
PyPhi Documentation

Release v0.8.1

William GP Mayner

Sep 20, 2017

Usage and Examples

1	Getting started	3
2	Basic Usage	5
3	IIT 3.0 Paper (2014)	7
4	Actual Causation	19
5	Conditional Independence	25
6	Emergence (coarse-graining and blackboxing)	29
7	Magic Cuts	35
8	Residue	39
9	XOR Network	43
10	Loading a configuration	47
11	Approximations and theoretical options	49
12	System resources	51
13	Caching	53
14	Logging	55
15	Numerical precision	57
16	Miscellaneous	59
17	The <code>config</code> API	61
18	Connectivity Matrices	63
19	LOLI: Low-Order bits correspond to Low-Index nodes	65
20	<code>actual</code>	67

21	<code>compute.big_phi</code>	73
22	<code>compute.concept</code>	77
23	<code>compute.distance</code>	79
24	<code>config</code>	81
25	<code>connectivity</code>	89
26	<code>constants</code>	91
27	<code>convert</code>	93
28	<code>distance</code>	103
29	<code>examples</code>	107
30	<code>exceptions</code>	115
31	<code>jsonify</code>	117
32	<code>macro</code>	119
33	<code>models.actual_causation</code>	125
34	<code>models.big_phi</code>	129
35	<code>models.concept</code>	131
36	<code>models.cuts</code>	135
37	<code>network</code>	139
38	<code>node</code>	143
39	<code>partition</code>	145
40	<code>subsystem</code>	149
41	<code>timescale</code>	157
42	<code>tpm</code>	159
43	<code>utils</code>	161
44	<code>validate</code>	163
	Python Module Index	165

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

Install IPython by running `pip install ipython` on the command line. Then run it with the command `ipython`.

Lines of code beginning with `>>>` and `...` can be pasted directly into IPython.

CHAPTER 2

Basic Usage

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 *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 $(i, j)^{\text{th}}$ 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]
... ])
```

We'll also make labels for the network nodes so that PyPhi's output is easier to read.

```
>>> labels = ('A', 'B', 'C')
```

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

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

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:

```
>>> node_indices = (0, 1, 2)
>>> subsystem = pyphi.Subsystem(network, state, node_indices)
```

Tip: Node labels can be used instead of indices when constructing a *Subsystem*:

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

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

```
>>> pyphi.compute.big_phi(subsystem)
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 *big_mip()*. This returns a 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
```

See the documentation for *BigMip* and *Concept* for more information on these objects.

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

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, in order to become more familiar with both the theory and the software.

Install IPython by running `pip install ipython` on the command line. Then run it with the command `ipython`.

Lines of code beginning with `>>>` and `...` can be pasted directly into IPython.

We begin by importing 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, 1, 0],
...     [1, 0, 0, 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, 0, 0, 0, 1, 0],
...     [1, 0, 0, 0, 0, 0],
...     [1, 1, 0, 0, 1, 0],
...     [1, 0, 0, 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],
...     [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 `examples` module with `network = pyphi.examples.fig0a()`. The TPM can then be accessed with `network.tpm`.

Next we'll define the connectivity matrix. In PyPhi, the $(i, j)^{\text{th}}$ 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 subsystem 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 $\varphi_{\text{cause}}^{\text{MIP}}$ and $\varphi_{\text{effect}}^{\text{MIP}}$ values in their respective `phi` attributes, and the minimal partitions in their `partition` attributes:

```

>>> mip_c.phi
0.499999
>>> mip_c.partition
 0      1,2
--  ---
   0,1,2
>>> mip_e.phi
0.25
>>> mip_e.partition
   0,1,2
--  ---
 1      0,2

```

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

Note that the minimal partition found for the past is

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

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, you 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 like so:

```
network = pyphi.network.from_json('path/to/network.json')
```

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(subsystem.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 `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:

```
>>> constellation.labeled_mechanisms
[['A'], ['B'], ['C'], ['A', 'B'], ['B', 'C'], ['A', 'B', 'C']]
>>> constellation.phis
[0.166667, 0.166667, 0.25, 0.25, 0.333334, 0.499999]
```

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

```
>>> subsystem.null_concept.phi
0.0
```

Figure 11

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

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

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

Figure 12

Assessing the integrated conceptual information Φ of a constellation C .

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

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

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

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

```
>>> big_mip.phi
1.9166650000000001
>>> big_mip.cut
Cut [0, 1] // [2]
```

Note: This Cut represents removing any connections from the nodes with indices 0 and 1 to the node with index 2.

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
(A, B, C)
```

Figure 15

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

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)
((A, B, C), 1.9166650000000001)
>>> (FG.subsystem.nodes, FG.phi)
((F, G), 0.069445)
```

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

Actual Causation

This section demonstrates how to evaluate actual causation with PyPhi. # TODO: add paper reference

```
>>> import pyphi
>>> from pyphi import actual, config
>>> from pyphi.constants import Direction
```

Configuration

Before we begin we need to set some configuration values. The correct way of partitioning for actual causation is using the 'ALL' partitions setting; 'TRI'-partitions are a reasonable approximation. In case of ties the smaller purview should be chosen. IIT 3.0 style bipartitions will give incorrect results.

```
>>> config.PARTITION_TYPE = 'TRI'
>>> config.PICK_SMALLEST_PURVIEW = True
```

When calculating a causal account of the transition between a set of elements X at time $t - 1$ and a set of elements Y at time t , with X and Y being subsets of the same system, the transition should be valid according to the system's TPM. However, the state of X at $t - 1$ does not necessarily need to have a valid past state so we can disable state validation:

```
>>> config.VALIDATE_SUBSYSTEM_STATES = False
```

Computation

We will look at how to perform computations over the basic *OR-AND* network introduced in Figure 1 of the paper.

```
>>> network = pyphi.examples.actual_causation()
```

This is a standard PyPhi *Network* so we can look at its TPM:

```
>>> pyphi.convert.state_by_node2state_by_state(network.tpm)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

The OR gate is element 0, and the AND gate is element 1 in the network.

```
>>> OR = 0
>>> AND = 1
```

We want to observe both elements at $t - 1$ and t , with OR on and AND off in both observations:

```
>>> X = Y = (OR, AND)
>>> X_state = Y_state = (1, 0)
```

The *Transition* object is the core of all actual causation calculations. To instantiate a *Transition*, we pass it a *Network*, the state of the network at $t - 1$ and t , and elements of interest at $t - 1$ and t . Note that PyPhi requires the state to be the state of the entire network, not just the state of the nodes in the transition.

```
>>> transition = actual.Transition(network, X_state, Y_state, X, Y)
```

Cause and effect repertoires can be obtained for the transition. For example, as shown on the right side of Figure 2B, we can compute the effect repertoire to see how $X_{t-1} = \{OR = 1\}$ constrains the probability distribution of the purview $Y_t = \{OR, AND\}$:

```
>>> transition.effect_repertoire((OR,), (OR, AND))
array([[ 0. ,  0. ],
       [ 0.5,  0.5]])
```

Similarly, as in Figure 2C, we can compute the cause repertoire of $Y_t = \{OR, AND = 10\}$ to see how it constrains the purview $X_{t-1} = \{OR\}$:

```
>>> transition.cause_repertoire((OR, AND), (OR,))
array([[ 0.5],
       [ 0.5]])
```

Note: In all *Transition* methods the constraining occurrence is passed as the *mechanism* argument and the constrained occurrence is the *purview* argument. This mirrors the terminology introduced in the IIT code.

Transition also provides methods for computing cause and effect ratios. For example, the effect ratio of $X_{t-1} = \{OR = 1\}$ constraining $Y_t = \{OR\}$ (as shown in Figure 3A) is computed as follows:

```
>>> transition.effect_ratio((OR,), (OR,))
0.415037
```

The effect ratio of $X_{t-1} = \{OR = 1\}$ constraining $Y_t = \{AND\}$ is negative:

```
>>> transition.effect_ratio((OR,), (AND,))
-0.584963
```

And the cause ratio of $Y_t = \{OR = 1\}$ constraining $X_{t-1} = \{OR, AND\}$ (Figure 3B) is:

```
>>> transition.cause_ratio((OR,), (OR, AND))
0.415037
```


We can evaluate α for a particular pair of occurrences, as in Figure 3C. For example, to find the irreducible effect ratio of $\{OR, AND\} \rightarrow \{OR, AND\}$, we use the `find_mip` method:

```
>>> link = transition.find_mip(Direction.FUTURE, (OR, AND), (OR, AND))
```

This returns a `AcMip` object, with a number of useful properties. This particular MIP is reducible, as we can see by checking the value of α :

```
>>> link.alpha
0.0
```

The `partition` property shows the minimum information partition that reduces the occurrence and candidate effect:

```
>>> link.partition
      0      1
--  --  --
      0      1
```

Let's look at the MIP for the irreducible occurrence $Y_t = \{OR, AND\}$ constraining $X_{t-1} = \{OR, AND\}$ (Figure 3D). This candidate causal link has positive α :

```
>>> link = transition.find_mip(Direction.PAST, (OR, AND), (OR, AND))
>>> link.alpha
0.169925
```

To find the actual cause or actual effect of a particular occurrence, use the `find_actual_cause` or `find_actual_effect` methods:

```
>>> transition.find_actual_cause((OR, AND))
CausalLink
   $\alpha = 0.1699$  (0, 1) (0, 1)
```

Accounts

The complete causal account of our transition can be computed with the `account` function:

```
>>> account = actual.account(transition)
>>> print(account)

Account (5 causal links)
*****
Irreducible effects
 $\alpha = 0.415$  (0,) (0,)
 $\alpha = 0.415$  (1,) (1,)
Irreducible causes
 $\alpha = 0.415$  (0,) (0,)
 $\alpha = 0.415$  (1,) (1,)
 $\alpha = 0.1699$  (0, 1) (0, 1)
```

We see that this function produces the causal links shown in Figure 4. The `Account` object is a subclass of `tuple`, and can be manipulated the same:

```
>>> len(account)
5
```

Irreducible Accounts

The irreducibility of the causal account of our transition of interest can be evaluated using the following function:

```
>>> big_mip = actual.big_acmip(transition)
>>> big_mip.alpha
0.169925
```

As shown in Figure 4, the second order occurrence $Y_t = \{OR, AND = 10\}$ is destroyed by the MIP:

```
>>> big_mip.partitioned_account

Account (4 causal links)
*****
Irreducible effects
α = 0.415 (0,) (0,)
α = 0.415 (1,) (1,)
Irreducible causes
α = 0.415 (0,) (0,)
α = 0.415 (1,) (1,)
```

The partition of the MIP is available in the `cut` property:

```
>>> big_mip.cut
(0,) / / (1,) && (1,) / / (0,)
```

To find all irreducible accounts within the transition of interest, use `nexus`:

```
>>> all_accounts = actual.nexus(network, X_state, Y_state)
```

This computes \mathcal{A} for all permutations of elements in X_{t-1} and Y_t and returns a tuple of all `AcBigMip` objects with $\mathcal{A} > 0$:

```
>>> for n in all_accounts:
...     print(n.transition, n.alpha)
Transition([OR] [OR]) 2.0
Transition([AND] [AND]) 2.0
Transition([OR, AND] [OR, AND]) 0.169925
```

The `causal_nexus` function computes the maximally irreducible account for the transition of interest:

```
>>> cn = actual.causal_nexus(network, X_state, Y_state)
>>> cn.alpha
2.0
>>> cn.transition
Transition([OR] [OR])
```

Disjunction of conjunctions

If you are interested in exploring further, the disjunction of conjunctions network from Figure 7 is provided as well:

```
>>> network = pyphi.examples.disjunction_conjunction_network()
>>> cn = actual.causal_nexus(network, (1, 0, 1, 0), (0, 0, 0, 1))
```

The only irreducible transition is from $X_{t-1} = C$ to $Y_t = D$, with \mathcal{A} of 2.0:

```
>>> cn.transition
Transition([C] [D])
>>> cn.alpha
2.0
```

Conditional Independence

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 exhibits “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.tpm.marginalize_out(C, sbn_tpm2)
```

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

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

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

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

Emergence (coarse-graining and blackboxing)

Coarse-graining

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

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

The connectivity matrix is all-to-all:

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

We'll set the state so that nodes are off.

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

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

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

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

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

We start by considering the first coarse grain:

```
>>> coarse_grain = grains[0]
```

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

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

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

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

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

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

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

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

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

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

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

For the second macro-element, the grouping means that the element will be off if its micro-element is off, and on if its micro-element is on.

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

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

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

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

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

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

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

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

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

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

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

Let's consider a different coarse-graining instead.

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

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

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

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

```
>>> macro_subsystem = pyphi.macro.MacroSubsystem(
...     network, state, network.node_indices, coarse_grain=coarse_grain)
>>> macro_subsystem
MacroSubsystem((m0, m1))
```

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

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

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

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

```
0.483323
>>> M.system
(0, 1, 2, 3)
>>> M.coarse_grain.partition
((0, 1), (2, 3))
>>> M.coarse_grain.grouping
(((0, 1), (2,)), ((0, 1), (2,)))
```

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

Blackboxing

- `pyphi.examples.blackbox_network()`

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

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

We consider the state where all nodes are off:

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

The system has minimal Φ without blackboxing:

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

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

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

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

Blackboxes have a few convenient attributes and methods. The `hidden_indices` attribute returns the elements which are hidden within blackboxes:

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

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

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

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

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

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

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

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

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

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

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

We can now compute Φ for this macro system:

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

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

Magic Cuts

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)
>>> print(macro.emergence)
-1.112671
```

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(A, B, C)
>>> print(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 equivalent cuts for this system; for concreteness we sever all connections from elements A and B to C .

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

Let's investigate the concepts in the unpartitioned constellation,

```
>>> constellation.labeled_mechanisms
[['A'], ['B'], ['C'], ['A', 'B'], ['A', 'C'], ['B', 'C']]
>>> constellation.phis
[0.125, 0.125, 0.125, 0.499999, 0.499999, 0.499999]
>>> print(sum(_))
1.874997
```

and also the concepts of the partitioned constellation.

```
>>> cut_constellation.labeled_mechanisms
[['A'], ['B'], ['C'], ['A', 'B'], ['B', 'C'], ['A', 'B', 'C']]
>>> cut_constellation.phis
[0.125, 0.125, 0.125, 0.499999, 0.266666, 0.333333]
>>> print(sum(_))
1.474998
```

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

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

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

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

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

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

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


Next, let's 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_on.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, ()))
>>> print(pyphi.distance.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 that 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 whose 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. Let's first load the network,

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

For this example, it is the subsystem consisting of A , B , and E 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 AE to B . Investigating the concepts in both the partitioned and unpartitioned constellations,

```
>>> mip.unpartitioned_constellation.labeled_mechanisms
[['A'], ['B'], ['A', 'B']]
>>> mip.unpartitioned_constellation.phis
[0.25, 0.166667, 0.178572]
>>> print(sum(_))
0.5952390000000001
```

The unpartitioned constellation has mechanisms A , B and AB with $\sum \varphi = 0.595239$.

```
>>> mip.partitioned_constellation.labeled_mechanisms
[['A'], ['B'], ['A', 'B']]
>>> mip.partitioned_constellation.phis
[0.25, 0.166667, 0.214286]
>>> print(sum(_))
0.630953
```

The partitioned constellation has mechanisms A , B and AB but 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 AB , resulting in an overall greater $\sum \varphi$ for the partitioned 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

To calculate the MIP structure of mechanism AB :

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

We can then determine what the specific partition is.

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

The indices (0, 1, 2, 3, 4) correspond to nodes A, B, C, D, E respectively. Thus the MIP is $\frac{AB}{DE} \times \emptyset$, 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 $\frac{A}{CD} \times \frac{\emptyset}{E}$ because E has no effect on A , so it has zero φ .

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

Instead, we should evaluate A over the purview CD .

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

In this case, there is a well defined MIP

```
>>> mip_A.partition
  0
-- --
  2  3
```

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

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

and irreducible cause information

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

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

XOR Network

This example describes a system of three fully connected XOR nodes, *A*, *B* and *C* (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)
```

According to IIT, existence is a holistic notion; 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(A, B, C)
```

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
>>> constellation.labeled_mechanisms
[['A', 'B'], ['A', 'C'], ['B', 'C']]
```

There are three concepts in the constellation. They are all the possible second order mechanisms: AB , AC and BC .

Focusing on the concept specified by mechanism AB , 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 ABC , 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 A and B 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 C . The mechanism AB is able to completely specify the next state of C . Since both nodes are off, the next state of C will be off.

The mechanism AB does not provide any information about the next state of either A or B , because the relationship depends on the value of C . That is, the next state of A (or B) may be either on or off, depending on the value of C . Any purview larger than C would be reducible by pruning away the additional elements.

Main Complex: ABC with $\Phi = 1.875$			
Mechanism	φ	Cause Purview	Effect Purview
AB	0.5	ABC	C
AC	0.5	ABC	B
BC	0.5	ABC	A

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 AB , AC , and BC .

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 A , B or C 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 A .

```
>>> subsystem = pyphi.examples.xor_subsystem()
>>> A = (0,)
>>> ABC = (0, 1, 2)
```

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

```
>>> subsystem.cause_info(A, ABC)
0.5
>>> subsystem.effect_info(A, ABC)
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 A . There is no self-loop, so A can have no effect on itself. Without knowing the current state of A , in the next state B could be either on or off. If we know that the current state of A is on, then B could still be either on or off, depending on the state of C . Thus, on its own, the current state of A does not provide any information about the next state of B . A similar result holds for the effect of A on C . Since A 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 ABC . Consider the cause information over the purview ABC :

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

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

```
>>> mip = subsystem.mip_past(ABC, ABC)
>>> mip.phi
0.0
>>> mip.partition
0      1,2
-- ---
      0,1,2
```

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

$$\frac{A}{\emptyset} \times \frac{BC}{ABC}$$

This result can be understood as follows: knowing that B and C are off in the current state is sufficient to know that A , B , and C were all off in the past state; there is no additional information gained by knowing that A is currently off.

Similarly for any other potential purview, the current state of B and C 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.

Loading a configuration

Various aspects of PyPhi's behavior can be configured.

When PyPhi is imported, it checks for a YAML file named `pyphi_config.yml` in the current directory and automatically loads it if it exists; otherwise the default configuration is used.

The various settings are listed here with their defaults.

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

It is also possible to manually load a configuration file:

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

Or load a dictionary of configuration values:

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

Many settings can also be changed on the fly by simply assigning them a new value:

```
>>> pyphi.config.PROGRESS_BARS = True
```

Approximations and theoretical options

These settings control the algorithms PyPhi uses.

- `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
```

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

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

- `MEASURE`: The measure to use when computing distances between repertoires and concepts. Users can dynamically register new measures with the `pyphi.distance.measures.register` decorator; see [distance](#) for examples. A full list of currently installed measures is available by calling `print(pyphi.distance.measures.all())`. Note that some measures cannot be used for calculating Φ because they are asymmetric.

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

- `PARTITION_TYPE`: Controls the type of partition used for φ computations.

If set to `'BI'`, partitions will have two parts.

If set to `'TRI'`, partitions will have three parts. In addition, computations will only consider partitions that strictly partition the mechanism the mechanism. That is, for the mechanism (A, B) and purview (B, C, D) the partition:

```
A, B
--  --
B    C, D
```

is not considered, but:

```
A      B
--  --
B      C, D
```

is. The following is also valid:

```
A, B
--  ----
      B, C, D
```

In addition, this setting introduces “wedge” tripartitions of the form:

```
A      B
--  --  --
B      C      D
```

where the mechanism in the third part is always empty.

In addition, in the case of a φ -tie when computing MICE, The 'TRIPARTITION' setting choses the MIP with smallest purview instead of the largest (which is the default).

Finally, if set to 'ALL', all possible partitions will be tested.

```
>>> defaults['PARTITION_TYPE']
'BI'
```

- PICK_SMALLEST_PURVIEW: When computing MICE, it is possible for several MIPs to have the same φ value. If this setting is set to True the MIP with the smallest purview is chosen; otherwise, the one with largest purview is chosen.

```
>>> defaults['PICK_SMALLEST_PURVIEW']
False
```

- USE_SMALL_PHI_DIFFERENCE_FOR_CONSTELLATION_DISTANCE: If set to True, the distance between constellations (when computing a *BigMip*) is calculated using the difference between the sum of φ in the constellations instead of the extended EMD.
- SYSTEM_CUTS: If set to '3.0_STYLE', then traditional IIT 3.0 cuts will be used when computing Φ . If set to 'CONCEPT_STYLE', then experimental concept- style system cuts will be used instead.

```
>>> defaults['SYSTEM_CUTS']
'3.0_STYLE'
```

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.

- `PARALLEL_CONCEPT_EVALUATION`: Controls whether concepts are evaluated in parallel when computing constellations.

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

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

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

- `PARALLEL_COMPLEX_EVALUATION`: Controls whether systems are evaluated in parallel when computing complexes.

```
>>> defaults['PARALLEL_COMPLEX_EVALUATION']  
False
```

Warning: Only one of `PARALLEL_CONCEPT_EVALUATION`, `PARALLEL_CUT_EVALUATION`, and `PARALLEL_COMPLEX_EVALUATION` can be set to `True` at a time. For maximal efficiency, you should parallelize the highest level computations possible, *e.g.*, parallelize complex evaluation instead of cut evaluation, but only if you are actually computing complexes. You should only parallelize concept evaluation if you are just computing constellations.

- `NUMBER_OF_CORES`: Controls 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
```

- `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 setting limits the percentage of a system's RAM that the caches can collectively use.

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


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

- `CACHE_BIGMIPS`: Controls whether *BigMip* objects are cached and automatically retrieved.

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

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

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

- `CACHING_BACKEND`: Controls whether precomputed results are stored and read from a local filesystem-based cache in the current directory or from a database. Set this to `'fs'` for the filesystem, `'db'` for the database.

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

- `FS_CACHE_VERBOSITY`: Controls how much caching information is printed if the filesystem cache is used. Takes a value between 0 and 11.

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

Warning: Printing during a loop iteration can slow down the loop considerably.

- `FS_CACHE_DIRECTORY`: If the filesystem is used for caching, 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 Python is being run.

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

- MONGODB_CONFIG: Set the configuration for the MongoDB database backend (only has an effect if CACHING_BACKEND is 'db').

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

- REDIS_CACHE: Specifies whether to use Redis to cache *Mice*.

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

- 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
```

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

Important: After PyPhi has been imported, changing these settings will have no effect unless you call `configure_logging()` afterwards.

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

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

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

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

- `LOG_FILE`: Controls the name of the log file.

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

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

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

Tip: If this is enabled and `LOG_FILE_LEVEL` is `INFO` or higher, then the log file can serve as an automatic record of which configuration settings you used to obtain results.

- `PROGRESS_BARS`: Controls whether to show progress bars on the console.

```
>>> defaults['PROGRESS_BARS']
True
```

Tip: If you are iterating over many systems rather than doing one long-running calculation, consider disabling this for speed.

Numerical precision

- **PRECISION:** If **MEASURE** is **EMD**, then the Earth Mover's Distance is calculated with an external C++ library that a numerical optimizer to find a good approximation. Consequently, systems with analytically zero Φ will sometimes be numerically found 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 $10e^{-\text{PRECISION}}$ will be considered insignificant and treated as zero. The default value is about as accurate as the EMD computations get.

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


- `VALIDATE_SUBSYSTEM_STATES`: Controls whether PyPhi checks if the subsystems's state is possible (reachable with nonzero probability 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
```

- `VALIDATE_CONDITIONAL_INDEPENDENCE`: Controls whether PyPhi checks if a system's TPM is conditionally independent.

```
>>> defaults['VALIDATE_CONDITIONAL_INDEPENDENCE']  
True
```

- `SINGLE_MICRO_NODES_WITH_SELFLOOPS_HAVE_PHI`: If set to `True`, the Phi value of single micro-node subsystems is the difference between their unpartitioned constellation (a single concept) and the null concept. If set to `False`, their Phi is defined to be zero. Single macro-node subsystems may always be cut, regardless of circumstances.

```
>>> defaults['SINGLE_MICRO_NODES_WITH_SELFLOOPS_HAVE_PHI']  
False
```

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

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

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

- `PRINT_FRACTIONS`: Controls whether numbers in a `repr` are printed as fractions. Numbers are still printed as decimals if the fraction's denominator would be large. This only has an effect if `REPR_VERBOSITY > 0`.

```
>>> defaults['PRINT_FRACTIONS']  
True
```


CHAPTER 17

The config API

`pyphi.config.load_config_dict (config)`

Load configuration values.

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

`pyphi.config.load_config_file (filename)`

Load config from a YAML file.

`pyphi.config.load_config_default ()`

Load default config values.

`pyphi.config.get_config_string ()`

Return a string representation of the currently loaded configuration.

`pyphi.config.print_config ()`

Print the current configuration.

`pyphi.config.configure_logging ()`

Reconfigure PyPhi logging based on the current configuration.

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

Decorator and context manager to override configuration values.

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

`pyphi.config.initialize()`

Initialize PyPhi config.

CHAPTER 18

Connectivity Matrices

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

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

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

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

State at t	$P(N = 1)$ at $t + 1$	
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**):

State at t	$P(N = 1)$ at $t + 1$	
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**.

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.

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

Methods for computing actual causation of subsystems and mechanisms.

`pyphi.actual.log2(x)`

Rounded version of \log_2 .

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

A state transition between two sets of nodes in a network.

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

Parameters

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

node_indices

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

network

Network – The network the system belongs to.

before_state

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

after_state

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

effect_system

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

cause_system

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

cause_system

Subsystem

system

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

cut

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

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

to_json()

Return a JSON-serializable representation.

apply_cut (*cut*)

Return a cut version of this transition.

cause_repertoire (*mechanism, purview*)

Return the cause repertoire.

effect_repertoire (*mechanism, purview*)

Return the effect repertoire.

unconstrained_cause_repertoire (*purview*)

Return the unconstrained cause repertoire of the occurrence.

unconstrained_effect_repertoire (*purview*)

Return the unconstrained effect repertoire of the occurrence.

repertoire (*direction, mechanism, purview*)

Returns the cause or effect repertoire function based on a direction.

Parameters *direction* (*str*) – The temporal direction, specifying the cause or effect repertoire.

state_probability (*direction, repertoire, purview*)

Compute the probability of the purview in its current state given the repertoire.

Collapses the dimensions of the repertoire that correspond to the purview nodes onto their state. All other dimension are already singular and thus receive 0 as the conditioning index.

Returns a single probability.

probability (*direction, mechanism, purview*)

Probability that the purview is in its current state given the state of the mechanism.

unconstrained_probability (*direction, purview*)

Unconstrained probability of the purview.

purview_state (*direction*)

The state of the purview when we are computing coefficients in *direction*.

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

mechanism_state (*direction*)

The state of the mechanism when we are computing coefficients in *direction*.

mechanism_indices (*direction*)

The indices of nodes in the mechanism system.

cause_ratio (*mechanism, purview*)

The cause ratio of the *purview* given mechanism.

effect_ratio (*mechanism, purview*)

The effect ratio of the *purview* given mechanism.

partitioned_repertoire (*direction, partition*)

Compute the repertoire over the partition in the given direction.

partitioned_probability (*direction, partition*)

Compute the probability of the mechanism over the purview in the partition.

find_mip (*direction, mechanism, purview, allow_neg=False*)

Find the ratio minimum information partition for a mechanism over a purview.

Parameters

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

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

Returns The found MIP.

Return type *AcMip*

potential_purviews (*direction, mechanism, purviews=False*)

Return all purviews that could belong to the core cause/effect.

Filters out trivially-reducible purviews.

Parameters

- **direction** (*str*) – Either *PAST* or *FUTURE*.
- **mechanism** (*tuple[int]*) – The mechanism of interest.

Keyword Argss: **purviews** (*tuple[int]*): Optional subset of purviews of interest.

find_causal_link (*direction, mechanism, purviews=False, allow_neg=False*)

Return the maximally irreducible cause or effect ratio for a mechanism.

Parameters

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

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

Returns The maximally-irreducible actual cause or effect.

Return type *CausalLink*

find_actual_cause (*mechanism, purviews=False*)
 Return the actual cause of a mechanism.

find_actual_effect (*mechanism, purviews=False*)
 Return the actual effect of a mechanism.

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

`pyphi.actual.directed_account` (*transition, direction, mechanisms=False, purviews=False, allow_neg=False*)
 Return the set of all `CausalLink` of the specified direction.

`pyphi.actual.account` (*transition, direction=<Direction.BIDIRECTIONAL: 2>*)
 Return the set of all causal links for a `Transition`.

Parameters `transition` (`Transition`) – The transition of interest.

Keyword Arguments `direction` (`Direction`) – By default the account contains actual causes and actual effects.

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

Parameters

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

Returns The distance between the two accounts.

Return type float

`pyphi.actual.big_acmip` (*transition, direction=<Direction.BIDIRECTIONAL: 2>*)
 Return the minimal information partition of a transition in a specific direction.

Parameters `transition` (`Transition`) – The candidate system.

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

Return type `AcBigMip`

`class pyphi.actual.FindBigAcMip` (*iterable, *context*)
 Computation engine for AC BigMips.

description = ‘Evaluating AC cuts’

empty_result (*transition, direction, unpartitioned_account*)

static compute (*cut, transition, direction, unpartitioned_account*)

process_result (*new_mip, min_mip*)

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

`pyphi.actual.nexus` (*network, before_state, after_state, direction=<Direction.BIDIRECTIONAL: 2>*)
 Return a tuple of all irreducible nexus of the network.

`pyphi.actual.causal_nexus` (*network, before_state, after_state, direction=<Direction.BIDIRECTIONAL: 2>*)
 Return the causal nexus of the network.

`pyphi.actual.nice_true_constellation` (*tc*)
 Format a true constellation.

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

`pyphi.actual.true_constellation` (*subsystem, past_state, future_state*)
Set of all sets of elements that have true causes and true effects.

Note: Since the true constellation is always about the full system, the background conditions don't matter and the subsystem should be conditioned on the current state.

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

Parameters

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

Keyword Arguments

- **indices** (*tuple[int]*) – The indices of the main complex.
- **main_complex** (*AcBigMip*) – The main complex. If `main_complex` is given then `indices` is ignored.

Returns List of true events in the main complex.

Return type `tuple[Event]`

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

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

Parameters

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

Keyword Arguments

- **indices** (*tuple[int]*) – The indices of the main complex.
- **main_complex** (*AcBigMip*) – The main complex. If `main_complex` is given then `indices` is ignored.

Returns List of extrinsic events in the main complex.

Return type `tuple(actions)`

Functions for computing integrated information and finding complexes.

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

Parameters

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

Returns The *BigMip* for that cut.

Return type *BigMip*

class `pyphi.compute.big_phi.FindMip` (*iterable*, **context*)
Computation engine for finding the minimal *BigMip*.

description = ‘Evaluating Φ cuts’

empty_result (*subsystem*, *unpartitioned_constellation*)
Begin with a mip with infinite Φ ; all actual mips will have less.

static compute (*cut*, *subsystem*, *unpartitioned_constellation*)
Evaluate a cut.

process_result (*new_mip*, *min_mip*)
Check if the new mip has smaller phi than the standing result.

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

This value changes based on `config.CUT_ONE_APPROXIMATION`.

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

Returns All unidirectional partitions.

Return type list[*Cut*]

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

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

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

Return type *BigMip*

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

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

Does not return subsystems that are in an impossible state.

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

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

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

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

Does not include subsystems in an impossible state.

Parameters

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

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

class `pyphi.compute.big_phi.FindComplexes(iterable, *context)`
 Computation engine for computing irreducible complexes of a network.

description = 'Finding complexes'

empty_result ()

static compute (*subsystem*)

process_result (*new_big_mip, complexes*)

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

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

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

class `pyphi.compute.big_phi.ConceptStyleSystem(subsystem, direction, cut=None)`
 A functional replacement for *Subsystem* implementing concept-style system cuts.

apply_cut (*cut*)

`__getattr__` (*name*)

Pass attribute access through to the basic subsystem.

cause_system

effect_system

concept (*mechanism, purviews=False, past_purviews=False, future_purviews=False*)

Compute a concept, using the appropriate system for each side of the cut.

`pyphi.compute.big_phi.concept_cuts` (*node_indices*)

Generator over all concept-style cuts for these nodes.

`pyphi.compute.big_phi.directional_big_mip` (*subsystem, direction, unpartitioned_constellation=None*)

Calculate a concept-style BigMipPast or BigMipFuture.

class `pyphi.compute.big_phi.BigMipConceptStyle` (*mip_past, mip_future, subsystem*)

Represents a Big Mip computed using concept-style system cuts.

phi

network

The network this BigMip belongs to.

unordered_unless_eq = ['network']

order_by ()

`pyphi.compute.big_phi.big_mip_concept_style` (*subsystem*)

Compute a concept-style Big Mip

Functions for computing concepts and constellations of concepts.

`pyphi.compute.concept.concept` (*subsystem, mechanism, purviews=False, past_purviews=False, future_purviews=False*)

Return the concept specified by a mechanism within a subsystem.

Parameters

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

Keyword Arguments

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

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

Return type *Concept*

class `pyphi.compute.concept.ComputeConstellation` (*iterable, *context*)

Engine for computing a constellation.

description = 'Computing concepts'

empty_result (**args*)

static compute (*mechanism, subsystem, purviews, past_purviews, future_purviews*)

Compute a concept for a mechanism, in this subsystem with the provided purviews.

process_result (*new_concept, concepts*)

Save all concepts with non-zero phi to the constellation.

`pyphi.compute.concept.constellation` (*subsystem*, *mechanisms=False*, *purviews=False*, *past_purviews=False*, *future_purviews=False*, *parallel=False*)

Return the conceptual structure of this subsystem, optionally restricted to concepts with the mechanisms and purviews given in keyword arguments.

If you don't need the full constellation, restricting the possible mechanisms and purviews can make this function much faster.

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

Keyword Arguments

- **mechanisms** (*tuple[tuple[int]]*) – Restrict possible mechanisms to those in this list.
- **purviews** (*tuple[tuple[int]]*) – Same as in `concept()`.
- **past_purviews** (*tuple[tuple[int]]*) – Same as in `concept()`.
- **future_purviews** (*tuple[tuple[int]]*) – Same as in `concept()`.
- **parallel** (*bool*) – Whether to compute concepts in parallel. If True, overrides `config.PARALLEL_CONCEPT_EVALUATION`.

Returns A tuple of every *Concept* in the constellation.

Return type *Constellation*

`pyphi.compute.concept.conceptual_information` (*subsystem*)

Return the conceptual information for a subsystem.

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

Functions for computing distances between various PyPhi objects.

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

Parameters

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

Returns The distance between the two concepts in concept space.

Return type float

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

Parameters

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

Returns The distance between the two constellations in concept space.

Return type float

`pyphi.compute.distance.small_phi_constellation_distance(C1, C2)`
Return the difference in φ between constellations.

Loading a configuration

Various aspects of PyPhi's behavior can be configured.

When PyPhi is imported, it checks for a YAML file named `pyphi_config.yml` in the current directory and automatically loads it if it exists; otherwise the default configuration is used.

The various settings are listed here with their defaults.

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

It is also possible to manually load a configuration file:

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

Or load a dictionary of configuration values:

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

Many settings can also be changed on the fly by simply assigning them a new value:

```
>>> pyphi.config.PROGRESS_BARS = True
```

Approximations and theoretical options

These settings control the algorithms PyPhi uses.

- `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
```

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

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

- **MEASURE:** The measure to use when computing distances between repertoires and concepts. Users can dynamically register new measures with the `pyphi.distance.measures.register` decorator; see *distance* for examples. A full list of currently installed measures is available by calling `print(pyphi.distance.measures.all())`. Note that some measures cannot be used for calculating Φ because they are asymmetric.

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

- **PARTITION_TYPE:** Controls the type of partition used for φ computations.

If set to 'BI', partitions will have two parts.

If set to 'TRI', partitions will have three parts. In addition, computations will only consider partitions that strictly partition the mechanism the mechanism. That is, for the mechanism (A, B) and purview (B, C, D) the partition:

```
A, B
--  --
B    C, D
```

is not considered, but:

```
A      B
--  --
B      C, D
```

is. The following is also valid:

```
A, B
--  ----
      B, C, D
```

In addition, this setting introduces “wedge” tripartitions of the form:

```
A      B
--  --  --
B      C      D
```

where the mechanism in the third part is always empty.

In addition, in the case of a φ -tie when computing MICE, The 'TRIPARTITION' setting chooses the MIP with smallest purview instead of the largest (which is the default).

Finally, if set to 'ALL', all possible partitions will be tested.

```
>>> defaults['PARTITION_TYPE']
'BI'
```

- `PICK_SMALLEST_PURVIEW`: When computing MICE, it is possible for several MIPs to have the same φ value. If this setting is set to `True` the MIP with the smallest purview is chosen; otherwise, the one with largest purview is chosen.

```
>>> defaults['PICK_SMALLEST_PURVIEW']
False
```

- `USE_SMALL_PHI_DIFFERENCE_FOR_CONSTELLATION_DISTANCE`: If set to `True`, the distance between constellations (when computing a *BigMip*) is calculated using the difference between the sum of φ in the constellations instead of the extended EMD.
- `SYSTEM_CUTS`: If set to `'3.0_STYLE'`, then traditional IIT 3.0 cuts will be used when computing Φ . If set to `'CONCEPT_STYLE'`, then experimental concept- style system cuts will be used instead.

```
>>> defaults['SYSTEM_CUTS']
'3.0_STYLE'
```

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.

- `PARALLEL_CONCEPT_EVALUATION`: Controls whether concepts are evaluated in parallel when computing constellations.

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

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

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

- `PARALLEL_COMPLEX_EVALUATION`: Controls whether systems are evaluated in parallel when computing complexes.

```
>>> defaults['PARALLEL_COMPLEX_EVALUATION']
False
```

Warning: Only one of `PARALLEL_CONCEPT_EVALUATION`, `PARALLEL_CUT_EVALUATION`, and `PARALLEL_COMPLEX_EVALUATION` can be set to `True` at a time. For maximal efficiency, you should parallelize the highest level computations possible, *e.g.*, parallelize complex evaluation instead of cut evaluation, but only if you are actually computing complexes. You should only parallelize concept evaluation if you are just computing constellations.

- `NUMBER_OF_CORES`: Controls 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
```

- `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 setting limits the percentage of a system's RAM that the caches can collectively use.

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

Caching

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

- `CACHE_BIGMIPS`: Controls whether *BigMip* objects are cached and automatically retrieved.

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

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

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

- `CACHING_BACKEND`: Controls whether precomputed results are stored and read from a local filesystem-based cache in the current directory or from a database. Set this to `'fs'` for the filesystem, `'db'` for the database.

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

- `FS_CACHE_VERBOSITY`: Controls how much caching information is printed if the filesystem cache is used. Takes a value between 0 and 11.

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

Warning: Printing during a loop iteration can slow down the loop considerably.

- `FS_CACHE_DIRECTORY`: If the filesystem is used for caching, 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 another computer, but it must be in the same directory from which Python is being run.

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

- `MONGODB_CONFIG`: Set the configuration for the MongoDB database backend (only has an effect if `CACHING_BACKEND` is `'db'`).

```
>>> defaults['MONGODB_CONFIG']['host']
'localhost'
>>> defaults['MONGODB_CONFIG']['port']
```



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

- REDIS_CACHE: Specifies whether to use Redis to cache *Mice*.

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

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

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

Logging

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

Important: After PyPhi has been imported, changing these settings will have no effect unless you call `configure_logging()` afterwards.

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

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

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

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

- LOG_FILE: Controls the name of the log file.

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

- LOG_CONFIG_ON_IMPORT: Controls whether the configuration is printed when PyPhi is imported.

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

Tip: If this is enabled and `LOG_FILE_LEVEL` is `INFO` or higher, then the log file can serve as an automatic record of which configuration settings you used to obtain results.

- `PROGRESS_BARS`: Controls whether to show progress bars on the console.

```
>>> defaults['PROGRESS_BARS']
True
```

Tip: If you are iterating over many systems rather than doing one long-running calculation, consider disabling this for speed.

Numerical precision

- `PRECISION`: If `MEASURE` is `EMD`, then the Earth Mover's Distance is calculated with an external C++ library that a numerical optimizer to find a good approximation. Consequently, systems with analytically zero Φ will sometimes be numerically found 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 $10e^{-\text{PRECISION}}$ will be considered insignificant and treated as zero. The default value is about as accurate as the EMD computations get.

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

Miscellaneous

- `VALIDATE_SUBSYSTEM_STATES`: Controls whether PyPhi checks if the subsystems's state is possible (reachable with nonzero probability 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
```

- `VALIDATE_CONDITIONAL_INDEPENDENCE`: Controls whether PyPhi checks if a system's TPM is conditionally independent.

```
>>> defaults['VALIDATE_CONDITIONAL_INDEPENDENCE']
True
```

- `SINGLE_MICRO_NODES_WITH_SELFLOOPS_HAVE_PHI`: If set to `True`, the `Phi` value of single micro-node subsystems is the difference between their unpartitioned constellation (a single concept) and the null concept. If set to `False`, their `Phi` is defined to be zero. Single macro-node subsystems may always be cut, regardless of circumstances.

```
>>> defaults['SINGLE_MICRO_NODES_WITH_SELFLOOPS_HAVE_PHI']
False
```

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

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

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

- `PRINT_FRACTIONS`: Controls whether numbers in a `repr` are printed as fractions. Numbers are still printed as decimals if the fraction's denominator would be large. This only has an effect if `REPR_VERBOSITY > 0`.

```
>>> defaults['PRINT_FRACTIONS']
True
```

The config API

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

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

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

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

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

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

`pyphi.config.configure_logging ()`
Reconfigure PyPhi logging based on the current configuration.

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

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

`pypHi.config.initialize()`

Initialize PyPhi config.

Functions for determining network connectivity properties.

`pyphi.connectivity.apply_boundary_conditions_to_cm(external_indices, cm)`
Remove connections to or from external nodes.

`pyphi.connectivity.get_inputs_from_cm(index, cm)`
Return indices of inputs to the node with the given index.

`pyphi.connectivity.get_outputs_from_cm(index, cm)`
Return indices of the outputs of node with the given index.

`pyphi.connectivity.causally_significant_nodes(cm)`
Return indices of nodes that have both inputs and outputs.

`pyphi.connectivity.relevant_connections(n, _from, to)`
Construct a connectivity matrix.

Parameters

- **n** (*int*) – The dimensions of the matrix
- **_from** (*tuple[int]*) – Nodes with outgoing connections to **to**
- **to** (*tuple[int]*) – Nodes with incoming connections from **_from**

Returns An $N \times N$ connectivity matrix with the $(i, j)^{\text{th}}$ entry is 1 if i is in **_from** and j is in **to**, and 0 otherwise.

Return type `np.ndarray`

`pyphi.connectivity.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, because the cut

A, B	C
--	--
D, E	F, G

does not change the structure of the graph.

`pyphi.connectivity.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.connectivity.is_strong` (*cm*, *nodes=None*)

Return whether the connectivity matrix is strongly connected.

Remember that a singleton graph is strongly connected.

Parameters **cm** (*np.ndarray*) – A square connectivity matrix.

Keyword Arguments **nodes** (*tuple[int]*) – A subset of nodes to consider.

`pyphi.connectivity.is_weak` (*cm*, *nodes=None*)

Return whether the connectivity matrix is weakly connected.

Parameters **cm** (*np.ndarray*) – A square connectivity matrix.

Keyword Arguments **nodes** (*tuple[int]*) – A subset of nodes to consider.

`pyphi.connectivity.is_full` (*cm*, *nodes1*, *nodes2*)

Test connectivity of one set of nodes to another.

Parameters

- **cm** (*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 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, or if either set of nodes is empty; False otherwise.

Return type bool

CHAPTER 26

constants

Package-wide constants.

class `pyphi.constants.Direction`

Constants that parametrize cause and effect methods.

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

PAST = 0

FUTURE = 1

BIDIRECTIONAL = 2

to_json()

classmethod `from_json(dct)`

`pyphi.constants.EPSILON = 1e-06`

The threshold below which we consider differences in phi values to be zero.

`pyphi.constants.FILESYSTEM = 'fs'`

Label for the filesystem cache backend.

`pyphi.constants.DATABASE = 'db'`

Label for the MongoDB cache backend.

`pyphi.constants.PICKLE_PROTOCOL = 4`

The protocol used for pickling objects.

`pyphi.constants.joblib_memory = Memory(cachedir='__pyphi_cache__/joblib')`

The joblib Memory object for persistent caching without a database.

`pyphi.constants.OFF = (0,)`

Node states

Conversion functions.

See the documentation on PyPhi *Connectivity Matrices* for information on the different representations that these functions convert between.

`pyphi.convert.reverse_bits(i, n)`
Reverse the bits of the n-bit decimal number i.

Examples

```
>>> reverse_bits(12, 7)
24
>>> reverse_bits(0, 1)
0
>>> reverse_bits(1, 2)
2
```

`pyphi.convert.nodes2indices(nodes)`
Convert nodes to a tuple of their indices.

`pyphi.convert.nodes2state(nodes)`
Convert nodes to a tuple of their states.

`pyphi.convert.holi2loli(i, n)`
Convert between HOLI and LOLI for indices in range(n).

`pyphi.convert.loli2holi(i, n)`
Convert between HOLI and LOLI for indices in range(n).

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

Return type int

Examples

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

Return type int

Examples

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

Return type `tuple[int]`

Examples

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

Return type tuple[int]

Examples

```
>>> 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.holi2loli_state_by_state` (*tpm*)

Convert a state-by-state TPM from HOLI to LOLI or vice versa.

Parameters *tpm* (*np.ndarray*) – A state-by-state TPM.

Returns The state-by-state TPM in the other indexing format.

Return type np.ndarray

Example

```
>>> tpm = np.arange(16).reshape([4, 4])
>>> holi2loli_state_by_state(tpm)
array([[ 0.,  1.,  2.,  3.],
       [ 8.,  9., 10., 11.],
       [ 4.,  5.,  6.,  7.],
       [12., 13., 14., 15.]])
```

`pyphi.convert.loli2holi_state_by_state` (*tpm*)

Convert a state-by-state TPM from HOLI to LOLI or vice versa.

Parameters *tpm* (*np.ndarray*) – A state-by-state TPM.

Returns The state-by-state TPM in the other indexing format.

Return type np.ndarray

Example

```
>>> tpm = np.arange(16).reshape([4, 4])
>>> holi2loli_state_by_state(tpm)
array([[ 0.,  1.,  2.,  3.],
       [ 8.,  9., 10., 11.],
       [ 4.,  5.,  6.,  7.],
       [12., 13., 14., 15.]])
```

`pyphi.convert.to_n_dimensional` (*tpm*)

Reshape a state-by-node TPM to the n-dimensional form.

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

`pyphi.convert.to_2_dimensional(tpm)`

Reshape a state-by-node TPM to the 2-dimensional 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.

Danger: Many nondeterministic state-by-state TPMs can be represented by a single a state-by-state TPM. However, the mapping can be made to be one-to-one if we assume the state-by-state TPM is conditionally independent, as this function does. **If the given TPM is not conditionally independent, the conditional dependencies will be silently lost.**

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 on PyPhi *Connectivity Matrices* more information.

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

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

Return type `np.ndarray`

Example

```
>>> 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.

Important: 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, **this function returns the corresponding conditionally independent state-by-state TPM.**

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. See the documentation on PyPhi *Connectivity Matrices* for more info.

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

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

Return type `np.ndarray`

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

`pyphi.convert.h2l(i, n)`

Convert between HOLI and LOLI for indices in range (n).

`pyphi.convert.l2h(i, n)`

Convert between HOLI and LOLI for indices in range (n).

`pyphi.convert.l2s(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 A state-tuple where the i^{th} element of the tuple gives the state of the i^{th} node.

Return type `tuple[int]`

Examples

```
>>> 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.h2s(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 A state-tuple where the i^{th} element of the tuple gives the state of the i^{th} node.

Return type tuple[int]

Examples

```
>>> 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.s2l`(*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 A decimal integer corresponding to a network state under the LOLI convention.

Return type int

Examples

```
>>> state2loli_index((1, 0, 0, 0, 0))
1
>>> state2loli_index((1, 1, 1, 0, 0, 0, 0, 0))
7
```

`pyphi.convert.s2h`(*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 A decimal integer corresponding to a network state under the HOLI convention.

Return type int

Examples

```
>>> state2holi_index((1, 0, 0, 0, 0))
16
>>> state2holi_index((1, 1, 1, 0, 0, 0, 0, 0))
224
```

`pyphi.convert.h2l_sbs`(*tpm*)

Convert a state-by-state TPM from HOLI to LOLI or vice versa.

Parameters *tpm* (*np.ndarray*) – A state-by-state TPM.

Returns The state-by-state TPM in the other indexing format.

Return type np.ndarray

Example

```
>>> tpm = np.arange(16).reshape([4, 4])
>>> holi2loli_state_by_state(tpm)
array([[ 0.,  1.,  2.,  3.],
       [ 8.,  9., 10., 11.],
       [ 4.,  5.,  6.,  7.],
       [12., 13., 14., 15.]])
```

`pyphi.convert.12h_sbs(tpm)`

Convert a state-by-state TPM from HOLI to LOLI or vice versa.

Parameters `tpm` (*np.ndarray*) – A state-by-state TPM.

Returns The state-by-state TPM in the other indexing format.

Return type np.ndarray

Example

```
>>> tpm = np.arange(16).reshape([4, 4])
>>> holi2loli_state_by_state(tpm)
array([[ 0.,  1.,  2.,  3.],
       [ 8.,  9., 10., 11.],
       [ 4.,  5.,  6.,  7.],
       [12., 13., 14., 15.]])
```

`pyphi.convert.to_n_d(tpm)`

Reshape a state-by-node TPM to the n-dimensional form.

See documentation for the [Network](#) object for more information on TPM formats.

`pyphi.convert.to_2_d(tpm)`

Reshape a state-by-node TPM to the 2-dimensional form.

See documentation for the [Network](#) object for more information on TPM formats.

`pyphi.convert.sbn2sbs(tpm)`

Convert a state-by-node TPM to a state-by-state TPM.

Important: 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, **this function returns the corresponding conditionally independent state-by-state TPM.**

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. See the documentation on PyPhi [Connectivity Matrices](#) for more info.

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

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

Return type `np.ndarray`

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

`pyphi.convert.sbs2sbn(tpm)`

Convert a state-by-state TPM to a state-by-node TPM.

Danger: Many nondeterministic state-by-state TPMs can be represented by a single a state-by-state TPM. However, the mapping can be made to be one-to-one if we assume the state-by-state TPM is conditionally independent, as this function does. **If the given TPM is not conditionally independent, the conditional dependencies will be silently lost.**

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 on PyPhi *Connectivity Matrices* more information.

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

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

Return type `np.ndarray`

Example

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


Functions for measuring distances.

class `pyphi.distance.MeasureRegistry`
Storage for measures registered with PyPhi.

Users can define custom measures:

Examples

```
>>> @measures.register('ALWAYS_ZERO')
... def always_zero(a, b):
...     return 0
```

And use them by setting `config.MEASURE = 'ALWAYS_ZERO'`.

register (*name*, *asymmetric=False*)
Register a measure with PyPhi.

Parameters *name* (*string*) – The name of the measure.

Keyword Arguments *asymmetric* (*boolean*) – True if the measure is asymmetric.

asymmetric ()

all ()

class `pyphi.distance.np_suppress` (**args*)
Decorator to suppress NumPy warnings about divide-by-zero and multiplication of NaN.

Note: This should only be used in cases where you are *sure* that these warnings are not indicative of deeper issues in your code.

`pyphi.distance.hamming_emd(d1, d2)`

Return the Earth Mover's Distance between two distributions (indexed by state, one dimension per node) using the Hamming distance between states as the transportation cost function.

Singleton dimensions are squeezed out.

`pyphi.distance.effect_emd(d1, d2)`

Compute the EMD between two effect repertoires.

Because the nodes are independent, the EMD between effect repertoires is equal to the sum of the EMDs between the marginal distributions of each node, and the EMD between marginal distribution for a node is the absolute difference in the probabilities that the node is off.

Parameters

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

Returns The EMD between d1 and d2.

Return type float

`pyphi.distance.l1(d1, d2)`

Return the L1 distance between two distributions.

Parameters

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

Returns The sum of absolute differences of d1 and d2.

Return type float

`pyphi.distance.kld(d1, d2)`

Return the Kullback-Leibler Divergence (KLD) between two distributions.

Parameters

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

Returns The KLD of d1 from d2.

Return type float

`pyphi.distance.entropy_difference(d1, d2)`

Return the difference in entropy between two distributions.

`pyphi.distance.psq2(d1, d2)`

Compute the PSQ2 measure.

Parameters

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

`pyphi.distance.mp2q(p, q)`

Compute the MP2Q measure.

Parameters

- **p** (*np.ndarray*) – The unpartitioned repertoire

- **q** (*np.ndarray*) – The partitioned repertoire

`pyphi.distance.bld(p, q)`

Compute the Buzz Lightyear (Billy-Leo) Divergence.

`pyphi.distance.directional_emd(direction, d1, d2)`

Compute the EMD between two repertoires for a given direction.

The full EMD computation is used for cause repertoires. A fast analytic solution is used for effect repertoires.

Parameters

- **direction** (*Direction*) – *PAST* or *FUTURE*.
- **d1** (*np.ndarray*) – The first repertoire.
- **d2** (*np.ndarray*) – The second repertoire.

Returns The EMD between d1 and d2, rounded to PRECISION.

Return type float

Raises `ValueError` – If `direction` is invalid.

`pyphi.distance.small_phi_measure(direction, d1, d2)`

Compute the distance between two repertoires for the given direction.

Parameters

- **direction** (*Direction*) – *PAST* or *FUTURE*.
- **d1** (*np.ndarray*) – The first repertoire.
- **d2** (*np.ndarray*) – The second repertoire.

Returns The distance between d1 and d2, rounded to PRECISION.

Return type float

`pyphi.distance.big_phi_measure(r1, r2)`

Compute the distance between two repertoires.

Parameters

- **r1** (*np.ndarray*) – The first repertoire.
- **r2** (*np.ndarray*) – The second repertoire.

Returns The distance between r1 and r2.

Return type float

CHAPTER 29

examples

Example networks and subsystems to go along with the documentation.

`pyphi.examples.basic_network (cm=False)`

A 3-node network of logic gates.

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()`

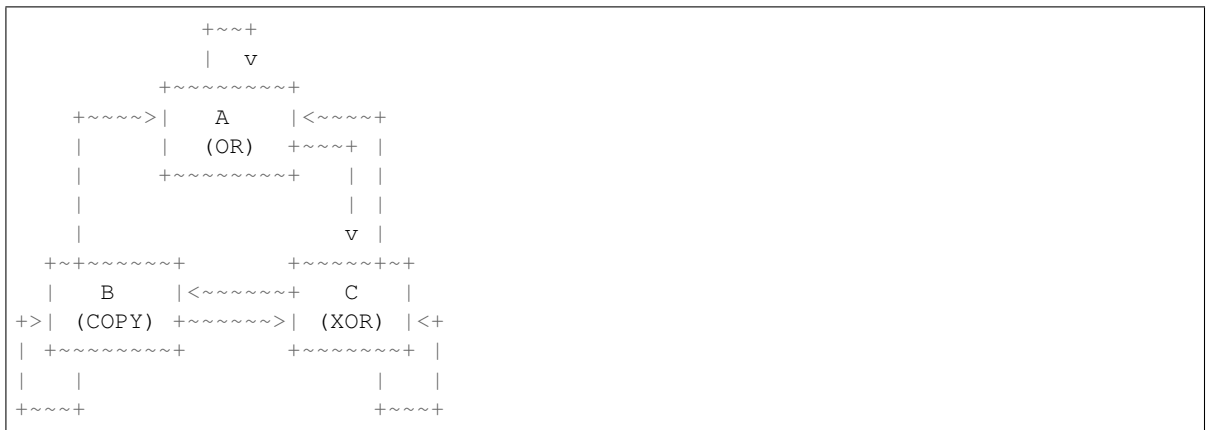
A subsystem containing all the nodes of the `basic_network()`.

`pyphi.examples.basic_noisy_selfloop_network()`

Based on the `basic_network`, but with added selfloops and noisy edges.

Nodes perform deterministic functions of their inputs, but those inputs may be flipped (i.e. what should be a 0 becomes a 1, and vice versa) with probability epsilon (eps = 0.1 here).

Diagram:



`pyphi.examples.basic_noisy_selfloop_subsystem()`

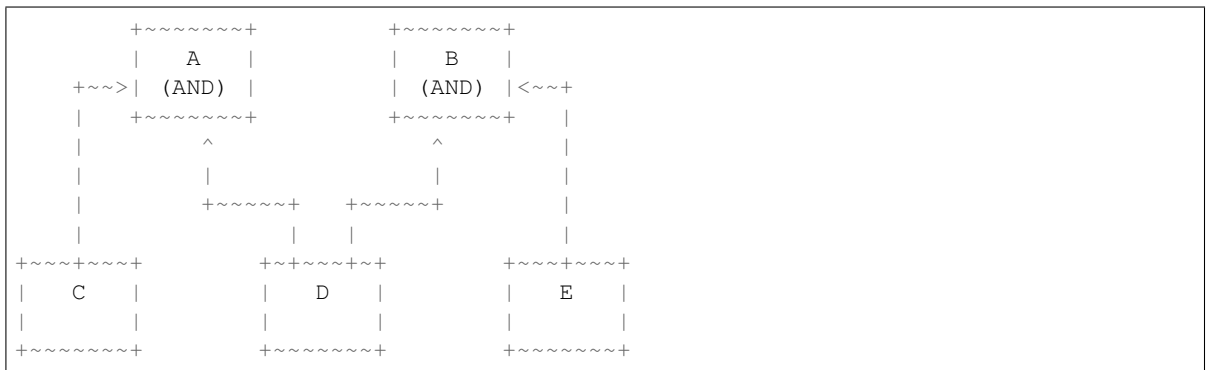
A subsystem containing all the nodes of the `basic_noisy_selfloop_network()`.

`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

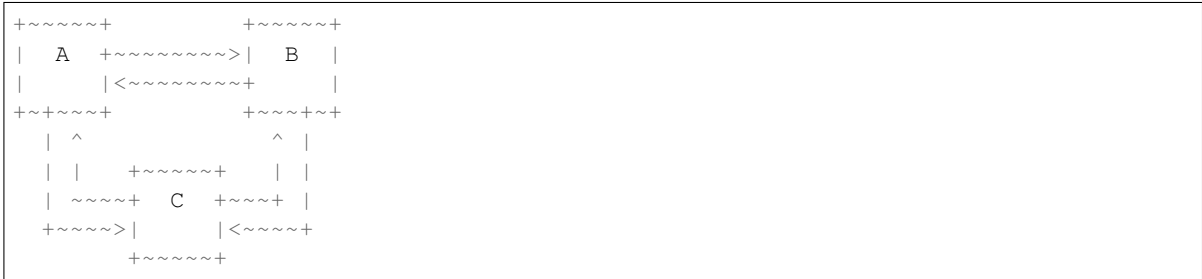
Connectivity matrix:

.	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:

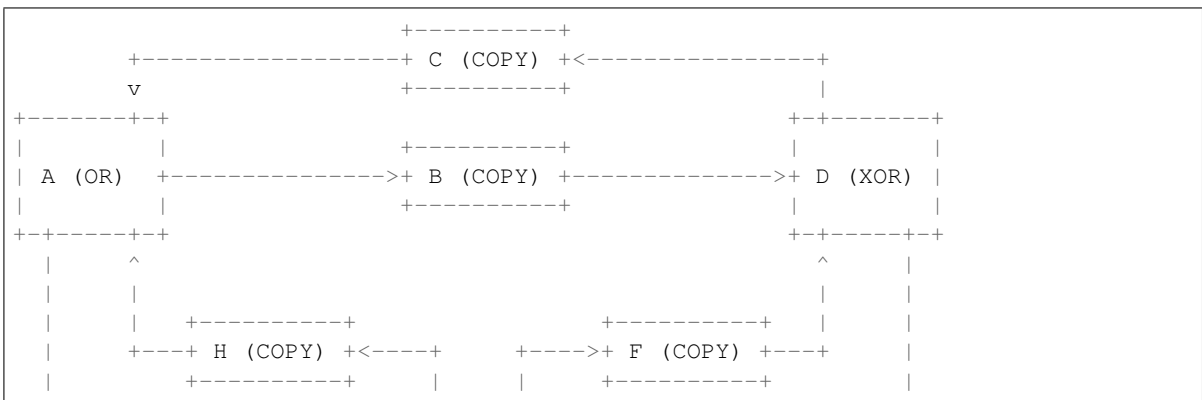
.	A	B	C
A	0	1	0
B	1	0	0
C	1	1	0

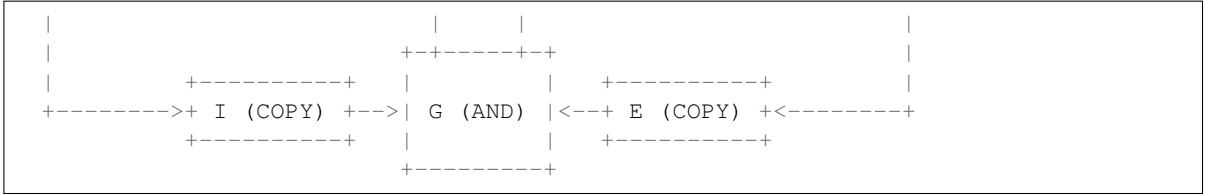
`pyphi.examples.propagation_delay_network()`

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 $(0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$.

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 $(1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$ is achieved, with corresponding past state $(0, 0, 1, 0, 1, 0, 0, 0, 0, 0)$.

`pyphi.examples.macro_network()`

A network of micro elements which has greater integrated information after coarse graining to a macro scale.

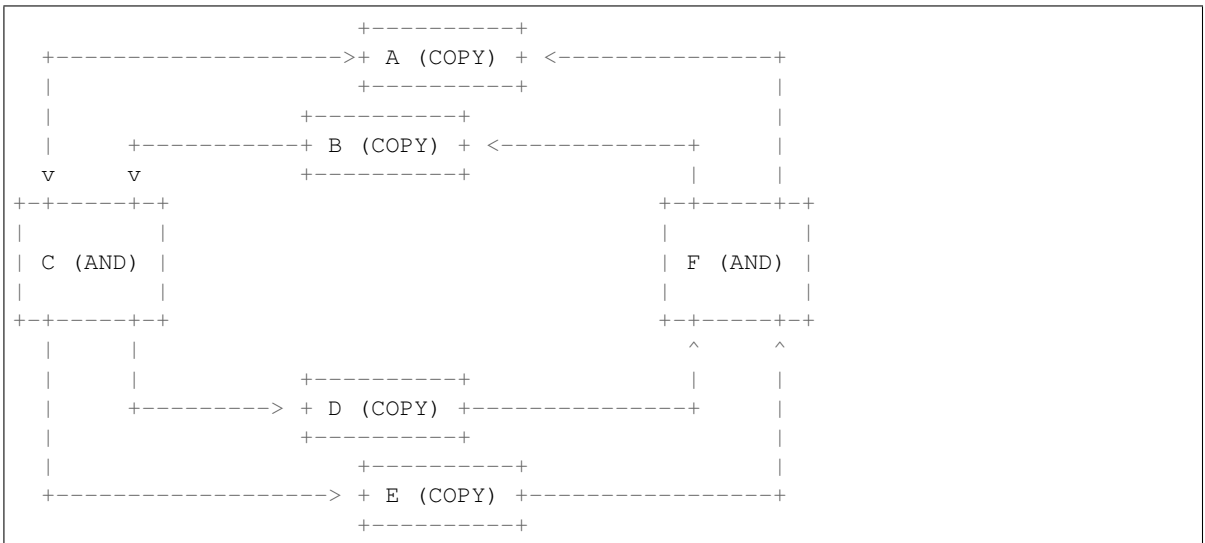
`pyphi.examples.macro_subsystem()`

A subsystem containing all the nodes of `macro_network()`.

`pyphi.examples.blackbox_network()`

A micro-network to demonstrate blackboxing.

Diagram:



Connectivity Matrix:

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

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

`pyphi.examples.rule110_network()`

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

`pyphi.examples.rule154_network()`

A network of three elements which follows the logic of the Rule 154 cellular automaton.

`pyphi.examples.fig1a()`

The network shown in Figure 1A of the 2014 IIT 3.0 paper.

`pyphi.examples.fig3a()`

The network shown in Figure 3A of the 2014 IIT 3.0 paper.

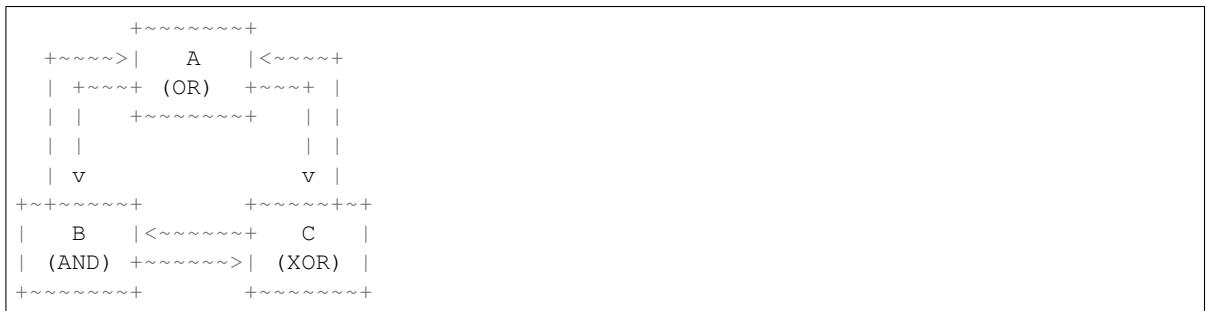
`pyphi.examples.fig3b()`

The network shown in Figure 3B of the 2014 IIT 3.0 paper.

`pyphi.examples.fig4()`

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

Diagram:



`pyphi.examples.fig5a()`

The network shown in Figure 5A of the 2014 IIT 3.0 paper.

Diagram:



`pyphi.examples.fig5b()`

The network shown in Figure 5B of the 2014 IIT 3.0 paper.

Diagram:



`pyphi.examples.fig14()`

The network shown in Figure 1A of the 2014 IIT 3.0 paper.

`pyphi.examples.fig16()`

The network shown in Figure 5B of the 2014 IIT 3.0 paper.

`pyphi.examples.actual_causation()`

The actual causation example network, consisting of an OR and AND gate with self-loops.

`pyphi.examples.disjunction_conjunction_network()`

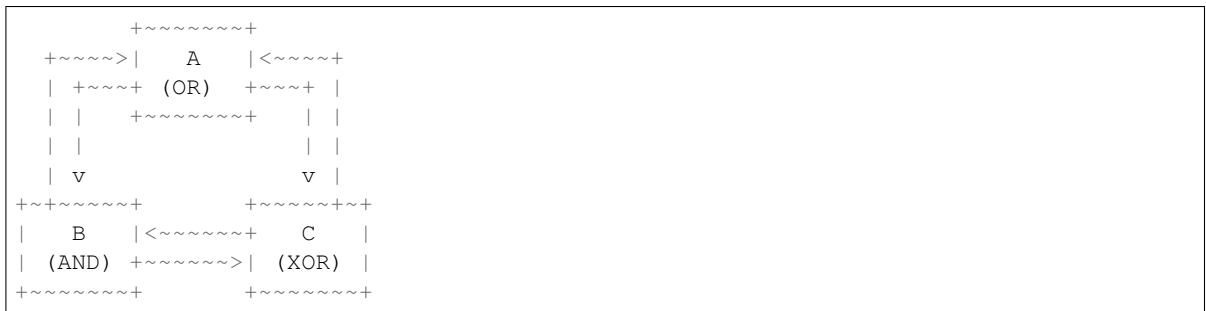
The disjunction-conjunction example from Actual Causation Figure 7.

A network of four elements, one output D with three inputs A B C. The output turns ON if A AND B are ON or if C is ON.

`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.

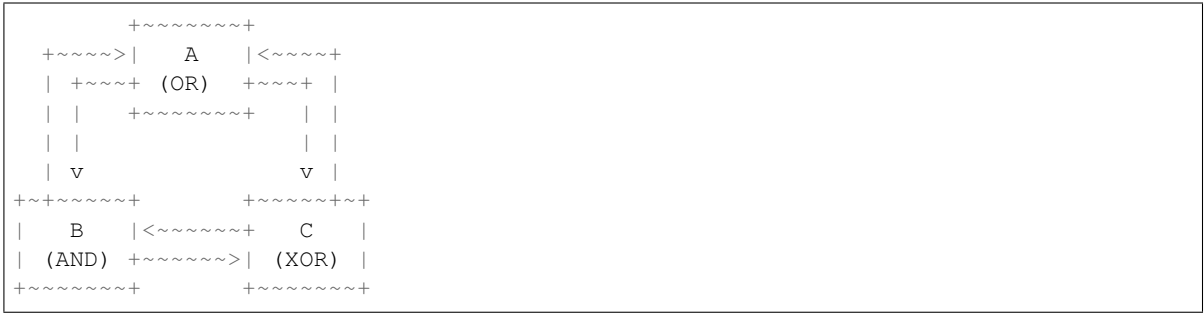
Diagram:



`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:



CHAPTER 30

exceptions

PyPhi exceptions.

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

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

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

PyPhi- and NumPy-aware JSON serialization.

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

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

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

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

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

The JSON encoder adds the name of the object and the current PyPhi version to the JSON stream. The JSON decoder uses this metadata to recursively deserialize the stream to a nested PyPhi object structure. The decoder will raise an exception if current PyPhi version doesn't match the version in the JSON data.

`pyphi.jsonify.jsonify(obj)`

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

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

JSONEncoder that allows serializing PyPhi objects with `jsonify`.

Constructor for JSONEncoder, with sensible defaults.

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

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

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

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

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

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

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

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

encode (*obj*)

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

iterencode (*obj*, ***kwargs*)

Analog to `encode` used by `json.dump`.

`pyphi.jsonify.dumps` (*obj*, ***user_kwargs*)

Serialize `obj` as JSON-formatted stream.

`pyphi.jsonify.dump` (*obj*, *fp*, ***user_kwargs*)

Serialize `obj` as a JSON-formatted stream and write to `fp` (a `.write()`-supporting file-like object).

class `pyphi.jsonify.PyPhiJSONDecoder` (**args*, ***kwargs*)

Extension of the default encoder which automatically deserializes PyPhi JSON to the appropriate model classes.

`pyphi.jsonify.loads` (*string*)

Deserialize a JSON string to a Python object.

`pyphi.jsonify.load` (*fp*)

Deserialize a JSON stream to a Python object.

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

`pyphi.macro.reindex` (*indices*)

Generate a new set of node indices, the size of indices.

`pyphi.macro.rebuild_system_tpm` (*node_tpm*s)

Reconstruct the network TPM from a collection of node TPMs.

`pyphi.macro.remove_singleton_dimensions` (*tpm*)

Remove singleton dimensions from the TPM.

Singleton dimensions are created by conditioning on a set of elements. This removes those elements from the TPM, leaving a TPM that only describes the non-conditioned elements.

Note that indices used in the original TPM must be reindexed for the smaller TPM.

`pyphi.macro.node_labels` (*indices*)

Labels for macro nodes.

`pyphi.macro.run_tpm` (*system*, *steps*, *blackbox*)

Iterate the TPM for the given number of timesteps.

Returns $tpm * (noise_tpm^{t-1})$

class `pyphi.macro.SystemAttrs`

An immutable container that holds all the attributes of a subsystem.

Versions of this object are passed down the steps of the micro-to-macro pipeline.

Create new instance of `SystemAttrs(tpm, cm, node_indices, state)`

nodes

static pack (*system*)

apply (*system*)

class `pyphi.macro.MacroSubsystem` (*network*, *state*, *nodes*, *cut=None*, *mice_cache=None*,
time_scale=1, *blackbox=None*, *coarse_grain=None*)

A subclass of `Subsystem` implementing macro computations.

This subsystem performs blackboxing and coarse-graining of elements.

Unlike *Subsystem*, whose TPM has dimensionality equal to that of the subsystem's network and represents nodes external to the system using singleton dimensions, *MacroSubsystem* squeezes the TPM to remove these singletons. As a result, the node indices of the system are also squeezed to $0..n$ so they properly index the TPM, and the state-tuple is reduced to the size of the system.

After each macro update (temporal blackboxing, spatial blackboxing, and spatial coarse-graining) the TPM, CM, nodes, and state are updated so that they correctly represent the updated system.

cut_indices

The indices of this system to be cut for Φ computations.

For macro computations the cut is applied to the underlying micro-system.

cut_mechanisms

The mechanisms of this system that are currently cut.

Note that although `cut_indices` returns micro indices, this returns macro mechanisms.

apply_cut (*cut*)

Return a cut version of this *MacroSubsystem*.

Parameters `cut` (*Cut*) – The cut to apply to this *MacroSubsystem*.

Returns The cut version of this *MacroSubsystem*.

Return type *MacroSubsystem*

potential_purviews (*direction, mechanism, purviews=False*)

Override Subsystem implementation using Network-level indices.

macro2micro (*macro_indices*)

Returns all micro indices which compose the elements specified by `macro_indices`.

macro2blackbox_outputs (*macro_indices*)

Given a set of macro elements, return the blackbox output elements which compose these elements.

__eq__ (*other*)

Two macro systems are equal if each underlying *Subsystem* is equal and all macro attributes are equal.

class `pyphi.macro.CoarseGrain`

Represents a coarse graining of a collection of nodes.

partition

tuple[tuple] – The partition of micro-elements into macro-elements.

grouping

tuple[tuple[tuple]] – The grouping of micro-states into macro-states.

Create new instance of `CoarseGrain(partition, grouping)`

micro_indices

Indices of micro elements represented in this coarse-graining.

macro_indices

Indices of macro elements of this coarse-graining.

reindex ()

Re-index this coarse graining to use squeezed indices.

The output grouping is translated to use indices $0..n$, where n is the number of micro indices in the coarse-graining. Re-indexing does not effect the state grouping, which is already index-independent.

Returns A new *CoarseGrain* object, indexed from $0..n$.

Return type *CoarseGrain*

Example

```
>>> partition = ((1, 2),)
>>> grouping = (((0,), (1, 2)),)
>>> coarse_grain = CoarseGrain(partition, grouping)
>>> coarse_grain.reindex()
CoarseGrain(partition=((0, 1),), grouping=(((0,), (1, 2)),))
```

macro_state (*micro_state*)

Translate a micro state to a macro state

Parameters **micro_state** (*tuple[int]*) – The state of the micro nodes in this coarse-graining.

Returns The state of the macro system, translated as specified by this coarse-graining.

Return type *tuple[int]*

Example

```
>>> coarse_grain = CoarseGrain(((1, 2),), (((0,), (1, 2)),))
>>> coarse_grain.macro_state((0, 0))
(0,)
>>> coarse_grain.macro_state((1, 0))
(1,)
>>> coarse_grain.macro_state((1, 1))
(1,)
```

make_mapping ()

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

Returns A mapping from micro-states to macro-states. The i^{th} entry in the mapping is the macro-state corresponding to the i^{th} micro-state.

Return type *nd.ndarray*

macro_tpm_sbs (*state_by_state_micro_tpm*)

Create a state-by-state coarse-grained macro TPM.

Parameters **micro_tpm** (*nd.array*) – The state-by-state TPM of the micro-system.

Returns The state-by-state TPM of the macro-system.

Return type *np.ndarray*

macro_tpm (*micro_tpm, check_independence=True*)

Create a coarse-grained macro TPM.

Parameters

- **micro_tpm** (*nd.array*) – The TPM of the micro-system.
- **check_independence** (*bool*) – Whether to check that the macro TPM is conditionally independent.

Raises `ConditionallyDependentError` – If `check_independence` is `True` and the macro TPM is not conditionally independent.

Returns The state-by-node TPM of the macro-system.

Return type `np.ndarray`

class `pyphi.macro.Blackbox`

Class representing a blackboxing of a system.

partition

`tuple[tuple[int]]` – The partition of nodes into boxes.

output_indices

`tuple[int]` – Outputs of the blackboxes.

Create new instance of `Blackbox(partition, output_indices)`

hidden_indices

All elements hidden inside the blackboxes.

micro_indices

Indices of micro-elements in this blackboxing.

macro_indices

Fresh indices of macro-elements of the blackboxing.

outputs_of (`partition_index`)

The outputs of the partition at `partition_index`.

Note that this returns a tuple of element indices, since coarse-grained blackboxes may have multiple outputs.

reindex ()

Squeeze the indices of this blackboxing to `0..n`.

Returns a new, reindexed `Blackbox`.

Return type `Blackbox`

Example

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

macro_state (`micro_state`)

Compute the macro-state of this blackbox.

This is just the state of the blackbox's output indices.

Parameters `micro_state` (`tuple[int]`) – The state of the micro-elements in the blackbox.

Returns The state of the output indices.

Return type `tuple[int]`

in_same_box (`a, b`)

Returns `True` if nodes `a` and `b`` are in the same box.

hidden_from (*a, b*)

Returns True if *a* is hidden in a different box than *b*.

`pyphi.macro.all_partitions` (*indices*)

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

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

Yields *tuple[tuple]* – A possible partition. Each element of the tuple is a tuple of micro-elements which correspond to macro-elements.

`pyphi.macro.all_groupings` (*partition*)

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

Parameters *partition* (*tuple[tuple]*) – A partition of micro-elements into macro elements.

Yields *tuple[tuple[tuple]]* – A grouping of micro-states into macro states of system.

TODO: document exactly how to interpret the grouping.

`pyphi.macro.all_coarse_grains` (*indices*)

Generator over all possible *CoarseGrain* of these indices.

Parameters *indices* (*tuple[int]*) – Node indices to coarse grain.

Yields *CoarseGrain* – The next *CoarseGrain* for indices.

`pyphi.macro.all_coarse_grains_for_blackbox` (*blackbox*)

Generator over all *CoarseGrain* for the given blackbox.

If a box has multiple outputs, those outputs are partitioned into the same coarse-grain macro-element.

`pyphi.macro.all_blackboxes` (*indices*)

Generator over all possible blackboxings of these indices.

Parameters *indices* (*tuple[int]*) – Nodes to blackbox.

Yields *Blackbox* – The next *Blackbox* of indices.

class `pyphi.macro.MacroNetwork` (*network, system, macro_phi, micro_phi, coarse_grain, time_scale=1, blackbox=None*)

A coarse-grained network of nodes.

See the *Emergence (coarse-graining and blackboxing)* example in the documentation for more information.

network

Network – The network object of the macro-system.

phi

float – The Φ of the network's main complex.

micro_network

Network – The network object of the corresponding micro system.

micro_phi

float – The Φ of the main complex of the corresponding micro-system.

coarse_grain

CoarseGrain – The coarse-graining of micro-elements into macro-elements.

time_scale

int – The time scale the macro-network run over.

blackbox

Blackbox – The blackboxing of micro elements in the network.

emergence

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

emergence

Difference between the Φ of the macro and micro systems

`pyphi.macro.coarse_grain` (*network, state, internal_indices*)

Find the maximal coarse-graining of a micro-system.

Parameters

- **network** (*Network*) – The network in question.
- **state** (*tuple[int]*) – The state of the network.
- **internal_indices** (*tuple[int]*) – Nodes in the micro-system.

Returns The phi-value of the maximal *CoarseGrain*.

Return type *tuple[int, CoarseGrain]*

`pyphi.macro.all_macro_systems` (*network, state, blackbox, coarse_grain, time_scales*)

Generator over all possible macro-systems for the network.

`pyphi.macro.emergence` (*network, state, blackbox=False, coarse_grain=True, time_scales=None*)

Check for the emergence of a micro-system into a macro-system.

Checks all possible blackboxings and coarse-grainings of a system to find the spatial scale with maximum integrated information.

Use the `blackbox` and `coarse_grain` args to specify whether to use blackboxing, coarse-graining, or both. The default is to just coarse-grain the system.

Parameters

- **network** (*Network*) – The network of the micro-system under investigation.
- **state** (*tuple[int]*) – The state of the network.
- **blackbox** (*bool*) – Set to `True` to enable blackboxing. Defaults to `False`.
- **coarse_grain** (*bool*) – Set to `True` to enable coarse-graining. Defaults to `True`.
- **time_scales** (*list[int]*) – List of all time steps over which to check for emergence.

Returns The maximal macro-system generated from the micro-system.

Return type *MacroNetwork*

`pyphi.macro.phi_by_grain` (*network, state*)

`pyphi.macro.effective_info` (*network*)

Return the effective information of the given network.

Note: For details, see:

Hoel, Erik P., Larissa Albantakis, and Giulio Tononi. “Quantifying causal emergence shows that macro can beat micro.” *Proceedings of the National Academy of Sciences* 110.49 (2013): 19790-19795.

Available online: doi: [10.1073/pnas.1314922110](https://doi.org/10.1073/pnas.1314922110).

`models.actual_causation`

Objects that represent structures used in actual causation.

`pyphi.models.actual_causation.greater_than_zero(alpha)`
Return True if alpha is greater than zero, accounting for numerical errors.

class `pyphi.models.actual_causation.AcMip`
A minimum information partition for `ac_coef` calculation.

These can be compared with the built-in Python comparison operators (`<`, `>`, etc.). First, α values are compared. Then, if these are equal up to `PRECISION`, the size of the mechanism is compared.

alpha
float – This is the difference between the mechanism’s unpartitioned and partitioned actual probability.

state
tuple[int] – state of system in specified direction (past, future)

direction
str – The temporal direction specifying whether this AcMIP should be calculated with cause or effect repertoires.

mechanism
tuple[int] – The mechanism over which to evaluate the AcMIP.

purview
tuple[int] – The purview over which the unpartitioned actual probability differs the least from the actual probability of the partition.

partition
tuple[Part, Part] – The partition that makes the least difference to the mechanism’s repertoire.

probability
float – The probability of the state in the past/future.

partitioned_probability
float – The probability of the state in the partitioned repertoire.

Create new instance of `AcMip(alpha, state, direction, mechanism, purview, partition, probability, partitioned_probability)`

unordered_unless_eq = ['direction']

order_by ()

__bool__ ()

An *AcMip* is True if it has $\alpha > 0$.

phi

Alias for α for PyPhi utility functions.

to_json ()

Return a JSON-serializable representation.

class `pyphi.models.actual_causation.CausalLink(mip)`

A maximally irreducible actual cause or effect.

These can be compared with the built-in Python comparison operators (`<`, `>`, etc.). First, α values are compared. Then, if these are equal up to `PRECISION`, the size of the mechanism is compared.

alpha

float – The difference between the mechanism’s unpartitioned and partitioned actual probabilities.

phi

Alias for α for PyPhi utility functions.

direction

Direction – Either *PAST* or *FUTURE*.

mechanism

list[int] – The mechanism for which the action is evaluated.

purview

list[int] – The purview over which this mechanism’s α is maximal.

mip

AcMip – The minimum information partition for this mechanism.

unordered_unless_eq = ['direction']

order_by ()

__bool__ ()

An *CausalLink* is True if $\alpha > 0$.

to_json ()

Return a JSON-serializable representation.

class `pyphi.models.actual_causation.Event`

A mechanism which has both an actual cause and an actual effect.

actual_cause

CausalLink – The actual cause of the mechanism.

actual_effect

CausalLink – The actual effect of the mechanism.

Create new instance of `Event(actual_cause, actual_effect)`

mechanism

The mechanism of the event.

```

class pyphi.models.actual_causation.Account
    The set of CausalLink with  $\alpha > 0$ . This includes both actual causes and actual effects.

    irreducible_causes
        The set of irreducible causes in this Account.

    irreducible_effects
        The set of irreducible effects in this Account.

    to_json()

    classmethod from_json(dct)

class pyphi.models.actual_causationDirectedAccount
    The set of CausalLink with  $\alpha > 0$  for one direction of a transition.

class pyphi.models.actual_causation.AcBigMip(alpha=None, direction=None, un-
    partitioned_account=None, parti-
    tioned_account=None, transition=None,
    cut=None)

    A minimum information partition for  $\mathcal{A}$  calculation.

    alpha
        float – The  $\mathcal{A}$  value for the transition when taken against this MIP, i.e. the difference between the unparti-
        tioned account and this MIP’s partitioned account.

    unpartitioned_account
        Account – The account of the whole transition.

    partitioned_account
        Account – The account of the partitioned transition.

    transition
        Transition – The transition this MIP was calculated for.

    cut
        ActualCut – The minimal partition.

    before_state
        Return the actual past state of the Transition.

    after_state
        Return the actual current state of the Transition.

    unorderable_unless_eq = ['direction']

    order_by()

    __bool__()
        An AcBigMip is True if it has  $\mathcal{A} > 0$ .

    to_json()

```


Objects that represent cause-effect structures.

```
class pyphi.models.big_phi.BigMip(phi=None, unpartitioned_constellation=None, par-
    tioned_constellation=None, subsystem=None,
    cut_subsystem=None, time=None, small_phi_time=None)
```

A minimum information partition for Φ calculation.

These can be compared with the built-in Python comparison operators (<, >, etc.). First, Φ values are compared. Then, if these are equal up to PRECISION, the one with the larger subsystem is greater.

phi

float – The Φ value for the subsystem when taken against this MIP, *i.e.* the difference between the unpartitioned constellation and this MIP’s partitioned constellation.

unpartitioned_constellation

Constellation – The constellation of the whole subsystem.

partitioned_constellation

Constellation – The constellation when the subsystem is cut.

subsystem

Subsystem – The subsystem this MIP was calculated for.

cut_subsystem

Subsystem – The subsystem with the minimal cut applied.

time

float – The number of seconds it took to calculate.

small_phi_time

float – The number of seconds it took to calculate the unpartitioned constellation.

print (*constellations=True*)

Print this BigMip, optionally without constellations.

cut

The unidirectional cut that makes the least difference to the subsystem.

network

The network this *BigMip* belongs to.

unordered_unless_eq = ['network']

order_by ()

__bool__ ()

A *BigMip* is True if it has $\Phi > 0$.

to_json ()

Return a JSON-serializable representation.

Objects that represent parts of cause-effect structures.

class `pyphi.models.concept.Mip` (*phi*, *direction*, *mechanism*, *purview*, *partition*, *unpartitioned_repertoire*, *partitioned_repertoire*, *subsystem=None*)

A minimum information partition for φ calculation.

These can be compared with the built-in Python comparison operators (<, >, etc.). First, φ values are compared. Then, if these are equal up to PRECISION, the size of the mechanism is compared (see the PICK_SMALLEST_PURVIEW option in *config*.)

phi

float – This is the difference between the mechanism’s unpartitioned and partitioned repertoires.

direction

Direction – PAST or FUTURE.

mechanism

tuple[int] – The mechanism over which to evaluate the MIP.

purview

tuple[int] – The purview over which the unpartitioned repertoire differs the least from the partitioned repertoire.

partition

KPartition – The partition that makes the least difference to the mechanism’s repertoire.

unpartitioned_repertoire

np.ndarray – The unpartitioned repertoire of the mechanism.

partitioned_repertoire

np.ndarray – The partitioned repertoire of the mechanism. This is the product of the repertoires of each part of the partition.

subsystem

Subsystem – The *Subsystem* this MIP belongs to.

unordered_unless_eq = ['direction']

order_by ()

__bool__ ()

A *Mip* is True if it has $\varphi > 0$.

to_json ()

class `pyphi.models.concept.Mice` (*mip*)

A maximally irreducible cause or effect.

These can be compared with the built-in Python comparison operators (<, >, etc.). First, φ values are compared. Then, if these are equal up to PRECISION, the size of the mechanism is compared (see the PICK_SMALLEST_PURVIEW option in *config*.)

phi

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

direction

Direction – PAST or FUTURE.

mechanism

list[int] – The mechanism for which the MICE is evaluated.

purview

list[int] – The purview over which this mechanism’s φ is maximal.

partition

KPartition – The partition that makes the least difference to the mechanism’s repertoire.

repertoire

np.ndarray – The unpartitioned repertoire of the mechanism over the purview.

partitioned_repertoire

np.ndarray – The partitioned repertoire of the mechanism over the purview.

mip

MIP – The minimum information partition for this mechanism.

unordered_unless_eq = ['direction']

order_by ()

to_json ()

damaged_by_cut (*subsystem*)

Return True if this *Mice* is affected by the subsystem’s cut.

The cut affects the *Mice* if it either splits the *Mice*’s mechanism or splits the connections between the purview and mechanism.

class `pyphi.models.concept.Concept` (*mechanism=None, cause=None, effect=None, subsystem=None, time=None*)

A the maximally irreducible cause and effect specified by a mechanism.

These can be compared with the built-in Python comparison operators (<, >, etc.). First, φ values are compared. Then, if these are equal up to PRECISION, the size of the mechanism is compared (see the PICK_SMALLEST_PURVIEW option in *config*.)

mechanism

tuple[int] – The mechanism that the concept consists of.

cause

Mice – The *Mice* representing the core cause of this concept.

effect

Mice – The *Mice* representing the core effect of this concept.

subsystem

Subsystem – This concept’s parent subsystem.

time

float – The number of seconds it took to calculate.

phi

float – The size of the concept.

This is the minimum of the φ values of the concept’s core cause and core effect.

cause_purview

tuple[int] – The cause purview.

effect_purview

tuple[int] – The effect purview.

cause_repertoire

np.ndarray – The cause repertoire.

effect_repertoire

np.ndarray – The effect repertoire.

unordered_unless_eq = ['subsystem']**order_by ()****__bool__ ()**

A concept is `True` if $\varphi > 0$.

eq_repertoires (other)

Return whether this concept has the same repertoires as another.

Warning: This only checks if the cause and effect repertoires are equal as arrays; mechanisms, purviews, or even the nodes that the mechanism and purview indices refer to, might be different.

emd_eq (other)

Return whether this concept is equal to another in the context of an EMD calculation.

expand_cause_repertoire (new_purview=None)

See `expand_repertoire ()`.

expand_effect_repertoire (new_purview=None)

See `expand_repertoire ()`.

expand_partitioned_cause_repertoire ()

See `expand_repertoire ()`.

expand_partitioned_effect_repertoire ()

See `expand_repertoire ()`.

to_json ()

Return a JSON-serializable representation.

classmethod from_json (dct)**class pyphi.models.concept.Constellation**

A constellation of concepts.

This is a wrapper around a tuple to provide a nice string representation and place to put constellation methods. Previously, constellations were represented as a `tuple[concept]`; this usage still works in all functions.

Normalize the order of concepts in the constellation.

static `__new__` (*concepts=()*)

Normalize the order of concepts in the constellation.

to_json ()

mechanisms

The mechanism of each concept.

phis

The φ values of each concept.

labeled_mechanisms

The labeled mechanism of each concept.

classmethod `from_json` (*json*)

`pyphi.models.concept.normalize_constellation` (*constellation*)

Deterministically reorder the concepts in a constellation.

Parameters `constellation` (`Constellation`) – The constellation in question.

Returns The constellation, ordered lexicographically by mechanism.

Return type `Constellation`

Objects that represent partitions of sets of nodes.

class `pypHi.models.cuts.Cut`
Represents a unidirectional cut.

from_nodes

tuple[int] – Connections from this group of nodes to those in `to_nodes` are from `from_nodes`.

to_nodes

tuple[int] – Connections to this group of nodes from those in `from_nodes` are from `from_nodes`.

Create new instance of `Cut(from_nodes, to_nodes)`

indices

Returns the indices of this cut.

cut_matrix (*n*)

Compute the cut matrix for this cut.

The cut matrix is a square matrix which represents connections severed by the cut.

Parameters `n` (*int*) – The size of the network.

Example

```
>>> cut = Cut((1,), (2,))
>>> cut.cut_matrix(3)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])
```

to_json ()

Return a JSON-serializable representation.

class `pypHi.models.cuts.KCut` (*partition*)

A cut that severs all connections between parts of a `K`-partition.

Note: since the `KCut` does not have a direction associated with it, connectivity is always considered to be from the purview of partition to the mechanism of the partition.

TODO: add a `direction` to the cut?

indices

cut_matrix (*n*)

The matrix of connections that are severed by this cut.

class `pyphi.models.cuts.ActualCut`

Represents an cut for a *Transition*.

This is a bipartition of the cause and effect elements.

cause_part1

tuple[int] – Connections from this group to those in *effect_part2* are cut.

cause_part2

tuple[int] – Connections from this group to those in *effect_part1* are cut.

effect_part1

tuple[int] – Connections to this group from *cause_part2* are cut.

effect_part2

tuple[int] – Connections to this group from *cause_part1* are cut.

Create new instance of `ActualCut(cause_part1, cause_part2, effect_part1, effect_part2)`

indices

tuple[int] – The indices in this cut.

cut_matrix (*n*)

The matrix of connections severed by this cut.

to_json ()

class `pyphi.models.cuts.Part`

Represents one part of a *Bipartition*.

mechanism

tuple[int] – The nodes in the mechanism for this part.

purview

tuple[int] – The nodes in the mechanism for this part.

Example

When calculating φ of a 3-node subsystem, we partition the system in the following way:

```
mechanism:  A,C      B
            --  --
purview:    B      A,C
```

This class represents one term in the above product.

Create new instance of `Part(mechanism, purview)`

to_json ()

Return a JSON-serializable representation.

```
class pyphi.models.cuts.KPartition
    A partition with an arbitrary number of parts.

    Construct the base tuple with multiple Part arguments.

    static __new__ (*args)
        Construct the base tuple with multiple Part arguments.

    __getnewargs__ ()
        And support unpickling with this __new__ signature.

    mechanism
        tuple[int] – The nodes of the mechanism in the partition.

    purview
        tuple[int] – The nodes of the purview in the partition.

    to_json ()

class pyphi.models.cuts.Bipartition
    A bipartition of a mechanism and purview.

    part0
        Part – The first part of the partition.

    part1
        Part – The second part of the partition.

    Construct the base tuple with multiple Part arguments.

    to_json ()
        Return a JSON-serializable representation.

    classmethod from_json (json)

class pyphi.models.cuts.Tripartition
    Construct the base tuple with multiple Part arguments.
```


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

```
class pyphi.network.Network(tpm, connectivity_matrix=None, node_labels=None,
                             purview_cache=None)
```

A network of nodes.

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

Parameters `tpm` (*np.ndarray*) – The transition probability matrix of the network.

The TPM can be provided in either state-by-node (either 2-dimensional or n-dimensional) or state-by-state form. In either form, row indices must follow the LOLI convention (see *LOLI: Low-Order bits correspond to Low-Index nodes*). In state-by-state form column indices must also follow the LOLI convention.

If given in state-by-node form, the TPM can be either 2-dimensional, so that `tpm[i]` gives the probabilities of each node being on if the past state is encoded by *i* according to LOLI, or in n-dimensional 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-dimensional form of the TPM must be $[2] * N + [N]$, where *S* is the number of states and *N* is the number of nodes in the network.

Keyword Arguments

- **connectivity_matrix** (*np.ndarray*) – A square binary adjacency matrix indicating the connections between nodes in the network. `connectivity_matrix[i][j] == 1` means that node *i* is connected to node *j*. If no connectivity matrix is given, every node is connected to every node (**including itself**).
- **node_labels** (*tuple[str]*) – Human-readable labels for each node in the network.

Example

In a 3-node network, `a_network.tpm[(0, 0, 1)]` gives the transition probabilities for each node at t given that state at $t - 1$ was $(n_0 = 0, n_1 = 0, n_2 = 1)$.

tpm

np.ndarray – The network’s transition probability matrix, in n-dimensional form.

cm

np.ndarray – The network’s connectivity matrix.

A square binary adjacency matrix indicating the connections between nodes in the network.

connectivity_matrix

np.ndarray – Alias for `cm`.

causally_significant_nodes

See `pyphi.connectivity.causally_significant_nodes()`.

size

int – The number of nodes in the network.

num_states

int – The number of possible states of the network.

node_indices

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

This is equivalent to `tuple(range(network.size))`.

node_labels

tuple[str] – The labels of nodes in the network.

labels2indices (*labels*)

Convert a tuple of node labels to node indices.

indices2labels (*indices*)

Convert a tuple of node indices to node labels.

parse_node_indices (*nodes*)

Returns the nodes indices for nodes, where *nodes* is either already integer indices or node labels.

potential_purviews (*direction, mechanism*)

All purviews which are not clearly reducible for mechanism.

Parameters

- **direction** (*Direction*) – *PAST* or *FUTURE*.
- **mechanism** (*tuple[int]*) – The mechanism which all purviews are checked for reducibility over.

Returns All purviews which are irreducible over *mechanism*.

Return type `list[tuple[int]]`

__eq__ (*other*)

Return whether this network equals the other object.

Networks are equal if they have the same TPM and CM.

to_json ()

Return a JSON-serializable representation.

classmethod `from_json` (*json_dict*)

Return a *Network* object from a JSON dictionary representation.

`pyphi.network.irreducible_purviews` (*cm, direction, mechanism, purviews*)

Returns all purview which are irreducible for the mechanism.

Parameters

- **cm** (*np.ndarray*) – An $N \times N$ connectivity matrix.
- **direction** (*Direction*) – *PAST* or *FUTURE*.
- **purviews** (*list[tuple[int]]*) – The purviews to check.
- **mechanism** (*tuple[int]*) – The mechanism in question.

Returns All purviews in purviews which are not reducible over mechanism.

Return type list[tuple[int]]

Raises *ValueError* – If *direction* is invalid.

`pyphi.network.from_json` (*filename*)

Convert a JSON network to a PyPhi network.

Parameters **filename** (*str*) – A path to a JSON file representing a network.

Returns The corresponding PyPhi network object.

Return type *Network*

Represents a node in a network. Each node has a unique index, its position in the network's list of nodes.

class `pyphi.node.Node` (*tpm, cm, index, state, label*)
 A node in a subsystem.

Parameters

- **tpm** (*np.ndarray*) – The TPM of the subsystem.
- **cm** (*np.ndarray*) – The CM of the subsystem.
- **index** (*int*) – The node's index in the network.
- **state** (*int*) – The state of this node.
- **label** (*str*) – An optional label for the node.

tpm

np.ndarray – The node TPM is an array with shape $2^{(n_inputs)}$ -by-2 matrix, where `node.tpm[i][j]` gives the marginal probability that the node is in state *j* at *t*+1 if the state of its inputs is *i* at *t*. If the node is a single element with a cut selfloop, (i.e. it has no inputs), the tpm is simply its unconstrained effect repertoire.

tpm_off

The TPM of this node containing only the 'OFF' probabilities.

tpm_on

The TPM of this node containing only the 'ON' probabilities.

inputs

The set of nodes with connections to this node.

outputs

The set of nodes this node has connections to.

__eq__ (*other*)

Return whether this node equals the other object.

Two nodes are equal if they belong to the same subsystem and have the same index (their TPMs must be the same in that case, so this method doesn't need to check TPM equality).

Labels are for display only, so two equal nodes may have different labels.

to_json()

Return a JSON-serializable representation.

`pyphi.node.default_label(index)`

Default label for a node.

`pyphi.node.default_labels(indices)`

Default labels for several nodes.

`pyphi.node.generate_nodes(tpm, cm, network_state, labels=None)`

Generate *Node* objects for a subsystem.

Parameters

- **tpm** (*np.ndarray*) – The system's TPM
- **cm** (*np.ndarray*) – The corresponding CM.
- **network_state** (*tuple*) – The state of the network.

Keyword Arguments **labels** (*tuple[str]*) – Textual labels for each node.

Returns The nodes of the system.

Return type *tuple[Node]*

`pyphi.node.expand_node_tpm(tpm)`

Broadcast a node TPM over the full network.

This is different from broadcasting the TPM of a full system since the last dimension (containing the state of the node) contains only the probability of *this* node being on, rather than the probabilities for each node.

Functions for generating partitions.

`pyphi.partition.partitions` (*collection*)
Generate all set partitions of a collection.

Example

```
>>> list(partitions(range(3)))
[[[0, 1, 2]],
 [[0], [1, 2]],
 [[0, 1], [2]],
 [[1], [0, 2]],
 [[0], [1], [2]]]
```

`pyphi.partition.bipartition_indices` (*N*)
Return indices for undirected bipartitions of a sequence.

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

Returns A list of tuples containing the indices for each of the two parts.

Return type list

Example

```
>>> N = 3
>>> bipartition_indices(N)
[((), (0, 1, 2)), ((0,), (1, 2)), ((1,), (0, 2)), ((0, 1), (2,))]
```

`pyphi.partition.bipartition` (*seq*)
Return a list of bipartitions for a sequence.

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

Returns A list of tuples containing each of the two partitions.

Return type list[tuple[tuple]]

Example

```
>>> bipartition((1,2,3))
[(() , (1, 2, 3)), ((1, ), (2, 3)), ((2, ), (1, 3)), ((1, 2), (3, ))]
```

`pyphi.partition.directed_bipartition_indices(N)`

Return indices for directed bipartitions of a sequence.

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

Returns A list of tuples containing the indices for each of the two parts.

Return type list

Example

```
>>> N = 3
>>> directed_bipartition_indices(N)
[(() , (0, 1, 2)),
 ((0, ), (1, 2)),
 ((1, ), (0, 2)),
 ((0, 1), (2, )),
 ((2, ), (0, 1)),
 ((0, 2), (1, )),
 ((1, 2), (0, )),
 ((0, 1, 2), ())]
```

`pyphi.partition.directed_bipartition(seq, nontrivial=False)`

Return a list of directed bipartitions for a sequence.

Parameters `seq` (*Iterable*) – The sequence to partition.

Returns A list of tuples containing each of the two parts.

Return type list[tuple[tuple]]

Example

```
>>> directed_bipartition((1, 2, 3))
[(() , (1, 2, 3)),
 ((1, ), (2, 3)),
 ((2, ), (1, 3)),
 ((1, 2), (3, )),
 ((3, ), (1, 2)),
 ((1, 3), (2, )),
 ((2, 3), (1, )),
 ((1, 2, 3), ())]
```

`pyphi.partition.bipartition_of_one(seq)`

Generate bipartitions where one part is of length 1.

`pyphi.partition.reverse_elements(seq)`

Reverse the elements of a sequence.

`pyphi.partition.directed_bipartition_of_one(seq)`

Generate directed bipartitions where one part is of length 1.

Parameters `seq` (*Iterable*) – The sequence to partition.

Returns A list of tuples containing each of the two partitions.

Return type `list[tuple[tuple]]`

Example

```
>>> partitions = directed_bipartition_of_one((1, 2, 3))
>>> list(partitions)
[(1,), (2, 3)],
 (2,), (1, 3)],
 (3,), (1, 2)],
 (2, 3), (1,)],
 (1, 3), (2,)],
 (1, 2), (3,)]
```

`pyphi.partition.directed_tripartition_indices(N)`

Return indices for directed tripartitions of a sequence.

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

Returns A list of tuples containing the indices for each partition.

Return type `list[tuple]`

Example

```
>>> N = 1
>>> directed_tripartition_indices(N)
[((0,), (), ()), ((), (0,), ()), ((), (), (0,))]
```

`pyphi.partition.directed_tripartition(seq)`

Generator over all directed tripartitions of a sequence.

Parameters `seq` (*Iterable*) – a sequence.

Yields `tuple[tuple]` – A tripartition of `seq`.

Example

```
>>> seq = (2, 5)
>>> list(directed_tripartition(seq))
[(2, 5), (), ()],
 (2,), (5,), ()],
 (2,), (), (5,)],
 (5,), (2,), ()],
 (), (2, 5), ()],
 (), (2,), (5,)],
 (5,), (), (2,)]
```

```
((), (5,), (2,)),  
((), (), (2, 5))]
```

`pyphi.partition.k_partitions(collection, k)`
Generate all k-partitions of a collection.

Example

```
>>> list(k_partitions(range(3), 2))  
[[[0, 1], [2]], [[0], [1, 2]], [[0, 2], [1]]]
```


Represents a candidate system for φ and Φ evaluation.

class `pyphi.subsystem.Subsystem`(*network*, *state*, *nodes*, *cut=None*, *mice_cache=None*, *repertoire_cache=None*, *single_node_repertoire_cache=None*)

A set of nodes in a network.

Parameters

- **network** (*Network*) – The network the subsystem belongs to.
- **state** (*tuple[int]*) – The state of the network.
- **nodes** (*tuple[int]* or *tuple[str]*) – The nodes of the network which are in this subsystem. Nodes can be specified either as indices or as labels if the *Network* was passed *node_labels*.

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

network

Network – The network the subsystem belongs to.

tpm

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

cm

np.ndarray – The connectivity matrix after applying the cut.

state

tuple[int] – The state of the network.

node_indices

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

cut

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

cut_matrix

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

null_cut

Cut – The cut object representing no cut.

nodes

tuple[Node] – The nodes in this *Subsystem*.

proper_state

tuple[int] – The state of the subsystem.

`proper_state[i]` gives the state of the i^{th} node **in the subsystem**. Note that this is **not** the state of `nodes[i]`.

connectivity_matrix

np.ndarray – Alias for `Subsystem.cm`.

size

int – The number of nodes in the subsystem.

is_cut

bool – True if this *Subsystem* has a cut applied to it.

cut_indices

tuple[int] – The nodes of this subsystem to cut for Φ computations.

This was added to support `MacroSubsystem`, which cuts indices other than `node_indices`.

cut_mechanisms

list[tuple[int]] – The mechanisms that are cut in this system.

tpm_size

int – The number of nodes in the TPM.

cache_info()

Report repertoire cache statistics.

clear_caches()

Clear the mice and repertoire caches.

__bool__()

Return `False` if the *Subsystem* has no nodes, `True` otherwise.

__eq__(other)

Return whether this *Subsystem* is equal to the other object.

Two *Subsystem*s are equal if their sets of nodes, networks, and cuts are equal.

__lt__(other)

Return whether this subsystem has fewer nodes than the other.

__gt__(other)

Return whether this subsystem has more nodes than the other.

__len__()

Return the number of nodes in this *Subsystem*.

to_json()

Return a JSON-serializable representation.

apply_cut(cut)

Return a cut version of this *Subsystem*.

Parameters `cut` (*Cut*) – The cut to apply to this *Subsystem*.

Returns The cut subsystem.

Return type *Subsystem*

indices2nodes (*indices*)

Return *Node* for these indices.

Parameters **indices** (*tuple[int]*) – The indices in question.

Returns The *Node* objects corresponding to these indices.

Return type *tuple[Node]*

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

indices2labels (*indices*)

Returns the node labels for these indices.

cause_repertoire (*mechanism, purview*)

Return the cause repertoire of a mechanism over a purview.

Parameters

- **mechanism** (*tuple[int]*) – The mechanism for which to calculate the cause repertoire.
- **purview** (*tuple[int]*) – The purview over which to calculate the cause repertoire.

Returns The cause repertoire of the mechanism over the purview.

Return type *np.ndarray*

Note: The returned repertoire is a distribution over purview node states, not the states of the whole network.

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 The effect repertoire of the mechanism over the purview.

Return type *np.ndarray*

Note: The returned repertoire is a distribution over purview node states, not the states of the whole network.

repertoire (*direction, mechanism, purview*)

Return the cause or effect repertoire based on a direction.

Parameters

- **direction** (*Direction*) – *PAST* or *FUTURE*.
- **mechanism** (*tuple[int]*) – The mechanism for which to calculate the repertoire.
- **purview** (*tuple[int]*) – The purview over which to calculate the repertoire.

Returns The cause or effect repertoire of the mechanism over the purview.

Return type np.ndarray

Raises ValueError – If `direction` is invalid.

unconstrained_repertoire (*direction, purview*)

Return the unconstrained cause/effect repertoire over a purview.

unconstrained_cause_repertoire (*purview*)

Return the unconstrained cause repertoire for a purview.

This is just the cause repertoire in the absence of any mechanism.

unconstrained_effect_repertoire (*purview*)

Return the unconstrained effect repertoire for a purview.

This is just the effect repertoire in the absence of any mechanism.

partitioned_repertoire (*direction, partition*)

Compute the repertoire of a partitioned mechanism and purview.

expand_repertoire (*direction, repertoire, new_purview=None*)

Distribute an effect repertoire over a larger purview.

Parameters

- **direction** (*Direction*) – *PAST* or *FUTURE*.
- **repertoire** (*np.ndarray*) – The repertoire to expand.

Keyword Arguments **new_purview** (*tuple[int]*) – The new purview to expand the repertoire over. If *None* (the default), the new purview is the entire network.

Returns A distribution over the new purview, where probability is spread out over the new nodes.

Return type np.ndarray

Raises ValueError – If the expanded purview doesn't contain the original purview.

expand_cause_repertoire (*repertoire, new_purview=None*)

Same as `expand_repertoire()` with `direction` set to *PAST*.

expand_effect_repertoire (*repertoire, new_purview=None*)

Same as `expand_repertoire()` with `direction` set to *FUTURE*.

cause_info (*mechanism, purview*)

Return the cause information for a mechanism over a purview.

effect_info (*mechanism, purview*)

Return the effect information for a mechanism over a purview.

cause_effect_info (*mechanism, purview*)

Return the cause-effect information for a mechanism over a purview.

This is the minimum of the cause and effect information.

evaluate_partition (*direction, mechanism, purview, partition, unpartitioned_repertoire=None*)

Return the φ of a mechanism over a purview for the given partition.

Parameters

- **direction** (*Direction*) – *PAST* or *FUTURE*.
- **mechanism** (*tuple[int]*) – The nodes in the mechanism.
- **purview** (*tuple[int]*) – The nodes in the purview.

- **partition** (*Bipartition*) – The partition to evaluate.

Keyword Arguments **unpartitioned_repertoire** (*np.array*) – The unpartitioned repertoire. If not supplied, it will be computed.

Returns The distance between the unpartitioned and partitioned repertoires, and the partitioned repertoire.

Return type *tuple[int, np.ndarray]*

find_mip (*direction, mechanism, purview*)

Return the minimum information partition for a mechanism over a purview.

Parameters

- **direction** (*Direction*) – *PAST* or *FUTURE*.
- **mechanism** (*tuple[int]*) – The nodes in the mechanism.
- **purview** (*tuple[int]*) – The nodes in the purview.

Returns The minimum-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 *PAST*.

mip_future (*mechanism, purview*)

Return the future minimum information partition.

Alias for *find_mip()* with *direction* set to *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.

potential_purviews (*direction, mechanism, purviews=False*)

Return all purviews that could belong to the core cause/effect.

Filters out trivially-reducible purviews.

Parameters

- **direction** (*Direction*) – *PAST* or *FUTURE*.
- **mechanism** (*tuple[int]*) – The mechanism of interest.

Keyword Arguments **purviews** (*tuple[int]*) – Optional subset of purviews of interest.

find_mice (*direction, mechanism, purviews=False*)

Return the maximally irreducible cause or effect for a mechanism.

Parameters

- **direction** (*Direction*) – *:PAST* or *FUTURE*.

- **mechanism** (*tuple[int]*) – The mechanism to be tested for irreducibility.

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

Returns The maximally-irreducible cause or effect in one temporal direction.

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, *PAST* or *FUTURE*, *i.e.*, we return a core cause or core effect, not the pair of them.

core_cause (*mechanism, purviews=False*)

Return the core cause repertoire of a mechanism.

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

core_effect (*mechanism, purviews=False*)

Return the core effect repertoire of a mechanism.

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

phi_max (*mechanism*)

Return the φ^{\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_partitions` (*mechanism, purview*)

Return a generator over all MIP partitions, based on the current configuration.

`pyphi.subsystem.mip_bipartitions` (*mechanism, purview*)

Return an generator of all φ bipartitions of a mechanism over a purview.

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

<pre>A -- -- B</pre>

is not valid, but

<pre>A -- -- B</pre>

is.

Parameters

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

Yields *Bipartition* –

Where each bipartition is:

```
bipart[0].mechanism  bipart[1].mechanism
-----  -----
bipart[0].purview    bipart[1].purview
```

Example

```
>>> mechanism = (0,)
>>> purview = (2, 3)
>>> for partition in mip_bipartitions(mechanism, purview):
...     print(partition, '\n')
0
-- --
2    3
    0
-- --
3    2
    0
-- --
2, 3
```

`pyphi.subsystem.wedge_partitions`(*mechanism, purview*)

Return an iterator over all wedge partitions.

These are partitions which strictly split the mechanism and allow a subset of the purview to be split into a third partition, e.g.:

```
A    B
-- -- --
B    C    D
```

See `PARTITION_TYPE` in *config* for more information.

Parameters

- **mechanism**(*tuple[int]*) – A mechanism.
- **purview**(*tuple[int]*) – A purview.

Yields *Tripartition* – all unique tripartitions of this mechanism and purview.

`pyphi.subsystem.all_partitions`(*mechanism, purview*)

Returns all possible partitions of a mechanism and purview.

Partitions can consist of any number of parts.

Parameters

- **mechanism**(*tuple[int]*) – A mechanism.
- **purview**(*tuple[int]*) – A purview.

Yields *KPartition* – A partition of this mechanism and purview into k parts.

Functions for converting the timescale of a TPM.

`pyphi.timescale.sparse` (*matrix*, *threshold=0.1*)

`pyphi.timescale.sparse_time` (*tpm*, *time_scale*)

`pyphi.timescale.dense_time` (*tpm*, *time_scale*)

`pyphi.timescale.run_tpm` (*tpm*, *time_scale*)

Iterate a TPM by the specified number of time steps.

Parameters

- **tpm** (*np.ndarray*) – A state-by-node tpm.
- **time_scale** (*int*) – The number of steps to run the tpm.

Returns `np.ndarray`

`pyphi.timescale.run_cm` (*cm*, *time_scale*)

Iterate a connectivity matrix the specified number of steps.

Parameters

- **cm** (*np.ndarray*) – A connectivity matrix.
- **time_scale** (*int*) – The number of steps to run.

Returns The connectivity matrix at the new timescale.

Return type `np.ndarray`

Functions for manipulating transition probability matrices.

`pyphi.tpm.tpm_indices` (*tpm*)
Indices of nodes in the TPM.

`pyphi.tpm.is_state_by_state` (*tpm*)
Return True if *tpm* is in state-by-state form, otherwise False.

`pyphi.tpm.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.tpm.expand_tpm` (*tpm*)
Broadcast a state-by-node TPM so that singleton dimensions are expanded over the full network.

`pyphi.tpm.marginalize_out` (*indices*, *tpm*)
Marginalize out a node from a TPM.

Parameters

- **indices** (*list[int]*) – The indices of nodes to be marginalized out.
- **tpm** (*np.ndarray*) – The TPM to marginalize the node out of.

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

Return type *np.ndarray*

`pyphi.tpm.infer_edge` (*tpm*, *a*, *b*, *contexts*)
Infer the presence or absence of an edge from node A to node B.

Let *S* be the set of all nodes in a network. Let $A' = S - \{A\}$. We call the state of A' the context *C* of *A*. There is an edge from *A* to *B* if there exists any context *C*(*A*) such that $p(B \mid C(A), A=0) \neq p(B \mid C(A), A=1)$.

Parameters

- **tpm** (*np.ndarray*) – The TPM in state-by-node, n-dimensional form.
- **a** (*int*) – The index of the putative source node.
- **b** (*int*) – The index of the putative sink node.

Returns True if the edge A->B exists, False otherwise.

Return type bool

`pyphi.tpm.infer_cm(tpm)`

Infer the connectivity matrix associated with a state-by-node TPM in n-dimensional form.

Functions used by more than one PyPhi module or class, or that might be of external use.

`pyphi.utils.state_of(nodes, network_state)`
Return the state-tuple of the given nodes.

`pyphi.utils.all_states(n, holi=False)`
Return all binary states for a system.

Parameters

- **n** (*int*) – The number of elements in the system.
- **holi** (*bool*) – Whether to return the states in HOLI order instead of LOLI order.

Yields *tuple[int]* – The next state of an n-element system, in LOLI order unless `holi` is `True`.

`pyphi.utils.np_immutable(a)`
Make a NumPy array immutable.

`pyphi.utils.np_hash(a)`
Return a hash of a NumPy array.

`pyphi.utils.eq(x, y)`
Compare two values up to 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 An array of combinations.

Return type `np.ndarray`

`pyphi.utils.comb_indices(n, k)`
n-dimensional version of `itertools.combinations`.

Parameters

- **a** (`np.ndarray`) – The array from which to get combinations.
- **k** (`int`) – The desired length of the combinations.

Returns 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, nonempty=False)`
Generate the power set of an iterable.

Parameters **iterable** (`Iterable`) – The iterable from which to generate the power set.

Returns An chained generator over the power set.

Return type generator

Example

```
>>> ps = powerset(np.arange(2))
>>> print(list(ps))
[(), (0,), (1,), (0, 1)]
>>> ps = powerset(np.arange(2), nonempty=True)
>>> print(list(ps))
[(0,), (1,), (0, 1)]
```

`pyphi.utils.load_data(directory, num)`
Load numpy data from the data directory.

The files should be stored in `../data/<dir>` and named `0.npy`, `1.npy`, ... `<num - 1>.npy`.

Returns A list of loaded data, such that `list[i]` contains the contents of `i.npy`.

Return type list

`pyphi.utils.to_string(array)`

Methods for validating arguments.

`pyphi.validate.direction(direction, allow_bi=False)`

Validate that the given direction is one of the allowed constants.

If `allow_bi` is `True` then `Direction.BIDIRECTIONAL` is acceptable.

`pyphi.validate.tpm(tpm, check_independence=True)`

Validate a TPM.

The TPM can be in

- 2-dimensional state-by-state form,
- 2-dimensional state-by-node form, or
- n-dimensional state-by-node form.

`pyphi.validate.conditionally_independent(tpm)`

Validate that the TPM is conditionally independent.

`pyphi.validate.connectivity_matrix(cm)`

Validate the given connectivity matrix.

`pyphi.validate.node_labels(node_labels, node_indices)`

Validate that there is a label for each node.

`pyphi.validate.network(n)`

Validate a *Network*.

Checks the TPM and connectivity matrix.

`pyphi.validate.is_network(network)`

Validate that the argument is a *Network*.

`pyphi.validate.node_states(state)`

Check that the state contains only zeros and ones.

`pyphi.validate.state_length(state, size)`

Check that the state is the given size.

`pyphi.validate.state_reachable` (*subsystem*)
Return whether a state can be reached according to the network's TPM.

`pyphi.validate.cut` (*cut, node_indices*)
Check that the cut is for only the given nodes.

`pyphi.validate.subsystem` (*s*)
Validate a *Subsystem*.
Checks its state and cut.

`pyphi.validate.time_scale` (*time_scale*)
Validate a macro temporal time scale.

`pyphi.validate.partition` (*partition*)
Validate a partition - used by blackboxes and coarse grains.

`pyphi.validate.coarse_grain` (*coarse_grain*)
Validate a macro coarse-graining.

`pyphi.validate.blackbox` (*blackbox*)
Validate a macro blackboxing.

`pyphi.validate.blackbox_and_coarse_grain` (*blackbox, coarse_grain*)
Validate that a coarse-graining properly combines the outputs of a blackboxing.

`pyphi.validate.partition_type` (*value*)
Validate a type of partition.

p

`pyphi.actual`, 67
`pyphi.compute.big_phi`, 73
`pyphi.compute.concept`, 77
`pyphi.compute.distance`, 79
`pyphi.config`, 45
`pyphi.connectivity`, 89
`pyphi.constants`, 91
`pyphi.convert`, 93
`pyphi.distance`, 103
`pyphi.examples`, 107
`pyphi.exceptions`, 115
`pyphi.jsonify`, 117
`pyphi.macro`, 119
`pyphi.models.actual_causation`, 125
`pyphi.models.big_phi`, 129
`pyphi.models.concept`, 131
`pyphi.models.cuts`, 135
`pyphi.network`, 139
`pyphi.node`, 143
`pyphi.partition`, 145
`pyphi.subsystem`, 149
`pyphi.timescale`, 157
`pyphi.tpm`, 159
`pyphi.utils`, 161
`pyphi.validate`, 163

Symbols

- `__bool__()` (pyphi.models.actual_causation.AcBigMip method), 127
 - `__bool__()` (pyphi.models.actual_causation.AcMip method), 126
 - `__bool__()` (pyphi.models.actual_causation.CausalLink method), 126
 - `__bool__()` (pyphi.models.big_phi.BigMip method), 130
 - `__bool__()` (pyphi.models.concept.Concept method), 133
 - `__bool__()` (pyphi.models.concept.Mip method), 132
 - `__bool__()` (pyphi.subsystem.Subsystem method), 150
 - `__enter__()` (pyphi.config.override method), 62
 - `__eq__()` (pyphi.macro.MacroSubsystem method), 120
 - `__eq__()` (pyphi.network.Network method), 140
 - `__eq__()` (pyphi.node.Node method), 143
 - `__eq__()` (pyphi.subsystem.Subsystem method), 150
 - `__exit__()` (pyphi.config.override method), 62
 - `__getattr__()` (pyphi.compute.big_phi.ConceptStyleSystem method), 74
 - `__getnewargs__()` (pyphi.models.cuts.KPartition method), 137
 - `__gt__()` (pyphi.subsystem.Subsystem method), 150
 - `__len__()` (pyphi.subsystem.Subsystem method), 150
 - `__lt__()` (pyphi.subsystem.Subsystem method), 150
 - `__new__()` (pyphi.models.concept.Constellation static method), 134
 - `__new__()` (pyphi.models.cuts.KPartition static method), 137
- ## A
- AcBigMip (class in pyphi.models.actual_causation), 127
 - Account (class in pyphi.models.actual_causation), 126
 - account() (in module pyphi.actual), 70
 - account_distance() (in module pyphi.actual), 70
 - AcMip (class in pyphi.models.actual_causation), 125
 - actual_causation() (in module pyphi.examples), 113
 - actual_cause (pyphi.models.actual_causation.Event attribute), 126
 - actual_effect (pyphi.models.actual_causation.Event attribute), 126
 - ActualCut (class in pyphi.models.cuts), 136
 - after_state (pyphi.actual.Transition attribute), 67
 - after_state (pyphi.models.actual_causation.AcBigMip attribute), 127
 - all() (pyphi.distance.MeasureRegistry method), 103
 - all_blackboxes() (in module pyphi.macro), 123
 - all_coarse_grains() (in module pyphi.macro), 123
 - all_coarse_grains_for_blackbox() (in module pyphi.macro), 123
 - all_complexes() (in module pyphi.compute.big_phi), 74
 - all_groupings() (in module pyphi.macro), 123
 - all_macro_systems() (in module pyphi.macro), 124
 - all_partitions() (in module pyphi.macro), 123
 - all_partitions() (in module pyphi.subsystem), 155
 - all_states() (in module pyphi.utils), 161
 - alpha (pyphi.models.actual_causation.AcBigMip attribute), 127
 - alpha (pyphi.models.actual_causation.AcMip attribute), 125
 - alpha (pyphi.models.actual_causation.CausalLink attribute), 126
 - apply() (pyphi.macro.SystemAttrs method), 119
 - apply_boundary_conditions_to_cm() (in module pyphi.connectivity), 89
 - apply_cut() (pyphi.actual.Transition method), 68
 - apply_cut() (pyphi.compute.big_phi.ConceptStyleSystem method), 74
 - apply_cut() (pyphi.macro.MacroSubsystem method), 120
 - apply_cut() (pyphi.subsystem.Subsystem method), 150
 - asymmetric() (pyphi.distance.MeasureRegistry method), 103
- ## B
- basic_network() (in module pyphi.examples), 107
 - basic_noisy_selfloop_network() (in module pyphi.examples), 108
 - basic_noisy_selfloop_subsystem() (in module pyphi.examples), 108

- basic_subsystem() (in module pyphi.examples), 108
 - before_state (pyphi.actual.Transition attribute), 67
 - before_state (pyphi.models.actual_causation.AcBigMip attribute), 127
 - BIDIRECTIONAL (pyphi.constants.Direction attribute), 91
 - big_acmip() (in module pyphi.actual), 70
 - big_mip() (in module pyphi.compute.big_phi), 74
 - big_mip_bipartitions() (in module pyphi.compute.big_phi), 73
 - big_mip_concept_style() (in module pyphi.compute.big_phi), 75
 - big_phi() (in module pyphi.compute.big_phi), 74
 - big_phi_measure() (in module pyphi.distance), 105
 - BigMip (class in pyphi.models.big_phi), 129
 - BigMipConceptStyle (class in pyphi.compute.big_phi), 75
 - Bipartition (class in pyphi.models.cuts), 137
 - bipartition() (in module pyphi.partition), 145
 - bipartition_indices() (in module pyphi.partition), 145
 - bipartition_of_one() (in module pyphi.partition), 146
 - Blackbox (class in pyphi.macro), 122
 - blackbox (pyphi.macro.MacroNetwork attribute), 123
 - blackbox() (in module pyphi.validate), 164
 - blackbox_and_coarse_grain() (in module pyphi.validate), 164
 - blackbox_network() (in module pyphi.examples), 111
 - bld() (in module pyphi.distance), 105
 - block_cm() (in module pyphi.connectivity), 89
 - block_reducible() (in module pyphi.connectivity), 90
- ## C
- cache_info() (pyphi.subsystem.Subsystem method), 150
 - causal_nexus() (in module pyphi.actual), 70
 - CausalLink (class in pyphi.models.actual_causation), 126
 - causally_significant_nodes (pyphi.network.Network attribute), 140
 - causally_significant_nodes() (in module pyphi.connectivity), 89
 - cause (pyphi.models.concept.Concept attribute), 132
 - cause_effect_info() (pyphi.subsystem.Subsystem method), 152
 - cause_info() (pyphi.subsystem.Subsystem method), 152
 - cause_part1 (pyphi.models.cuts.ActualCut attribute), 136
 - cause_part2 (pyphi.models.cuts.ActualCut attribute), 136
 - cause_purview (pyphi.models.concept.Concept attribute), 133
 - cause_ratio() (pyphi.actual.Transition method), 69
 - cause_repertoire (pyphi.models.concept.Concept attribute), 133
 - cause_repertoire() (pyphi.actual.Transition method), 68
 - cause_repertoire() (pyphi.subsystem.Subsystem method), 151
 - cause_system (pyphi.actual.Transition attribute), 68
 - cause_system (pyphi.compute.big_phi.ConceptStyleSystem attribute), 75
 - clear_caches() (pyphi.subsystem.Subsystem method), 150
 - cm (pyphi.network.Network attribute), 140
 - cm (pyphi.subsystem.Subsystem attribute), 149
 - coarse_grain (pyphi.macro.MacroNetwork attribute), 123
 - coarse_grain() (in module pyphi.macro), 124
 - coarse_grain() (in module pyphi.validate), 164
 - CoarseGrain (class in pyphi.macro), 120
 - comb_indices() (in module pyphi.utils), 161
 - combs() (in module pyphi.utils), 161
 - complexes() (in module pyphi.compute.big_phi), 74
 - compute() (pyphi.actual.FindBigAcMip static method), 70
 - compute() (pyphi.compute.big_phi.FindComplexes static method), 74
 - compute() (pyphi.compute.big_phi.FindMip static method), 73
 - compute() (pyphi.compute.concept.ComputeConstellation static method), 77
 - ComputeConstellation (class in pyphi.compute.concept), 77
 - Concept (class in pyphi.models.concept), 132
 - concept() (in module pyphi.compute.concept), 77
 - concept() (pyphi.compute.big_phi.ConceptStyleSystem method), 75
 - concept() (pyphi.subsystem.Subsystem method), 154
 - concept_cuts() (in module pyphi.compute.big_phi), 75
 - concept_distance() (in module pyphi.compute.distance), 79
 - ConceptStyleSystem (class in pyphi.compute.big_phi), 74
 - conceptual_information() (in module pyphi.compute.concept), 78
 - cond_depend_tpm() (in module pyphi.examples), 109
 - cond_independ_tpm() (in module pyphi.examples), 109
 - condensed() (in module pyphi.compute.big_phi), 74
 - condition_tpm() (in module pyphi.tpm), 159
 - conditionally_independent() (in module pyphi.validate), 163
 - ConditionallyDependentError, 115
 - configure_logging() (in module pyphi.config), 61
 - connectivity_matrix (pyphi.network.Network attribute), 140
 - connectivity_matrix (pyphi.subsystem.Subsystem attribute), 150
 - connectivity_matrix() (in module pyphi.validate), 163
 - Constellation (class in pyphi.models.concept), 133
 - constellation() (in module pyphi.compute.concept), 77
 - constellation_distance() (in module pyphi.compute.distance), 79
 - core_cause() (pyphi.subsystem.Subsystem method), 154
 - core_effect() (pyphi.subsystem.Subsystem method), 154

- Cut (class in `pyphi.models.cuts`), 135
 cut (`pyphi.actual.Transition` attribute), 68
 cut (`pyphi.models.actual_causation.AcBigMip` attribute), 127
 cut (`pyphi.models.big_phi.BigMip` attribute), 129
 cut (`pyphi.subsystem.Subsystem` attribute), 149
 cut() (in module `pyphi.validate`), 164
 cut_indices (`pyphi.macro.MacroSubsystem` attribute), 120
 cut_indices (`pyphi.subsystem.Subsystem` attribute), 150
 cut_matrix (`pyphi.subsystem.Subsystem` attribute), 149
 cut_matrix() (`pyphi.models.cuts.ActualCut` method), 136
 cut_matrix() (`pyphi.models.cuts.Cut` method), 135
 cut_matrix() (`pyphi.models.cuts.KCut` method), 136
 cut_mechanisms (`pyphi.macro.MacroSubsystem` attribute), 120
 cut_mechanisms (`pyphi.subsystem.Subsystem` attribute), 150
 cut_subsystem (`pyphi.models.big_phi.BigMip` attribute), 129
- ## D
- damaged_by_cut() (`pyphi.models.concept.Mice` method), 132
 DATABASE (in module `pyphi.constants`), 91
 default_label() (in module `pyphi.node`), 144
 default_labels() (in module `pyphi.node`), 144
 dense_time() (in module `pyphi.timescale`), 157
 description (`pyphi.actual.FindBigAcMip` attribute), 70
 description (`pyphi.compute.big_phi.FindComplexes` attribute), 74
 description (`pyphi.compute.big_phi.FindMip` attribute), 73
 description (`pyphi.compute.concept.ComputeConstellation` attribute), 77
 directed_account() (in module `pyphi.actual`), 70
 directed_bipartition() (in module `pyphi.partition`), 146
 directed_bipartition_indices() (in module `pyphi.partition`), 146
 directed_bipartition_of_one() (in module `pyphi.partition`), 147
 directed_tripartition() (in module `pyphi.partition`), 147
 directed_tripartition_indices() (in module `pyphi.partition`), 147
 DirectedAccount (class in `pyphi.models.actual_causation`), 127
 Direction (class in `pyphi.constants`), 91
 direction (`pyphi.models.actual_causation.AcMip` attribute), 125
 direction (`pyphi.models.actual_causation.CausalLink` attribute), 126
 direction (`pyphi.models.concept.Mice` attribute), 132
 direction (`pyphi.models.concept.Mip` attribute), 131
 direction() (in module `pyphi.validate`), 163
 directional_big_mip() (in module `pyphi.compute.big_phi`), 75
 directional_emd() (in module `pyphi.distance`), 105
 disjunction_conjunction_network() (in module `pyphi.examples`), 113
 dump() (in module `pyphi.jsonify`), 118
 dumps() (in module `pyphi.jsonify`), 118
- ## E
- effect (`pyphi.models.concept.Concept` attribute), 132
 effect_emd() (in module `pyphi.distance`), 104
 effect_info() (`pyphi.subsystem.Subsystem` method), 152
 effect_part1 (`pyphi.models.cuts.ActualCut` attribute), 136
 effect_part2 (`pyphi.models.cuts.ActualCut` attribute), 136
 effect_purview (`pyphi.models.concept.Concept` attribute), 133
 effect_ratio() (`pyphi.actual.Transition` method), 69
 effect_repertoire (`pyphi.models.concept.Concept` attribute), 133
 effect_repertoire() (`pyphi.actual.Transition` method), 68
 effect_repertoire() (`pyphi.subsystem.Subsystem` method), 151
 effect_system (`pyphi.actual.Transition` attribute), 67
 effect_system (`pyphi.compute.big_phi.ConceptStyleSystem` attribute), 75
 effective_info() (in module `pyphi.macro`), 124
 emd_eq() (`pyphi.models.concept.Concept` method), 133
 emergence (`pyphi.macro.MacroNetwork` attribute), 123, 124
 emergence() (in module `pyphi.macro`), 124
 empty_result() (`pyphi.actual.FindBigAcMip` method), 70
 empty_result() (`pyphi.compute.big_phi.FindComplexes` method), 74
 empty_result() (`pyphi.compute.big_phi.FindMip` method), 73
 empty_result() (`pyphi.compute.concept.ComputeConstellation` method), 77
 encode() (`pyphi.jsonify.PyPhiJSONEncoder` method), 118
 entropy_difference() (in module `pyphi.distance`), 104
 EPSILON (in module `pyphi.constants`), 91
 eq() (in module `pyphi.utils`), 161
 eq_repertoires() (`pyphi.models.concept.Concept` method), 133
 evaluate_cut() (in module `pyphi.compute.big_phi`), 73
 evaluate_partition() (`pyphi.subsystem.Subsystem` method), 152
 Event (class in `pyphi.models.actual_causation`), 126
 events() (in module `pyphi.actual`), 70
 expand_cause_repertoire() (`pyphi.models.concept.Concept` method), 133
 expand_cause_repertoire() (`pyphi.subsystem.Subsystem` method), 152

expand_effect_repertoire()
 (pyphi.models.concept.Concept method),
 133

expand_effect_repertoire() (pyphi.subsystem.Subsystem
 method), 152

expand_node_tpm() (in module pyphi.node), 144

expand_partitioned_cause_repertoire()
 (pyphi.models.concept.Concept method),
 133

expand_partitioned_effect_repertoire()
 (pyphi.models.concept.Concept method),
 133

expand_repertoire() (pyphi.subsystem.Subsystem
 method), 152

expand_tpm() (in module pyphi.tpm), 159

extrinsic_events() (in module pyphi.actual), 71

F

fig10() (in module pyphi.examples), 113

fig14() (in module pyphi.examples), 113

fig16() (in module pyphi.examples), 113

fig1a() (in module pyphi.examples), 112

fig3a() (in module pyphi.examples), 112

fig3b() (in module pyphi.examples), 112

fig4() (in module pyphi.examples), 112

fig5a() (in module pyphi.examples), 112

fig5b() (in module pyphi.examples), 112

fig6() (in module pyphi.examples), 113

fig8() (in module pyphi.examples), 113

fig9() (in module pyphi.examples), 114

FILESYSTEM (in module pyphi.constants), 91

find_actual_cause() (pyphi.actual.Transition method), 69

find_actual_effect() (pyphi.actual.Transition method), 70

find_causal_link() (pyphi.actual.Transition method), 69

find_mice() (pyphi.actual.Transition method), 70

find_mice() (pyphi.subsystem.Subsystem method), 153

find_mip() (pyphi.actual.Transition method), 69

find_mip() (pyphi.subsystem.Subsystem method), 153

FindBigAcMip (class in pyphi.actual), 70

FindComplexes (class in pyphi.compute.big_phi), 74

FindMip (class in pyphi.compute.big_phi), 73

from_json() (in module pyphi.network), 141

from_json() (pyphi.constants.Direction class method), 91

from_json() (pyphi.models.actual_causation.Account
 class method), 127

from_json() (pyphi.models.concept.Concept class
 method), 133

from_json() (pyphi.models.concept.Constellation class
 method), 134

from_json() (pyphi.models.cuts.Bipartition class
 method), 137

from_json() (pyphi.network.Network class method), 140

from_nodes (pyphi.models.cuts.Cut attribute), 135

FUTURE (pyphi.constants.Direction attribute), 91

G

generate_nodes() (in module pyphi.node), 144

get_config_string() (in module pyphi.config), 61

get_inputs_from_cm() (in module pyphi.connectivity), 89

get_outputs_from_cm() (in module pyphi.connectivity),
 89

greater_than_zero() (in module
 pyphi.models.actual_causation), 125

grouping (pyphi.macro.CoarseGrain attribute), 120

H

h2l() (in module pyphi.convert), 97

h2l_sbs() (in module pyphi.convert), 98

h2s() (in module pyphi.convert), 97

hamming_emd() (in module pyphi.distance), 103

hidden_from() (pyphi.macro.Blackbox method), 122

hidden_indices (pyphi.macro.Blackbox attribute), 122

holi2loli() (in module pyphi.convert), 93

holi2loli_state_by_state() (in module pyphi.convert), 95

holi_index2state() (in module pyphi.convert), 94

I

in_same_box() (pyphi.macro.Blackbox method), 122

indices (pyphi.models.cuts.ActualCut attribute), 136

indices (pyphi.models.cuts.Cut attribute), 135

indices (pyphi.models.cuts.KCut attribute), 136

indices2labels() (pyphi.network.Network method), 140

indices2labels() (pyphi.subsystem.Subsystem method),
 151

indices2nodes() (pyphi.subsystem.Subsystem method),
 151

infer_cm() (in module pyphi.tpm), 160

infer_edge() (in module pyphi.tpm), 159

initialize() (in module pyphi.config), 62

inputs (pyphi.node.Node attribute), 143

irreducible_causes (pyphi.models.actual_causation.Account
 attribute), 127

irreducible_effects (pyphi.models.actual_causation.Account
 attribute), 127

irreducible_purviews() (in module pyphi.network), 141

is_cut (pyphi.subsystem.Subsystem attribute), 150

is_full() (in module pyphi.connectivity), 90

is_network() (in module pyphi.validate), 163

is_state_by_state() (in module pyphi.tpm), 159

is_strong() (in module pyphi.connectivity), 90

is_weak() (in module pyphi.connectivity), 90

iterencode() (pyphi.jsonify.PyPhiJSONEncoder method),
 118

J

joblib_memory (in module pyphi.constants), 91

jsonify() (in module pyphi.jsonify), 117

JSONVersionError, 115

K

k_partitions() (in module pyphi.partition), 148
 KCut (class in pyphi.models.cuts), 135
 kld() (in module pyphi.distance), 104
 KPartition (class in pyphi.models.cuts), 136

L

l1() (in module pyphi.distance), 104
 l2h() (in module pyphi.convert), 97
 l2h_sbs() (in module pyphi.convert), 99
 l2s() (in module pyphi.convert), 97
 labeled_mechanisms (pyphi.models.concept.Constellation attribute), 134
 labels2indices() (pyphi.network.Network method), 140
 load() (in module pyphi.jsonify), 118
 load_config_default() (in module pyphi.config), 61
 load_config_dict() (in module pyphi.config), 61
 load_config_file() (in module pyphi.config), 61
 load_data() (in module pyphi.utils), 162
 loads() (in module pyphi.jsonify), 118
 log2() (in module pyphi.actual), 67
 loli2holi() (in module pyphi.convert), 93
 loli2holi_state_by_state() (in module pyphi.convert), 95
 loli_index2state() (in module pyphi.convert), 94

M

macro2blackbox_outputs()
 (pyphi.macro.MacroSubsystem method),
 120
 macro2micro() (pyphi.macro.MacroSubsystem method),
 120
 macro_indices (pyphi.macro.Blackbox attribute), 122
 macro_indices (pyphi.macro.CoarseGrain attribute), 120
 macro_network() (in module pyphi.examples), 111
 macro_state() (pyphi.macro.Blackbox method), 122
 macro_state() (pyphi.macro.CoarseGrain method), 121
 macro_subsystem() (in module pyphi.examples), 111
 macro_tpm() (pyphi.macro.CoarseGrain method), 121
 macro_tpm_sbs() (pyphi.macro.CoarseGrain method),
 121
 MacroNetwork (class in pyphi.macro), 123
 MacroSubsystem (class in pyphi.macro), 119
 main_complex() (in module pyphi.compute.big_phi), 74
 make_mapping() (pyphi.macro.CoarseGrain method),
 121
 marginalize_out() (in module pyphi.tpm), 159
 MeasureRegistry (class in pyphi.distance), 103
 mechanism (pyphi.models.actual_causation.AcMip at-
 tribute), 125
 mechanism (pyphi.models.actual_causation.CausalLink
 attribute), 126
 mechanism (pyphi.models.actual_causation.Event at-
 tribute), 126

mechanism (pyphi.models.concept.Concept attribute),
 132
 mechanism (pyphi.models.concept.Mice attribute), 132
 mechanism (pyphi.models.concept.Mip attribute), 131
 mechanism (pyphi.models.cuts.KPartition attribute), 137
 mechanism (pyphi.models.cuts.Part attribute), 136
 mechanism_indices() (pyphi.actual.Transition method),
 69
 mechanism_state() (pyphi.actual.Transition method), 68
 mechanisms (pyphi.models.concept.Constellation at-
 tribute), 134
 Mice (class in pyphi.models.concept), 132
 micro_indices (pyphi.macro.Blackbox attribute), 122
 micro_indices (pyphi.macro.CoarseGrain attribute), 120
 micro_network (pyphi.macro.MacroNetwork attribute),
 123
 micro_phi (pyphi.macro.MacroNetwork attribute), 123
 Mip (class in pyphi.models.concept), 131
 mip (pyphi.models.actual_causation.CausalLink at-
 tribute), 126
 mip (pyphi.models.concept.Mice attribute), 132
 mip_bipartitions() (in module pyphi.subsystem), 154
 mip_future() (pyphi.subsystem.Subsystem method), 153
 mip_partitions() (in module pyphi.subsystem), 154
 mip_past() (pyphi.subsystem.Subsystem method), 153
 mp2q() (in module pyphi.distance), 104

N

Network (class in pyphi.network), 139
 network (pyphi.actual.Transition attribute), 67
 network (pyphi.compute.big_phi.BigMipConceptStyle
 attribute), 75
 network (pyphi.macro.MacroNetwork attribute), 123
 network (pyphi.models.big_phi.BigMip attribute), 129
 network (pyphi.subsystem.Subsystem attribute), 149
 network() (in module pyphi.validate), 163
 nexus() (in module pyphi.actual), 70
 nice_true_constellation() (in module pyphi.actual), 70
 Node (class in pyphi.node), 143
 node_indices (pyphi.actual.Transition attribute), 67
 node_indices (pyphi.network.Network attribute), 140
 node_indices (pyphi.subsystem.Subsystem attribute), 149
 node_labels (pyphi.network.Network attribute), 140
 node_labels() (in module pyphi.macro), 119
 node_labels() (in module pyphi.validate), 163
 node_states() (in module pyphi.validate), 163
 nodes (pyphi.macro.SystemAttrs attribute), 119
 nodes (pyphi.subsystem.Subsystem attribute), 150
 nodes2indices() (in module pyphi.convert), 93
 nodes2state() (in module pyphi.convert), 93
 normalize_constellation() (in
 pyphi.models.concept), 134
 np_hash() (in module pyphi.utils), 161
 np_immutable() (in module pyphi.utils), 161

np_suppress (class in pyphi.distance), 103
 null_concept (pyphi.subsystem.Subsystem attribute), 154
 null_cut (pyphi.subsystem.Subsystem attribute), 149
 num_states (pyphi.network.Network attribute), 140

O

OFF (in module pyphi.constants), 91
 order_by() (pyphi.compute.big_phi.BigMipConceptStyle method), 75
 order_by() (pyphi.models.actual_causation.AcBigMip method), 127
 order_by() (pyphi.models.actual_causation.AcMip method), 126
 order_by() (pyphi.models.actual_causation.CausalLink method), 126
 order_by() (pyphi.models.big_phi.BigMip method), 130
 order_by() (pyphi.models.concept.Concept method), 133
 order_by() (pyphi.models.concept.Mice method), 132
 order_by() (pyphi.models.concept.Mip method), 131
 output_indices (pyphi.macro.Blackbox attribute), 122
 outputs (pyphi.node.Node attribute), 143
 outputs_of() (pyphi.macro.Blackbox method), 122
 override (class in pyphi.config), 61

P

pack() (pyphi.macro.SystemAttrs static method), 119
 parse_node_indices() (pyphi.network.Network method), 140
 Part (class in pyphi.models.cuts), 136
 part0 (pyphi.models.cuts.Bipartition attribute), 137
 part1 (pyphi.models.cuts.Bipartition attribute), 137
 partition (pyphi.macro.Blackbox attribute), 122
 partition (pyphi.macro.CoarseGrain attribute), 120
 partition (pyphi.models.actual_causation.AcMip attribute), 125
 partition (pyphi.models.concept.Mice attribute), 132
 partition (pyphi.models.concept.Mip attribute), 131
 partition() (in module pyphi.validate), 164
 partition_type() (in module pyphi.validate), 164
 partitioned_account (pyphi.models.actual_causation.AcBigMip attribute), 127
 partitioned_constellation (pyphi.models.big_phi.BigMip attribute), 129
 partitioned_probability (pyphi.models.actual_causation.AcMip attribute), 125
 partitioned_probability() (pyphi.actual.Transition method), 69
 partitioned_repertoire (pyphi.models.concept.Mice attribute), 132
 partitioned_repertoire (pyphi.models.concept.Mip attribute), 131
 partitioned_repertoire() (pyphi.actual.Transition method), 69

partitioned_repertoire() (pyphi.subsystem.Subsystem method), 152
 partitions() (in module pyphi.partition), 145
 PAST (pyphi.constants.Direction attribute), 91
 phi (pyphi.compute.big_phi.BigMipConceptStyle attribute), 75
 phi (pyphi.macro.MacroNetwork attribute), 123
 phi (pyphi.models.actual_causation.AcMip attribute), 126
 phi (pyphi.models.actual_causation.CausalLink attribute), 126
 phi (pyphi.models.big_phi.BigMip attribute), 129
 phi (pyphi.models.concept.Concept attribute), 133
 phi (pyphi.models.concept.Mice attribute), 132
 phi (pyphi.models.concept.Mip attribute), 131
 phi() (pyphi.subsystem.Subsystem method), 153
 phi_by_grain() (in module pyphi.macro), 124
 phi_max() (pyphi.subsystem.Subsystem method), 154
 phi_mip_future() (pyphi.subsystem.Subsystem method), 153
 phi_mip_past() (pyphi.subsystem.Subsystem method), 153
 phis (pyphi.models.concept.Constellation attribute), 134
 PICKLE_PROTOCOL (in module pyphi.constants), 91
 possible_complexes() (in module pyphi.compute.big_phi), 74
 potential_purviews() (pyphi.actual.Transition method), 69
 potential_purviews() (pyphi.macro.MacroSubsystem method), 120
 potential_purviews() (pyphi.network.Network method), 140
 potential_purviews() (pyphi.subsystem.Subsystem method), 153
 powerset() (in module pyphi.utils), 162
 print() (pyphi.models.big_phi.BigMip method), 129
 print_config() (in module pyphi.config), 61
 probability (pyphi.models.actual_causation.AcMip attribute), 125
 probability() (pyphi.actual.Transition method), 68
 process_result() (pyphi.actual.FindBigAcMip method), 70
 process_result() (pyphi.compute.big_phi.FindComplexes method), 74
 process_result() (pyphi.compute.big_phi.FindMip method), 73
 process_result() (pyphi.compute.concept.ComputeConstellation method), 77
 propagation_delay_network() (in module pyphi.examples), 110
 proper_state (pyphi.subsystem.Subsystem attribute), 150
 psq2() (in module pyphi.distance), 104
 purview (pyphi.models.actual_causation.AcMip attribute), 125

purview (pyphi.models.actual_causation.CausalLink attribute), 126
 purview (pyphi.models.concept.Mice attribute), 132
 purview (pyphi.models.concept.Mip attribute), 131
 purview (pyphi.models.cuts.KPartition attribute), 137
 purview (pyphi.models.cuts.Part attribute), 136
 purview_state() (pyphi.actual.Transition method), 68
 pyphi.actual (module), 67
 pyphi.compute.big_phi (module), 73
 pyphi.compute.concept (module), 77
 pyphi.compute.distance (module), 79
 pyphi.config (module), 45
 pyphi.connectivity (module), 89
 pyphi.constants (module), 91
 pyphi.convert (module), 93
 pyphi.distance (module), 103
 pyphi.examples (module), 107
 pyphi.exceptions (module), 115
 pyphi.jsonify (module), 117
 pyphi.macro (module), 119
 pyphi.models.actual_causation (module), 125
 pyphi.models.big_phi (module), 129
 pyphi.models.concept (module), 131
 pyphi.models.cuts (module), 135
 pyphi.network (module), 139
 pyphi.node (module), 143
 pyphi.partition (module), 145
 pyphi.subsystem (module), 149
 pyphi.timescale (module), 157
 pyphi.tpm (module), 159
 pyphi.utils (module), 161
 pyphi.validate (module), 163
 PyPhiJSONDecoder (class in pyphi.jsonify), 118
 PyPhiJSONEncoder (class in pyphi.jsonify), 117

R

rebuild_system_tpm() (in module pyphi.macro), 119
 register() (pyphi.distance.MeasureRegistry method), 103
 reindex() (in module pyphi.macro), 119
 reindex() (pyphi.macro.Blackbox method), 122
 reindex() (pyphi.macro.CoarseGrain method), 120
 relevant_connections() (in module pyphi.connectivity), 89
 remove_singleton_dimensions() (in module pyphi.macro), 119
 repertoire (pyphi.models.concept.Mice attribute), 132
 repertoire() (pyphi.actual.Transition method), 68
 repertoire() (pyphi.subsystem.Subsystem method), 151
 residue_network() (in module pyphi.examples), 108
 residue_subsystem() (in module pyphi.examples), 109
 reverse_bits() (in module pyphi.convert), 93
 reverse_elements() (in module pyphi.partition), 146
 rule110_network() (in module pyphi.examples), 112
 rule154_network() (in module pyphi.examples), 112

run_cm() (in module pyphi.timescale), 157
 run_tpm() (in module pyphi.macro), 119
 run_tpm() (in module pyphi.timescale), 157

S

s2h() (in module pyphi.convert), 98
 s2l() (in module pyphi.convert), 98
 sbn2sbs() (in module pyphi.convert), 99
 sbs2sbn() (in module pyphi.convert), 100
 size (pyphi.network.Network attribute), 140
 size (pyphi.subsystem.Subsystem attribute), 150
 small_phi_constellation_distance() (in module pyphi.compute.distance), 79
 small_phi_measure() (in module pyphi.distance), 105
 small_phi_time (pyphi.models.big_phi.BigMip attribute), 129
 sparse() (in module pyphi.timescale), 157
 sparse_time() (in module pyphi.timescale), 157
 state (pyphi.models.actual_causation.AcMip attribute), 125
 state (pyphi.subsystem.Subsystem attribute), 149
 state2holi_index() (in module pyphi.convert), 93
 state2loli_index() (in module pyphi.convert), 94
 state_by_node2state_by_state() (in module pyphi.convert), 96
 state_by_state2state_by_node() (in module pyphi.convert), 96
 state_length() (in module pyphi.validate), 163
 state_of() (in module pyphi.utils), 161
 state_probability() (pyphi.actual.Transition method), 68
 state_reachable() (in module pyphi.validate), 164
 StateUnreachableError, 115
 Subsystem (class in pyphi.subsystem), 149
 subsystem (pyphi.models.big_phi.BigMip attribute), 129
 subsystem (pyphi.models.concept.Concept attribute), 133
 subsystem (pyphi.models.concept.Mip attribute), 131
 subsystem() (in module pyphi.validate), 164
 subsystems() (in module pyphi.compute.big_phi), 74
 system (pyphi.actual.Transition attribute), 68
 SystemAttrs (class in pyphi.macro), 119

T

time (pyphi.models.big_phi.BigMip attribute), 129
 time (pyphi.models.concept.Concept attribute), 133
 time_scale (pyphi.macro.MacroNetwork attribute), 123
 time_scale() (in module pyphi.validate), 164
 to_2_d() (in module pyphi.convert), 99
 to_2_dimensional() (in module pyphi.convert), 95
 to_json() (pyphi.actual.Transition method), 68
 to_json() (pyphi.constants.Direction method), 91
 to_json() (pyphi.models.actual_causation.AcBigMip method), 127
 to_json() (pyphi.models.actual_causation.Account method), 127

to_json() (pyphi.models.actual_causation.AcMip method), 126
 to_json() (pyphi.models.actual_causation.CausalLink method), 126
 to_json() (pyphi.models.big_phi.BigMip method), 130
 to_json() (pyphi.models.concept.Concept method), 133
 to_json() (pyphi.models.concept.Constellation method), 134
 to_json() (pyphi.models.concept.Mice method), 132
 to_json() (pyphi.models.concept.Mip method), 132
 to_json() (pyphi.models.cuts.ActualCut method), 136
 to_json() (pyphi.models.cuts.Bipartition method), 137
 to_json() (pyphi.models.cuts.Cut method), 135
 to_json() (pyphi.models.cuts.KPartition method), 137
 to_json() (pyphi.models.cuts.Part method), 136
 to_json() (pyphi.network.Network method), 140
 to_json() (pyphi.node.Node method), 144
 to_json() (pyphi.subsystem.Subsystem method), 150
 to_n_d() (in module pyphi.convert), 99
 to_n_dimensional() (in module pyphi.convert), 95
 to_nodes (pyphi.models.cuts.Cut attribute), 135
 to_string() (in module pyphi.utils), 162
 tpm (pyphi.network.Network attribute), 140
 tpm (pyphi.node.Node attribute), 143
 tpm (pyphi.subsystem.Subsystem attribute), 149
 tpm() (in module pyphi.validate), 163
 tpm_indices() (in module pyphi.tpm), 159
 tpm_off (pyphi.node.Node attribute), 143
 tpm_on (pyphi.node.Node attribute), 143
 tpm_size (pyphi.subsystem.Subsystem attribute), 150
 Transition (class in pyphi.actual), 67
 transition (pyphi.models.actual_causation.AcBigMip attribute), 127
 transitions() (in module pyphi.actual), 70
 Tripartition (class in pyphi.models.cuts), 137
 true_constellation() (in module pyphi.actual), 71
 true_events() (in module pyphi.actual), 71

U

unconstrained_cause_repertoire() (pyphi.actual.Transition method), 68
 unconstrained_cause_repertoire() (pyphi.subsystem.Subsystem method), 152
 unconstrained_effect_repertoire() (pyphi.actual.Transition method), 68
 unconstrained_effect_repertoire() (pyphi.subsystem.Subsystem method), 152
 unconstrained_probability() (pyphi.actual.Transition method), 68
 unconstrained_repertoire() (pyphi.subsystem.Subsystem method), 152
 unordered_unless_eq (pyphi.compute.big_phi.BigMipConceptStyle attribute), 75

unordered_unless_eq (pyphi.models.actual_causation.AcBigMip attribute), 127
 unordered_unless_eq (pyphi.models.actual_causation.AcMip attribute), 126
 unordered_unless_eq (pyphi.models.actual_causation.CausalLink attribute), 126
 unordered_unless_eq (pyphi.models.big_phi.BigMip attribute), 130
 unordered_unless_eq (pyphi.models.concept.Concept attribute), 133
 unordered_unless_eq (pyphi.models.concept.Mice attribute), 132
 unordered_unless_eq (pyphi.models.concept.Mip attribute), 131
 unpartitioned_account (pyphi.models.actual_causation.AcBigMip attribute), 127
 unpartitioned_constellation (pyphi.models.big_phi.BigMip attribute), 129
 unpartitioned_repertoire (pyphi.models.concept.Mip attribute), 131

W

wedge_partitions() (in module pyphi.subsystem), 155

X

xor_network() (in module pyphi.examples), 109
 xor_subsystem() (in module pyphi.examples), 109