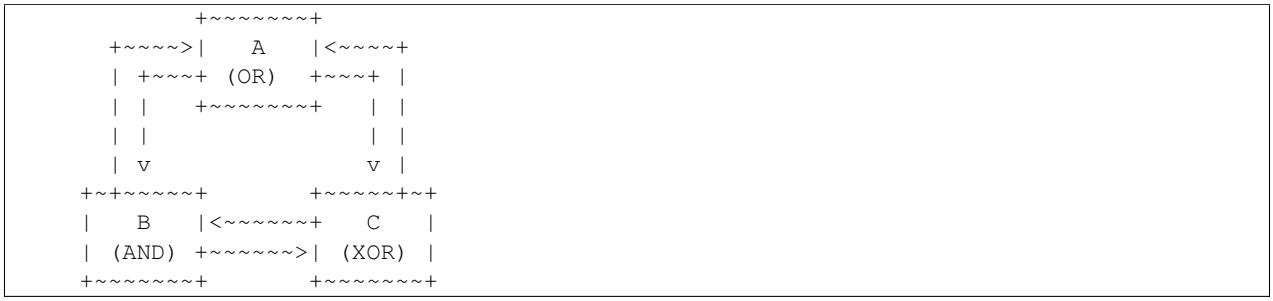
pyphi.examples.**fig1a**()
The network shown in Figure 1A of the 2014 IIT 3.0 paper.

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.

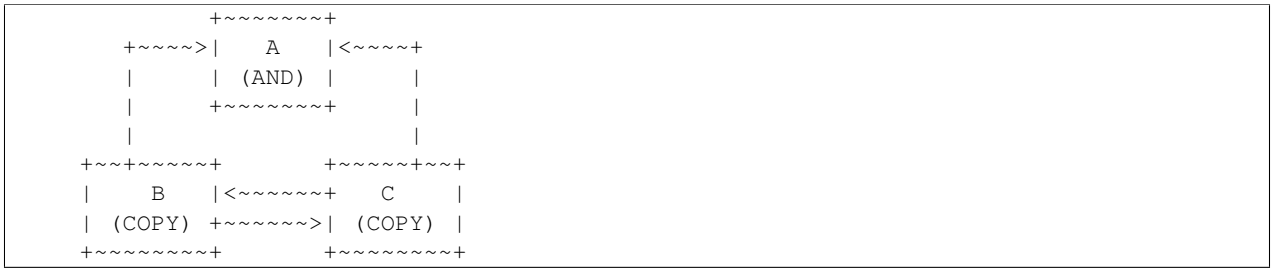
pyphi.examples.**fig4**()
The network shown in Figure 4 of the 2014 IIT 3.0 paper.

Diagram:



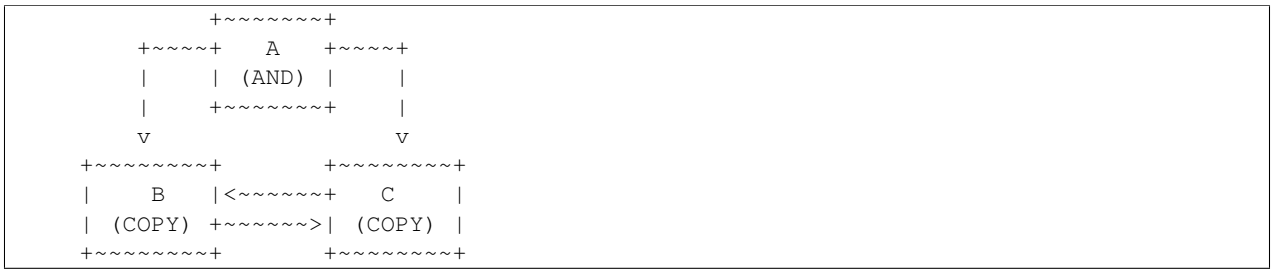
pyphi.examples.**fig5a**()
The network shown in Figure 5A of the 2014 IIT 3.0 paper.

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.

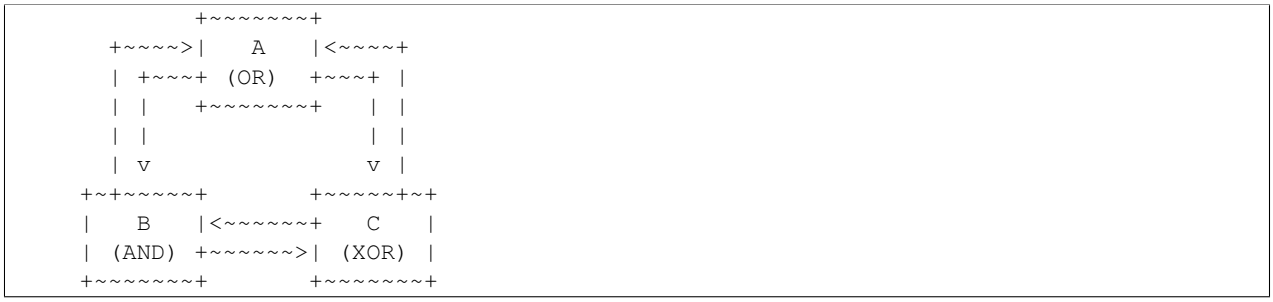
pyphi.examples.**fig10**()
The network shown in Figure 4 of the 2014 IIT 3.0 paper.

Diagram:

pyphi.examples.**fig6**()

The network shown in Figure 4 of the 2014 IIT 3.0 paper.

Diagram:



pyphi.examples.**fig8**()

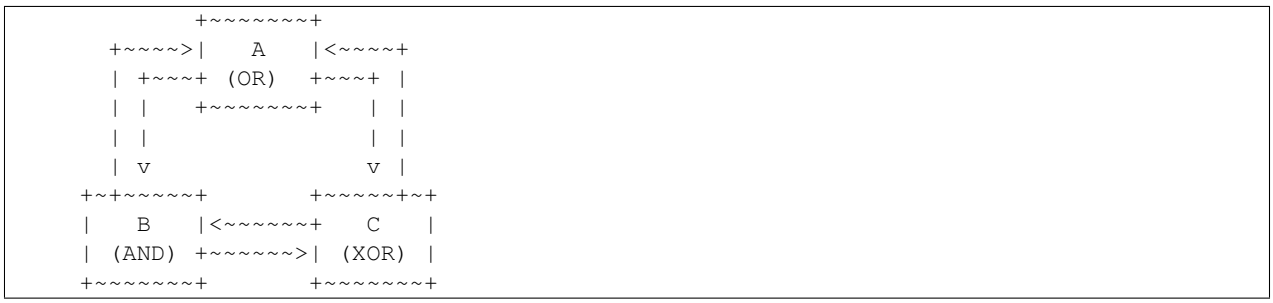
The network shown in Figure 4 of the 2014 IIT 3.0 paper.

Diagram:

pyphi.examples.**fig9**()

The network shown in Figure 4 of the 2014 IIT 3.0 paper.

Diagram:



4.1.7 macro

Methods for coarse-graining systems to different levels of spatial analysis.

class pyphi.macro.**MacroNetwork**(*macro_network*, *macro_phi*, *micro_network*, *micro_phi*, *partition*, *grouping*)

A coarse-grained network of nodes.

See the *Emergence (Macro/Micro)* 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.

partition
list – The partition which defines macro-elements in terms of micro-elements.

grouping
list(list – The correspondence between micro-states and macro-states.

emergence
float – The difference between the Φ of the macro- and the micro-system.

pyphi.macro.**list_all_partitions**(*network*)

Return a list of all possible coarse grains of a network.

Parameters **network** (Network) – The physical system to act as the 'micro' level.

Returns

partitions –

A list of possible partitions. Each element of the list is a list of micro-elements which correspong to macro-elements.

Return type ''list(list

pyphi.macro.**list_all_groupings**(*partition*)

Return all possible groupings of states for a particular coarse graining (partition) of a network.

Parameters

- **network** (Network) – The physical system on the micro level.
- **partitions** (*list(list*) – The partition of micro-elements into macro elements.

Returns

groupings –

A list of all possible correspondences between micro-states and macro-states for the partition.

Return type ''list(list(list(list

pyphi.macro.**make_mapping**(*partition*, *grouping*)

Return a mapping from micro-state to the macro-states based on the partition of elements and grouping of states.

Parameters

- **partition** (*list(list*) – A partition of micro-elements into macro elements.
- **grouping** (*list(list(list*) – For each macro-element, a list of micro states which set it to ON or OFF.

Returns **mapping** – A mapping from micro-states to macro-states.

Return type nd.array

pyphi.macro.**make_macro_tpm**(*micro_tpm*, *mapping*)
Create the macro TPM for a given mapping from micro to macro-states.

Parameters

- **micro_tpm** (*nd.array*) – The TPM of the micro-system.
- **mapping** (*nd.array*) – A mapping from micro-states to macro-states.

Returns **macro_tpm** – The TPM of the macro-system.

Return type nd.array

pyphi.macro.**make_macro_network**(*network*, *state*, *mapping*)
Create the macro-network for a given mapping from micro to macro-states.

Returns None if the macro TPM does not satisfy the conditional independence assumption.

Parameters

- **micro_tpm** (*nd.array*) – TPM of the micro-system.
- **mapping** (*nd.array*) – Mapping from micro-states to macro-states.

Returns macro_network (Network): Network of the macro-system, or None.

pyphi.macro.**emergence**(*network*, *state*)
Check for emergence of a macro-system into a macro-system.

Checks all possible partitions and groupings of the micro-system to find the spatial scale with maximum integrated information.

Parameters **network** (Network) – The network of the micro-system under investigation.

Returns

macro_network –

The maximal coarse-graining of the micro-system.

Return type MacroNetwork

pyphi.macro.**effective_info**(*network*)
Return the effective information of the given network.

This is equivalent to the average of the *effect_info()* (with the entire network as the mechanism and purview) over all posisble states of the network. It can be interpreted as the "noise in the network's TPM," weighted by the size of its state space.

Warning: If config.VALIDATE_SUBSYSTEM_STATES is enabled, then unreachable states are omitted from the average.

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.

4.1.8 memory

Decorators and objects for memoization.

pyphi.memory.**cache**(*ignore=[]*)
Decorator for memoizing a function using either the filesystem or a database.

class pyphi.memory.**DbMemoizedFunc**(*func*, *ignore*)
A memoized function, with a databse backing the cache.

get_output_key(*args*, *kwargs*)
Return the key that the output should be cached with, given arguments, keyword arguments, and a list of arguments to ignore.

load_output(*args*, *kwargs*)
Return cached output.

4.1.9 models

4.1.10 network

Represents the network of interest. This is the primary object of PyPhi and the context of all φ and Φ computation.

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.

Return type Network

class pyphi.network.**Network**(*tpm*, *connectivity_matrix=None*, *perturb_vector=None*, *purview_cache=None*)
A network of nodes.

Represents the network we're analyzing and holds auxilary data about it.

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 $\{N_0 = 0, N_1 = 0, N_2 = 1\}$.

Parameters **tpm** (*np.ndarray*) – See the corresponding attribute.

Keyword Arguments **connectivity_matrix** (*array or sequence*) – 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)**.

tpm
np.ndarray – The network's transition probability matrix. It 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), and in state-by-state form, so must column indices. If given in state-by-node form, it 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.

connectivity_matrix
np.ndarray – A square binary adjacency matrix indicating the connections between nodes in the network.

size
int – The number of nodes in the network.

num_states
int – The number of possible states of the network.

size

num_states

node_indices

tpm

connectivity_matrix

perturb_vector

__eq__(*other*)
Return whether this network equals the other object.

Two networks are equal if they have the same TPM, connectivity matrix, and perturbation vector.

to_json()

4.1.11 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*, *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.

get_marbl(*normalize=True*)
Generate a Marbl for this node TPM.

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.

marbl
The normalized representation of this node's Markov blanket, conditioned on the fixed state of boundary-condition nodes in the current timestep.

raw_marbl
The un-normalized representation of this node's Markov blanket, conditioned on the fixed state of boundary-condition nodes in the current timestep.

__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()

4.1.12 subsystem

Represents a candidate system for φ and Φ evaluation.

class pyphi.subsystem.**Subsystem**(*network*, *state*, *node_indices*, *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.
- **node_indices** (*tuple(int*) – A sequence of indices of the nodes in this subsystem.

Keyword Arguments **cut** (*Cut*) – The unidirectional Cut to apply to this subsystem.

nodes
list(Node – A list of nodes in the subsystem.

node_indices
tuple(int – The indices of the nodes in the subsystem.

size
int – The number of nodes in the subsystem.

network
Network – The network the subsystem belongs to.

state
tuple(int – The state of the subsystem's network. state[i] gives the state of node i.

proper_state
tuple(int – The state of the subsystem. proper_state[i] gives the i^{th} node in the subsystem. Note that this is **not** the state of node i.

cut
Cut – The cut that has been applied to this subsystem.

connectivity_matrix
np.array – The connectivity matrix after applying the cut.

cut_matrix
np.array – A matrix of connections which have been severed by the cut.

perturb_vector
np.array – The vector of perturbation probabilities for each node.

null_cut
Cut – The cut object representing no cut.

tpm
np.array – The TPM conditioned on the state of the external nodes.

Construct a Subsystem.

state
The state of the Network this Subsystem belongs to.

proper_state
The state of the nodes in this Subsystem.

size
The size of this Subsystem.

is_cut()
Return whether this Subsystem has a cut applied to it.

repertoire_cache_info()
Report repertoire cache statistics.

__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.

indices2nodes(*indices*)
Return nodes for these indices.

Parameters **indices** (*iterable(int)*) –

Returns

nodes –

The *Node* objects corresponding to these indices.

Return type tuple(Node

Raises ValueError – If requested indices are not in the subsystem.

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

cause_repertoire –

The cause repertoire of the mechanism over the purview.

Return type np.ndarray

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

effect_repertoire –

The effect repertoire of the mechanism over the purview.

Return type np.ndarray

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.

expand_repertoire(*direction*, *purview*, *repertoire*, *new_purview=None*)
Expand a partial repertoire over a purview to a distribution over a new state space.

Parameters

- **direction** (*str*) – Either `DIRECTIONS[PAST]` or `DIRECTIONS[FUTURE]`.
- **purview** (*tuple(int)*) – The purview over which the repertoire was calculated.
- **repertoire** (`np.ndarray`) – A repertoire computed over `purview`.

Keyword Arguments **new_purview** (*tuple(int)*) – The purview to expand the repertoire over. Defaults to the entire subsystem.

Returns The expanded repertoire.

Return type `np.ndarray`

expand_cause_repertoire(*purview*, *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(*purview*, *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.

find_mip(*direction*, *mechanism*, *purview*)
Return the minimum information partition for a mechanism over a purview.

Parameters

- **direction** (*str*) – Either `DIRECTIONS[PAST]` or `DIRECTIONS[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.

Return type `Mip`

mip_past(*mechanism*, *purview*)
Return the past minimum information partition.

Alias for *find_mip()* with `direction` set to `DIRECTIONS[PAST]`.

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** (*str*) – The temporal direction (`DIRECTIONS[PAST]` or `DIRECTIONS[FUTURE]`) 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 **mice** – The maximally-irreducible cause or effect.

Return type `Mice`

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, `DIRECTIONS[PAST]` or `DIRECTIONS[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 `DIRECTIONS[PAST]`.

core_effect(*mechanism*, *purviews=False*)
Return the core effect repertoire of a mechanism.

Alias for *find_mice()* with `direction` set to `DIRECTIONS[PAST]`.

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 all φ bipartitions of a mechanism over a purview.

Excludes all bipartitions where one half is entirely empty, e.g:

A [] A []

— X – is not valid, but — X — is. B [] [] B

Parameters

- **mechanism** (*tuple(int)*) – The mechanism to partition
- **purview** (*tuple(int)*) – The purview to partition

Returns

bipartitions –

Where each partition is

bipart[0].mechanism bipart[1].mechanism ———————- X ———————-

bipart[0].purview bipart[1].purview

Return type list(tuple((Part, Part

Example

```
>>> from pyphi.subsystem import mip_bipartitions
>>> mechanism = (0,)
>>> purview = (2, 3)
>>> mip_bipartitions(mechanism, purview)
[(Part(mechanism=(), purview=(2,)), Part(mechanism=(0,), purview=(3,))), (Part(mechanism=(), pur
```

4.1.13 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.`**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.`**apply_cut** (*cut*, *connectivity_matrix*)
Return a modified connectivity matrix where the connections from one set of nodes to the other are destroyed.

`pyphi.utils.`**fully_connected** (*connectivity_matrix*, *nodes1*, *nodes2*)
Test connectivity of one set of nodes to another.

Parameters

- **connectivity_matrix** (np.ndarray) – 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

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.

Return type bool

pyphi.utils.**apply_boundary_conditions_to_cm**(*external_indices*, *connectivity_matrix*)
Return a connectivity matrix with all connections to or from external nodes removed.

pyphi.utils.**get_inputs_from_cm**(*index*, *connectivity_matrix*)
Return a tuple of node indices that have connections to the node with the given index.

pyphi.utils.**get_outputs_from_cm**(*index*, *connectivity_matrix*)
Return a tuple of node indices that the node with the given index has connections to.

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.**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 **combinations** – An array of combinations.

Return type np.ndarray

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

combination_indices –

Indices that give the k-combinations of n elements.

Return type np.ndarray

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 **chain** – An chained iterator over the power set.

Return type Iterable

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

distribution –

The uniform distribution over the set of nodes.

Return type np.ndarray

pyphi.utils.**marginalize_out**(*index*, *tpm*, *perturb_value=0.5*)

Marginalize out a node from a TPM.

Parameters

- **index** (*list*) – The index of the node to be marginalized out.
- **tpm** (*np.ndarray*) – The TPM to marginalize the node out of.

Returns

tpm –

A TPM with the same number of dimensions, with the node marginalized out.

Return type np.ndarray

pyphi.utils.**max_entropy_distribution**(*node_indices*, *number_of_nodes*, *perturb_vector=None*)

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

distribution –

The maximum entropy distribution over the set of nodes.

Return type `np.ndarray`

`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.`**bipartition**(*a*)

Return a list of bipartitions for a sequence.

Parameters **a** (*Iterable*) – The iterable to partition.

Returns

bipartition –

A list of tuples containing each of the two partitions.

Return type ‘‘list(tuple(tuple

Example

```
>>> from pyphi.utils import bipartition
>>> 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

bipartition –

A list of tuples containing each of the two partitions.

Return type ‘‘list(tuple(tuple

Example

```
>>> from pyphi.utils import directed_bipartition
>>> directed_bipartition((1, 2, 3))
[((), (1, 2, 3)), ((1,), (2, 3)), ((2,), (1, 3)), ((1, 2), (3,)), ((3,), (1, 2)), ((1, 3), (2,))
```

`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

bipartition –

A list of tuples containing each of the two partitions.

Return type ‘‘list(tuple(tuple

Example

```
>>> from pyphi.utils import directed_bipartition_of_one
>>> 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.

The directed bipartion

Parameters **N** (*int*) – The length of the sequence.

Returns

bipartition_indices –

A list of tuples containing the indices for each of the two partitions.

Return type list

Example

```
>>> from pyphi.utils import directed_bipartition_indices
>>> N = 3
>>> directed_bipartition_indices(N)
[((), (0, 1, 2)), ((0,), (1, 2)), ((1,), (0, 2)), ((0, 1), (2,)), ((2,), (0, 1)), ((0, 2), (1,))]
```

pyphi.utils.**bipartition_indices**(*N*)

Return indices for bipartitions of a sequence.

Parameters **N** (*int*) – The length of the sequence.

Returns

bipartition_indices –

A list of tuples containing the indices for each of the two partitions.

Return type list

Example

```
>>> from pyphi.utils import bipartition_indices
>>> N = 3
>>> bipartition_indices(N)
[((), (0, 1, 2)), ((0,), (1, 2)), ((1,), (0, 2)), ((0, 1), (2,))]
```

pyphi.utils.**submatrix**(*cm*, *nodes1*, *nodes2*)

Return the submatrix of connections from nodes1 to nodes2.

Parameters

- **cm** (*np.ndarray*) – The matrix
- **nodes1** (*tuple(int)*) – Source nodes
- **nodes2** (*tuple(int)*) – Sink nodes

pyphi.utils.**relevant_connections**(*n*, *_from*, *to*)

Construct a connectivity matrix.

Returns 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`.

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`

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.**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.

4.1.14 validate

Methods for validating common types of input.

exception pyphi.validate.**StateUnreachableError**(*state*, *message*)
Raised when the current state cannot be reached from any past state.

pyphi.validate.**direction**(*direction*)
Validate that the given direction is one of the allowed constants.

pyphi.validate.**tpm**(*tpm*)
Validate a TPM.

pyphi.validate.**conditionally_independent**(*tpm*)
Validate that the TPM is conditionally independent.

pyphi.validate.**connectivity_matrix**(*cm*)
Validate the given connectivity matrix.

pyphi.validate.**perturb_vector**(*pv*, *size*)
Validate a network's pertubation vector.

pyphi.validate.**network**(*n*)
Validate a *Network*.

Checks the TPM, connectivity matrix, and perturbation vector.

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.

If constrained_nodes is provided, then nodes not in constrained_nodes will be left free (their state will not considered restricted by the TPM). Otherwise, any nodes without inputs will be left free.

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.

p

Symbols

A

B

C

D

E

P

R

S

T

U

X