

aigverse

aigverse
A Python library for working with logic networks, synthesis, and optimization

Version 0.1.2.dev7

Marcel Walter
Technical University of Munich

Jun 13, 2026

Abstract

`aigverse` is an open-source C++17 and Python infrastructure library for working with logic networks, synthesis, and optimization in Python-first workflows.

It builds directly upon the [EPFL Logic Synthesis Libraries](#), particularly [mockturtle](#), [kitty](#), and [lorina](#), and exposes these mature C++ capabilities through an idiomatic Python interface. This allows users to run synthesis workflows from Python without reimplementing core algorithms there. The result is reusable support for And-Inverter Graph (AIG) construction and manipulation, optimization and equivalence checking, dataset generation, and export into graph and numeric representations for downstream data science and ML pipelines.

Key features include:

- Efficient logic representation using And-Inverter Graphs (AIGs)
- Support for various file formats (AIGER, Verilog, Bench, PLA)
- High-performance C++ backend with a Pythonic interface
- Optional adapters for graph and numeric interoperability in machine learning and data science workflows
- Comprehensive tools for logic synthesis and optimization

`aigverse` is designed as reusable bridge infrastructure rather than as a full end-to-end EDA toolchain.

This documentation provides a comprehensive guide to the `aigverse` library, including [installation instructions](#), a [quickstart guide](#), and detailed [API documentation](#). The source code of `aigverse` is publicly available on GitHub at [marcelwa/aigverse](#), while pre-built binaries are available via [PyPI](#) for all major operating systems and all modern Python versions.

➔ See also

For a deeper dive into the vision and technical details behind `aigverse`, see the presentation “**aigverse: Toward machine learning-driven logic synthesis**” from the [Free Silicon Conference \(FSiC\) 2025](#). The slides are available on the [FSiC wiki](#) and cover the motivation, architecture, and future directions of the `aigverse` project.

Note

A live version of this document is available at aigverse.readthedocs.io.

Contents

I	Installation	2
I-A	Core Library	3
I-B	Machine Learning Adapters	3
II	And-Inverter Graphs (AIGs)	4
II-A	Quickstart	4
II-B	Basic AIG Concepts	4
II-C	AIG Views	6
II-D	File I/O	9
II-E	Index Lists	10
II-F	<code>pickle</code> Support	11
III	Truth Tables	12
III-A	Creating Truth Tables	12
III-B	Basic Manipulation	12
III-C	Truth Table Properties	13
III-D	Truth Table Simulation	13
III-E	<code>pickle</code> Support	13
IV	Algorithms	14
IV-A	Simulation	14
IV-B	Optimization	15
IV-C	Equivalence Checking	17
V	Generators	18
V-A	Random AIG Generation	18
V-B	Python-Side Batching	18
V-C	Structured Benchmark Builders	19
V-D	Control Generators	19
VI	Machine Learning Integration	19
VI-A	Adapters	19
VI-B	Dataset Generation	19
VI-C	Truth Tables	22
VII	<code>aigverse</code>	23
VII-A	Submodules	23
VII-B	Package Contents	43
	Index	44

I Installation

`aigverse` wraps mature C/C++ synthesis backends from the [EPFL Logic Synthesis Libraries](#) with an idiomatic Python interface for Python-first workflows. Optional adapters extend this core with graph and array interoperability for downstream ML and data science pipelines. The resulting Python package is available on PyPI and can be installed on all major operating systems and all active Python versions.

💡 Tip

We highly recommend using `uv` for working with Python projects. It is an extremely fast Python package and project manager, written in Rust and developed by [Astral](#) (the same team behind `ruff`). It can act as a drop-in replacement for `pip` and `virtualenv`, and provides a more modern and faster alternative to the traditional Python package management tools. It automatically handles the creation of virtual environments and the installation of packages, and is much faster than `pip`. Additionally, it can even set up Python for you if it is not installed yet.

If you do not have `uv` installed yet, you can install it via:

macOS and Linux

```
$ curl -LsSf https://astral.sh/uv/install.sh | sh
```

Windows

```
$ powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Check out their excellent [documentation](#) for more information.

I-A Core Library

To install the core `aigverse` library, you can use `uv` or `pip`.

`uv` (recommended)

```
$ uv pip install aigverse
```

`pip`

```
(.venv) $ python -m pip install aigverse
```

In most practical cases (under 64-bit Linux, macOS incl. Apple Silicon, and Windows), this requires no compilation and merely downloads and installs a platform-specific pre-built wheel.

Once installed, you can check if the installation was successful by running:

```
(.venv) $ python -c "import aigverse; print(aigverse.__version__)"
```

which should print the installed version of the library.

If you want to use the `aigverse` Python package in your own project, you can simply add it as a dependency to your `pyproject.toml` or `setup.py` file. This will automatically install the `aigverse` package and its dependencies when your project is installed.

`uv` (recommended)

```
$ uv add aigverse
```

`pyproject.toml`

```
[project]
# ...
dependencies = ["aigverse"]
# ...
```

`setup.py`

```
from setuptools import setup

setup(
    # ...
    install_requires=["aigverse"],
    # ...
)
```

I-B Machine Learning Adapters

The base installation intentionally excludes ML and data science adapters so that core synthesis workflows remain lightweight and free of heavy optional dependencies. Install the `aigverse[adapters]` extra when you need graph or numeric interoperability in Python ML/data science pipelines:

uv (recommended)

```
$ uv pip install "aigverse[adapters]"
```

pip

```
(.venv) $ python -m pip install "aigverse[adapters]"
```

The same syntax applies to adding the `aigverse` package with adapters as a dependency to your own project.

II And-Inverter Graphs (AIGs)

The central data structure for working with And-Inverter Graphs (AIGs) in `aigverse` is the `Aig` class. It enables efficient representation and manipulation of logic circuits in a form that's well-suited for optimization and verification tasks.

The following will demonstrate how to work with the `Aig` class in Python.

Note

AIGs (And-Inverter Graphs) are a compact representation of Boolean functions using only AND gates and inverters (NOT gates). They are widely used in formal verification, hardware design, and optimization tasks.

II-A Quickstart

The following code snippet demonstrates how to create a simple AIG representing basic logic operations.

```
1 from aigverse.networks import Aig
2
3 # Create a new AIG network
4 aig = Aig()
5
6 # Create primary inputs
7 x1 = aig.create_pi()
8 x2 = aig.create_pi()
9
10 # Create logic gates
11 f_and = aig.create_and(x1, x2) # AND gate
12 f_or = aig.create_or(x1, x2) # OR gate
13
14 # Create primary outputs
15 aig.create_po(f_and)
16 aig.create_po(f_or)
17
18 # Print the size of the AIG network
19 print(f"AIG Size: {aig.size}")
```

```
AIG Size: 5
```

Note

All primary inputs (PIs) must be created before any logic gates.

II-B Basic AIG Concepts

An AIG consists of nodes and signals:

- **Nodes** represent either primary inputs, constants, or logic gates
- **Signals** reference nodes, possibly with complementation (inversion)

```

1 # Create a new AIG
2 aig = Aig()
3
4 # Create a constant (false)
5 const0 = aig.get_constant(False)
6 print(f"Constant 0: {const0}")
7
8 # Create primary inputs
9 a = aig.create_pi()
10 b = aig.create_pi()
11 print(f"Input a: {a}")
12 print(f"Input b: {b}")
13
14 # Create an AND gate and its complement (NAND)
15 and_gate = aig.create_and(a, b)
16 nand_gate = aig.create_not(and_gate)
17 print(f"AND gate: {and_gate}")
18 print(f"NAND gate: {nand_gate}")
19
20 # Get the node from a signal
21 node = aig.get_node(and_gate)
22 print(f"Node of AND gate: {node}")
23
24 # Check if a signal is complemented
25 is_complemented = aig.is_complemented(nand_gate)
26 print(f"Is NAND complemented? {is_complemented}")

```

```

Constant 0: Signal(0)
Input a: Signal(1)
Input b: Signal(2)
AND gate: Signal(3)
NAND gate: Signal(!3)
Node of AND gate: 3
Is NAND complemented? True

```

Exploring AIG Structure

You can iterate over all nodes in the AIG, or specific subsets like primary inputs or logic gates.

```

1 # Create a sample AIG
2 aig = Aig()
3 a = aig.create_pi()
4 b = aig.create_pi()
5 c = aig.create_pi()
6 f1 = aig.create_and(a, b)
7 f2 = aig.create_or(f1, c)
8 aig.create_po(f2)
9
10 # Iterate over all nodes in the AIG
11 print("All nodes:")
12 for node in aig.nodes():
13     print(f" Node: {node}")
14
15 # Iterate only over primary inputs
16 print("\nPrimary inputs:")
17 for pi in aig.pis():
18     print(f" PI: {pi}")
19
20 # Iterate only over logic gates
21 print("\nLogic gates:")
22 for gate in aig.gates():
23     print(f" Gate: {gate}")
24
25 # Iterate over the fanins of a node
26 print("\nFanins of the OR gate:")
27 or_node = aig.get_node(f2)
28 for fanin in aig.fanins(or_node):
29     print(f" Fanin: {fanin}")

```

```

All nodes:
Node: 0
Node: 1
Node: 2
Node: 3
Node: 4
Node: 5

Primary inputs:
PI: 1
PI: 2
PI: 3

Logic gates:
Gate: 4
Gate: 5

Fanins of the OR gate:
Fanin: Signal(!3)
Fanin: Signal(!4)

```

Building Complex Functions

AIGs support a variety of logic functions beyond just AND gates. Internally, those are decomposed into multiple nodes.

```

1  aig = Aig()
2  a = aig.create_pi()
3  b = aig.create_pi()
4  c = aig.create_pi()
5
6  # Create various logic functions
7  and_gate = aig.create_and(a, b)
8  or_gate = aig.create_or(a, b)
9  xor_gate = aig.create_xor(a, b)
10 maj_gate = aig.create_maj(a, b, c) # Majority function (a&b | a&c | b&c)
11 ite_gate = aig.create_ite(a, b, c) # If-then-else: a ? b : c
12
13 # Add outputs
14 aig.create_po(and_gate)
15 aig.create_po(or_gate)
16 aig.create_po(xor_gate)
17 aig.create_po(maj_gate)
18 aig.create_po(ite_gate)
19
20 # Print statistics
21 print(f"AIG Size: {aig.size}")
22 print(f"Number of gates: {aig.num_gates}")
23 print(f"Number of primary inputs: {aig.num_pis}")
24 print(f"Number of primary outputs: {aig.num_pos}")

```

```

AIG Size: 13
Number of gates: 9
Number of primary inputs: 3
Number of primary outputs: 5

```

II-C AIG Views

AIG views provide alternative representations of AIGs for specific tasks, such as depth computation or fanout analysis. These views can be layered on top of the original AIG, allowing you to work with the same underlying structure while adding additional functionality.

Network and Signal Names

The *NamedAig* class extends the standard AIG with the ability to assign names to the network itself, primary inputs, primary outputs, and internal signals. This is particularly useful for debugging, visualization, and interfacing with external tools that rely on human-readable signal names.

```

1  from aigverse.networks import NamedAig
2
3  # Create a new named AIG (or construct from existing AIG)
4  named_aig = NamedAig()
5
6  # Set the network name
7  named_aig.set_network_name("full_adder")
8
9  # Create named primary inputs
10 a = named_aig.create_pi("a")
11 b = named_aig.create_pi("b")
12 cin = named_aig.create_pi("cin")
13
14 # Create full adder logic
15 sum = named_aig.create_xor3(a, b, cin)
16 carry = named_aig.create_maj(a, b, cin)
17
18 # Assign names to internal signals
19 named_aig.set_name(sum, "sum")
20 named_aig.set_name(carry, "carry")
21
22 # Create named primary outputs
23 named_aig.create_po(sum, "sum_output")
24 named_aig.create_po(carry, "carry_output")
25
26 # Retrieve names
27 print(f"Network name: {named_aig.get_network_name()}")
28 print(f"PI names: {[named_aig.get_name(pi) for pi in named_aig.pis()]}")
29 print(f"Signal name (sum): {named_aig.get_name(sum)}")
30 print(f"Signal name (carry): {named_aig.get_name(carry)}")
31 print(f"PO name at index 0: {named_aig.get_output_name(0)}")
32 print(f"PO name at index 1: {named_aig.get_output_name(1)}")
33
34 # Check if a signal has a name
35 print(f"Has name: {named_aig.has_name(sum)}")

```

```
Network name: full_adder
```

```

-----
TypeError                                Traceback (most recent call last)
Cell In[6], line 28
    24 named_aig.create_po(carry, "carry_output")
    25
    26 # Retrieve names
    27 print(f"Network name: {named_aig.get_network_name()}")
--> 28 print(f"PI names: {[named_aig.get_name(pi) for pi in named_aig.pis()]}")
    29 print(f"Signal name (sum): {named_aig.get_name(sum)}")
    30 print(f"Signal name (carry): {named_aig.get_name(carry)}")
    31 print(f"PO name at index 0: {named_aig.get_output_name(0)}")

TypeError: get_name(): incompatible function arguments. The following argument types are
supported:
  1. get_name(self, s: aigverse.networks.AigSignal) -> str

Invoked with types: aigverse.networks.NamedAig, int

```

Named AIGs are automatically created when reading from file formats that contain naming information, e.g., Verilog and AIGER. See *File I/O* for more details.

Note

Names are tied to the specific AIG structure. When you apply optimization algorithms or structural modifications (such as `cleanup_dangling()`), the names will be lost as the return type will be downcast to `Aig`. If preserving names is important, consider reapplying them after optimization.

Depth and Level Computation

The depth of an AIG network represents the longest path from any input to any output, which corresponds to the critical path delay in a circuit. You can compute the depth and level of each node using the `DepthAig` class.

```
1 from aigverse.networks import DepthAig
2
3 # Create a sample AIG
4 aig = Aig()
5 a = aig.create_pi()
6 b = aig.create_pi()
7 c = aig.create_pi()
8 f1 = aig.create_and(a, b)
9 f2 = aig.create_and(f1, c)
10 aig.create_po(f2)
11
12 # Create a depth view of the AIG
13 depth_aig = DepthAig(aig)
14
15 # Get the depth of the AIG
16 print(f"Depth of AIG: {depth_aig.num_levels}")
17
18 # Print the level of each node
19 print("\nLevel of each node:")
20 for node in aig.nodes():
21     print(f" Node {node}: Level {depth_aig.level(node)}")
22
23 # Check which nodes are on the critical path
24 print("\nNodes on critical path:")
25 for node in aig.nodes():
26     if depth_aig.is_on_critical_path(node):
27         print(f" Node {node}")
```

AIGs with Fanout Information

Fanouts of AIG nodes can be collected using `FanoutAig`.

```
1 from aigverse.networks import FanoutAig
2
3 # Create a sample AIG
4 aig = Aig()
5 x1 = aig.create_pi()
6 x2 = aig.create_pi()
7 x3 = aig.create_pi()
8
9 # Create AND gates
10 n4 = aig.create_and(x1, x2)
11 n5 = aig.create_and(n4, x3)
12 n6 = aig.create_and(n4, n5)
13 # Create primary outputs
14 aig.create_po(n6)
15
16 fanout_aig = FanoutAig(aig)
17 print("\nFanout nodes of n4:")
18 for node in fanout_aig.fanouts(aig.get_node(n4)):
19     print(f" Node {node}")
```

Sequential AIGs

SequentialAigs extend standard AIGs to include registers, which allow modeling sequential circuits with memory elements.

```
1 from aigverse.networks import SequentialAig
2
3 # Create a sequential AIG
4 seq_aig = SequentialAig()
5
6 # Create a primary input and a register output
7 x = seq_aig.create_pi()      # Regular PI
8 r = seq_aig.create_ro()     # Register output (sequential PI)
9
10 # Create logic
11 f_and = seq_aig.create_and(x, r)  # AND gate
12
13 # Create a primary output and a register input
14 seq_aig.create_po(f_and)      # Regular PO
15 seq_aig.create_ri(f_and)     # Register input (sequential PO)
16
17 # Print information about the sequential AIG
18 print(f"Number of PIs: {seq_aig.num_pis}")
19 print(f"Number of POs: {seq_aig.num_pos}")
20 print(f"Number of registers: {seq_aig.num_registers}")
21
22 # Get the register associations (RI-R0 pairs)
23 print("\nRegister associations:")
24 registers = seq_aig.registers()
25 for reg in registers:
26     ri, ro = reg
27     print(f" RI: {ri} -> R0: {ro}")
```

Note

When creating sequential AIGs, follow these rules:

1. All register outputs (ROs) must be created after all primary inputs (PIs).
2. All register inputs (RIs) must be created after all primary outputs (POs).
3. As with regular AIGs, all PIs and ROs must be created before any logic gates.

II-D File I/O

AIGs can be read from and written to various file formats.

```

1 from aigverse.io import write_aiger, write_verilog, write_dot, read_aiger_into_aig, read_
  ↳verilog_into_aig, read_pla_into_aig
2
3 # Create a sample AIG
4 aig = Aig()
5 a = aig.create_pi()
6 b = aig.create_pi()
7 f = aig.create_and(a, b)
8 aig.create_po(f)
9
10 # Write to AIGER format
11 write_aiger(aig, "example.aig")
12
13 # Write to Verilog format
14 write_verilog(aig, "example.v")
15
16 # Write to DOT format
17 write_dot(aig, "example.dot")
18
19 # Read from AIGER format
20 read_aig = read_aiger_into_aig("example.aig")
21
22 # Read from Verilog format
23 read_verilog_aig = read_verilog_into_aig("example.v")
24
25 # Read from PLA format
26 read_pla_aig = read_pla_into_aig("example.pla")
27
28 print(f"Original AIG size: {aig.size}")
29 print(f"Read AIGER AIG size: {read_aig.size}")
30 print(f"Read Verilog AIG size: {read_verilog_aig.size}")
31 print(f"Read PLA AIG size: {read_pla_aig.size}")

```

Note

The gate-level Verilog file support constitutes a very small subset of the Verilog standard, similar in extent to what ABC supports. For more information, see the [lorina parser](#) used by this project.

II-E Index Lists

Alternatively, index lists provide a compact, serialization-friendly representation of an AIG's structure as a flat list of integers. This is useful for ML pipelines, dataset generation, or exporting AIGs for use in environments where fixed-size numeric arrays are required.

```

1 from aigverse.networks import AigIndexList
2
3 # Create a sample AIG
4 aig = Aig()
5 a = aig.create_pi()
6 b = aig.create_pi()
7 c = aig.create_pi()
8 d = aig.create_pi()
9 t0 = aig.create_and(a, b)
10 t1 = aig.create_and(~c, ~d)
11 t2 = aig.create_xor(t0, t1)
12 aig.create_po(t2)
13
14 # Convert an AIG to an index list
15 indices = aig.to_index_list()
16
17 # Convert an index list back to an AIG
18 aig2 = indices.to_aig()
19
20 # Convert to a Python list
21 indices = [int(i) for i in indices]
22 print(indices)

```

The first three entries encode number of PIs, number of POs, and number of gates, respectively. In the example above, those are 4, 1, and 5. Successive pairs of indices refer to the fanins signals of nodes. Each fanin exists in two polarities: negated = odd index, and non-negated = even index. In the example, 2 and 4 refer to the non-negated signals originating from PIs 1 and 2. These form the first AND gate. The subsequent 7 and 9 are odd, hence, negated. If the first index is lower than the second, an AND gate is encoded. Otherwise, it is an XOR gate. The final indices of the list refer to the PO signals. It must be ensured that they match the encoded number of POs.

For more information on the index list format, see [mockturtle's documentation](#).

II-F pickle Support

AIGs support Python's `pickle` protocol, allowing you to serialize and deserialize AIG objects for persistent storage. This is useful for saving intermediate results, sharing AIGs between processes, quickly restoring previously computed networks, or integrating with Python-first data science and machine learning workflows.

```

1 import pickle
2
3 # Save AIG to a file using pickle
4 with open("aig.pkl", "wb") as f:
5     pickle.dump(aig, f)
6
7 # Load the AIG from the pickle file
8 with open("aig.pkl", "rb") as f:
9     unpickled_aig = pickle.load(f)

```

You can also pickle and unpickle multiple AIGs at once by storing them in a tuple or list.

```

1 aig1 = Aig()
2 a1 = aig1.create_pi()
3 b1 = aig1.create_pi()
4 f1 = aig1.create_and(a1, b1)
5 aig1.create_po(f1)
6
7 aig2 = Aig()
8 a2 = aig2.create_pi()
9 b2 = aig2.create_pi()
10 f2 = aig2.create_or(a2, b2)
11 aig2.create_po(f2)
12
13 # Pickle both AIGs together
14 with open("aigs.pkl", "wb") as f:
15     pickle.dump((aig1, aig2), f)
16
17 # Unpickle them
18 with open("aigs.pkl", "rb") as f:
19     unpickled_aig1, unpickled_aig2 = pickle.load(f)

```

III Truth Tables

The aigverse library provides support for working with truth tables, which are a fundamental representation of Boolean functions. The TruthTable class offers efficient manipulation and analysis of Boolean functions.

Note

Truth tables provide a complete specification of a Boolean function by listing all possible input combinations and their corresponding outputs. They are particularly useful for small functions where exhaustive enumeration is feasible.

III-A Creating Truth Tables

Truth tables can be created in several ways:

```

1 from aigverse.utils import TruthTable
2
3 # Initialize a truth table with 3 variables (2^3 = 8 entries)
4 tt = TruthTable(3)
5
6 # Create a truth table from a hex string (representing the Majority function)
7 tt.create_from_hex_string("e8")
8 print(f"Truth table from hex string: {tt.to_binary()}")
9
10 # Create a truth table for an AND function
11 tt_and = TruthTable(2)
12 tt_and.create_from_hex_string("8")
13 print(f"AND function: {tt_and.to_binary()}")
14
15 # Create a truth table for an OR function
16 tt_or = TruthTable(2)
17 tt_or.create_from_hex_string("e")
18 print(f"OR function: {tt_or.to_binary()}")

```

```

Truth table from hex string: 11101000
AND function: 1000
OR function: 1110

```

III-B Basic Manipulation

Truth tables provide various methods for bit manipulation:

```

1 # Create a truth table
2 tt = TruthTable(3)
3 tt.create_from_hex_string("e8") # Majority function
4
5 # Get individual bits
6 print(f"Original truth table: {tt.to_binary()}")
7 print(f"Bit at position 0: {int(tt.get_bit(0))}")
8 print(f"Bit at position 7: {int(tt.get_bit(7))}")
9
10 # Flip bits
11 tt.flip_bit(0)
12 tt.flip_bit(7)
13 print(f"After flipping bits 0 and 7: {tt.to_binary()}")
14
15 # Clear the truth table
16 tt.clear()
17 print(f"After clearing: {tt.to_binary()}")
18
19 # Check if constant
20 print(f"Is constant 0? {tt.is_const0()}")

```

```

Original truth table: 11101000
Bit at position 0: 0
Bit at position 7: 1
After flipping bits 0 and 7: 01101001
After clearing: 00000000
Is constant 0? True

```

III-C Truth Table Properties

You can analyze various properties of truth tables:

```

1 # Create a truth table for XOR
2 tt_xor = TruthTable(2)
3 tt_xor.create_from_hex_string("6") # XOR function
4 print(f"XOR function: {tt_xor.to_binary()}")
5
6 # Get number of variables and bits
7 print(f"Number of variables: {tt_xor.num_vars()}")
8 print(f"Number of bits: {tt_xor.num_bits()}")
9
10 # Check if the function is balanced (equal number of 0s and 1s)
11 num_ones = sum(int(tt_xor.get_bit(i)) for i in range(tt_xor.num_bits()))
12 is_balanced = num_ones == tt_xor.num_bits() // 2
13 print(f"Is balanced? {is_balanced}")

```

```

XOR function: 0110
Number of variables: 2
Number of bits: 4
Is balanced? True

```

III-D Truth Table Simulation

The simulation of AIGs and other logic networks using truth tables is covered in the *Simulation section* of the Algorithms documentation. This approach allows you to obtain the truth tables for outputs and internal nodes of a logic network.

III-E pickle Support

Truth tables support Python's pickle protocol, allowing you to serialize and deserialize them for persistent storage or use in data science workflows.

```

1 import pickle
2
3 # Create a truth table
4 tt = TruthTable(3)
5 tt.create_from_hex_string("d8") # ITE function
6
7 # Pickle the truth table
8 with open("tt.pkl", "wb") as f:
9     pickle.dump(tt, f)
10
11 # Unpickle the truth table
12 with open("tt.pkl", "rb") as f:
13     unpickled_tt = pickle.load(f)
14
15 # Verify that the unpickled object is identical
16 print(f"Original:      {tt.to_binary()}")
17 print(f"Unpickled:    {unpickled_tt.to_binary()}")
18 print(f"Equivalent:   {tt == unpickled_tt}")

```

```

Original:      11011000
Unpickled:    11011000
Equivalent:   True

```

You can also pickle multiple truth tables at once by storing them in a list or tuple.

IV Algorithms

This section covers the various algorithms available in `aigverse` for working with And-Inverter Graphs (AIGs) and other logic representations. These algorithms enable simulation, optimization, and verification of logic networks.

IV-A Simulation

Simulation algorithms allow you to evaluate the outputs of a logic network for all possible input combinations, effectively generating truth tables for the network's outputs and internal nodes.

Functional Simulation

For simulating AIGs with truth tables, the `simulate()` and `{py:func} ~aigverse.algorithms.simulate_nodes` functions allow you to obtain truth tables for outputs and internal nodes of an AIG.

```

1 from aigverse.networks import Aig
2 from aigverse.algorithms import simulate, simulate_nodes
3
4 # Create a sample AIG
5 aig = Aig()
6 a = aig.create_pi()
7 b = aig.create_pi()
8 f_and = aig.create_and(a, b)
9 f_or = aig.create_or(a, b)
10 aig.create_po(f_and)
11 aig.create_po(f_or)
12
13 # Simulate the outputs
14 output_tts = simulate(aig)
15
16 # Print the truth tables
17 print("Truth tables of outputs:")
18 for i, tt in enumerate(output_tts):
19     print(f"  Output {i}: {tt.to_binary()}")
20
21 # Simulate all nodes
22 node_tts = simulate_nodes(aig)
23
24 # Print the truth table of each node
25 print("\nTruth tables of nodes:")
26 for node, tt in node_tts.items():
27     print(f"  Node {node}: {tt.to_binary()}")

```

Truth tables of outputs:

Output 0: 1000
Output 1: 1110

Truth tables of nodes:

Node 4: 0001
Node 3: 1000
Node 2: 1100
Node 1: 1010
Node 0: 0000

IV-B Optimization

AIG optimization aims to reduce the number of AND gates and inverters in a circuit while maintaining its logical functionality. Different optimization techniques target various aspects of the AIG structure.

Basic Optimization Workflow

The typical optimization workflow involves:

1. Creating or loading an AIG
2. Applying one or more optimization algorithms
3. Verifying correctness through equivalence checking

```
1 from aigverse.io import read_aiger_into_aig
2
3 # Load the i10 benchmark circuit - a real-world example
4 aig = read_aiger_into_aig("examples/i10.aig")
5
6 # Print statistics about the loaded circuit
7 print(f"i10 benchmark:")
8 print(f"  I/O: {aig.num_pis}/{aig.num_pos}")
9 print(f"  AND gates: {aig.num_gates}")
```

```
i10 benchmark:
I/O: 257/224
AND gates: 2675
```

Resubstitution

Resubstitution identifies portions of logic that can be expressed using existing signals in the network. This technique is particularly effective at identifying and eliminating redundant logic.

```
1 from aigverse.algorithms import aig_resubstitution
2
3 # Clone the AIG for comparison
4 aig_resub = aig.clone()
5
6 # Apply resubstitution
7 aig_resub = aig_resubstitution(aig_resub, window_size=12)
8
9 print(f"Original AND gates: {aig.num_gates}")
10 print(f"After resubstitution: {aig_resub.num_gates} AND gates")
11 print(f"Reduction: {aig.num_gates - aig_resub.num_gates} gates ({(aig.num_gates - aig_resub.
↵ num_gates) / aig.num_gates * 100:.2f}%)")
```

```
Original AND gates: 2675
After resubstitution: 2173 AND gates
Reduction: 502 gates (18.77%)
```

Sum-of-Products Refactoring

SOP (Sum of Products) refactoring collapses parts of the AIG into truth tables, then re-synthesizes those portions using Sum-of-Products representations. This can find more efficient implementations for complex logic functions.

```

1 from aigverse.algorithms import sop_refactoring
2
3 # Clone the AIG for comparison
4 aig_refactor = aig.clone()
5
6 # Apply SOP refactoring
7 aig_refactor = sop_refactoring(aig_refactor, use_reconvergence_cut=True)
8
9 print(f"Original AND gates: {aig.num_gates}")
10 print(f"After SOP refactoring: {aig_refactor.num_gates} AND gates")
11 print(f"Reduction: {aig.num_gates - aig_refactor.num_gates} gates ({(aig.num_gates - aig_
↳ refactor.num_gates) / aig.num_gates * 100:.2f}%)")

```

```

Original AND gates: 2675
After SOP refactoring: 2239 AND gates
Reduction: 436 gates (16.30%)

```

Cut Rewriting

Cut rewriting identifies small subgraphs (cuts) in the AIG and replaces them with pre-computed optimal implementations from a library. This technique leverages NPN-equivalence classes to find the best possible implementation for each cut.

```

1 from aigverse.algorithms import aig_cut_rewriting
2
3 # Clone the AIG for comparison
4 aig_rewrite = aig.clone()
5
6 # Apply cut rewriting
7 aig_rewrite = aig_cut_rewriting(aig_rewrite, cut_size=4)
8
9 print(f"Original AND gates: {aig.num_gates}")
10 print(f"After cut rewriting: {aig_rewrite.num_gates} AND gates")
11 print(f"Reduction: {aig.num_gates - aig_rewrite.num_gates} gates ({(aig.num_gates - aig_
↳ rewrite.num_gates) / aig.num_gates * 100:.2f}%)")

```

```

Original AND gates: 2675
After cut rewriting: 2200 AND gates
Reduction: 475 gates (17.76%)

```

Balancing

Balancing performs (E)SOP factoring to minimize the number of levels in the AIG.

```

1 from aigverse.algorithms import balancing
2 from aigverse.networks import DepthAig
3
4 # Clone the AIG for comparison
5 aig_balance = aig.clone()
6
7 # Apply balancing
8 aig_balance = balancing(aig_balance, rebalance_function="sop")
9
10 # Compute depth
11 original_depth = DepthAig(aig).num_levels
12 balanced_depth = DepthAig(aig_balance).num_levels
13
14 print(f"Original depth: {original_depth} levels")
15 print(f"After balancing: {balanced_depth} levels")
16 print(f"Reduction in depth: {original_depth - balanced_depth} levels ({(original_depth -
↳ balanced_depth) / original_depth * 100:.2f}%)")

```

```

Original depth: 50 levels
After balancing: 32 levels
Reduction in depth: 18 levels (36.00%)

```

Combining Optimization Techniques

For best results, optimization techniques are typically applied in combination, often in multiple passes. The order of application can significantly impact the final result.

```
1 # Apply optimization techniques in sequence
2 aig_opt = aig.clone()
3
4 # First pass
5 aig_opt = aig_resubstitution(aig_opt)
6 aig_opt = sop_refactoring(aig_opt)
7 aig_opt = aig_cut_rewriting(aig_opt)
8
9 # Second pass
10 aig_opt = aig_resubstitution(aig_opt)
11 aig_opt = sop_refactoring(aig_opt)
12
13 print(f"\nTotal optimization results:")
14 print(f"- Original: {aig.num_gates} AND gates")
15 print(f"- Optimized: {aig_opt.num_gates} AND gates")
16 print(f"- Total reduction: {aig.num_gates - aig_opt.num_gates} gates ({(aig.num_gates - aig_
  ↳opt.num_gates) / aig.num_gates * 100:.2f}%)")
```

```
Total optimization results:
- Original: 2675 AND gates
- Optimized: 1950 AND gates
- Total reduction: 725 gates (27.10%)
```

Some algorithms offer the `inplace=True` keyword argument for performance-sensitive pipelines of chained optimization. Calling functions such as `aig_resubstitution()` and `sop_refactoring()` with `inplace=True` mutates the passed network and returns `None`:

```
1 from aigverse.algorithms import cleanup_dangling
2
3 aig_fast = aig.clone()
4 aig_resubstitution(aig_fast, inplace=True)
5 sop_refactoring(aig_fast, inplace=True)
6
7 # Explicit cleanup step after in-place chaining
8 aig_fast = cleanup_dangling(aig_fast)
```

i Note

When choosing this route, users are responsible to call `cleanup_dangling()` to obtain a structurally valid AIG.

IV-C Equivalence Checking

Equivalence checking algorithms verify that two logic networks implement the same function, which is especially important after performing optimizations.

```
1 from aigverse.algorithms import equivalence_checking
2
3 # Verify that our optimized circuit from the previous section maintains functional equivalence
4 are_equivalent = equivalence_checking(aig, aig_opt)
5 print(f"\nOriginal and optimized benchmark circuits are equivalent: {are_equivalent}")
6 print(f"This confirms our optimization preserved the circuit's functionality while reducing")
7 print(f"the gate count from {aig.num_gates} to {aig_opt.num_gates} AND gates.")
```

```
Original and optimized benchmark circuits are equivalent: True
This confirms our optimization preserved the circuit's functionality while reducing
the gate count from 2086 to 1950 AND gates.
```

V Generators

The module provides generation helpers for reproducible random dataset creation and structured arithmetic/control benchmarks.

V-A Random AIG Generation

Use `random_aig()` for reproducible one-shot generation.

```
1 from aigverse.generators import random_aig
2
3 # One-shot random AIG with fixed size
4 single = random_aig(num_pis=4, num_gates=12, seed=42)
5 print(single.num_pis, single.num_gates)
```

```
4 12
```

V-B Python-Side Batching

Batching is intentionally kept on the Python side to keep experiment orchestration reproducible and easy to compose with downstream data science and ML code. For fixed-size datasets, call `random_aig()` repeatedly with fixed `num_pis` and `num_gates`.

Note

`random_aig()` does not guarantee a fixed `num_pos` across seeds: mockturtle's random topology can leave a different number of dangling nodes depending on the sampled structure. As a result, repeated calls with identical `num_pis/num_gates` may still produce different output counts if the seed varies. For ML training loops, when you need a uniform label shape, you can filter generated examples by `num_pos` or resample seeds until `num_pos` matches your target.

```
1 from aigverse.generators import random_aig
2
3 batch_fixed = [random_aig(num_pis=4, num_gates=12, seed=1000 + i) for i in range(8)]
4 print(len(batch_fixed), batch_fixed[0].num_pis, batch_fixed[0].num_gates)
```

```
8 4 12
```

For size-diverse datasets, sample sizes in Python and pass them to `random_aig()`.

```
1 import random
2
3 from aigverse.generators import random_aig
4
5 rng = random.Random(7)
6
7 dataset = [
8     random_aig(
9         num_pis=rng.randint(3, 5),
10        num_gates=rng.randint(10, 16),
11        seed=2000 + i,
12    )
13    for i in range(8)
14 ]
15
16 print([(aig.num_pis, aig.num_gates) for aig in dataset])
```

```
[(4, 11), (4, 15), (3, 10), (5, 10), (4, 14), (3, 14), (3, 10), (3, 13)]
```

V-C Structured Benchmark Builders

Complete benchmark networks of arithmetic and control circuits of arbitrary bitwidth can be created via respective high-level generator functions.

```
1 from aigverse.generators import ripple_carry_adder, carry_lookahead_adder
2
3 rca4 = ripple_carry_adder(bitwidth=4)
4 print(f"RCA: I/O: {rca4.num_pis}/{rca4.num_pos}, gates: {rca4.num_gates}")
5 cla4 = carry_lookahead_adder(bitwidth=4)
6 print(f"CLA: I/O: {cla4.num_pis}/{cla4.num_pos}, gates: {cla4.num_gates}")
```

```
RCA: I/O: 8/5, gates: 24
CLA: I/O: 8/5, gates: 36
```

V-D Control Generators

Control builders follow the same high-level model.

```
1 from aigverse.generators import binary_decoder, multiplexer
2
3 mux = multiplexer(bitwidth=4)
4 print(f"MUX: I/O: {mux.num_pis}/{mux.num_pos}, gates: {mux.num_gates}")
5
6 decoder = binary_decoder(num_select_bits=4)
7 print(f"Decoder: I/O: {decoder.num_pis}/{decoder.num_pos}, gates: {decoder.num_gates}")
```

```
MUX: I/O: 9/4, gates: 12
Decoder: I/O: 4/16, gates: 24
```

Available structured builders include `ripple_carry_adder()`, `carry_lookahead_adder()`, `sideways_sum_adder()`, `ripple_carry_multiplier()`, `multiplexer()`, and `binary_decoder()`.

VI Machine Learning Integration

Many ML and data science workflows are Python-first, while mature logic synthesis data structures and algorithms are predominantly implemented in C/C++ toolchains. Without reusable infrastructure, projects often reimplement synthesis functionality in Python or rely on brittle wrappers and file-conversion scripts around external tools. `aigverse` addresses this gap by exposing native AIG construction, manipulation, optimization, and analysis through an idiomatic Python API, and by providing optional adapters for graph and numeric representations. This enables reproducible dataset generation, feature extraction, and downstream model experimentation from within standard Python workflows.

VI-A Adapters

Adapters are the interoperability layer for ML and data science workflows. To keep the base library lightweight, they are optional and installed via the `adapters` extra only when needed. See the [Installation](#) documentation for more details.

VI-B Dataset Generation

The generators module offers a reproducible way to create synthetic AIG datasets before converting them into ML-ready formats.

```

1 import random
2
3 from aigverse.generators import random_aig
4
5 rng = random.Random(1234)
6
7 # Generate a reproducible, size-diverse AIG dataset
8 dataset = [
9     random_aig(
10         num_pis=rng.randint(4, 6),
11         num_gates=rng.randint(20, 40),
12         seed=5000 + i,
13     )
14     for i in range(16)
15 ]
16
17 print(len(dataset), dataset[0].num_pis, dataset[0].num_gates)

```

```
16 5 23
```

NetworkX

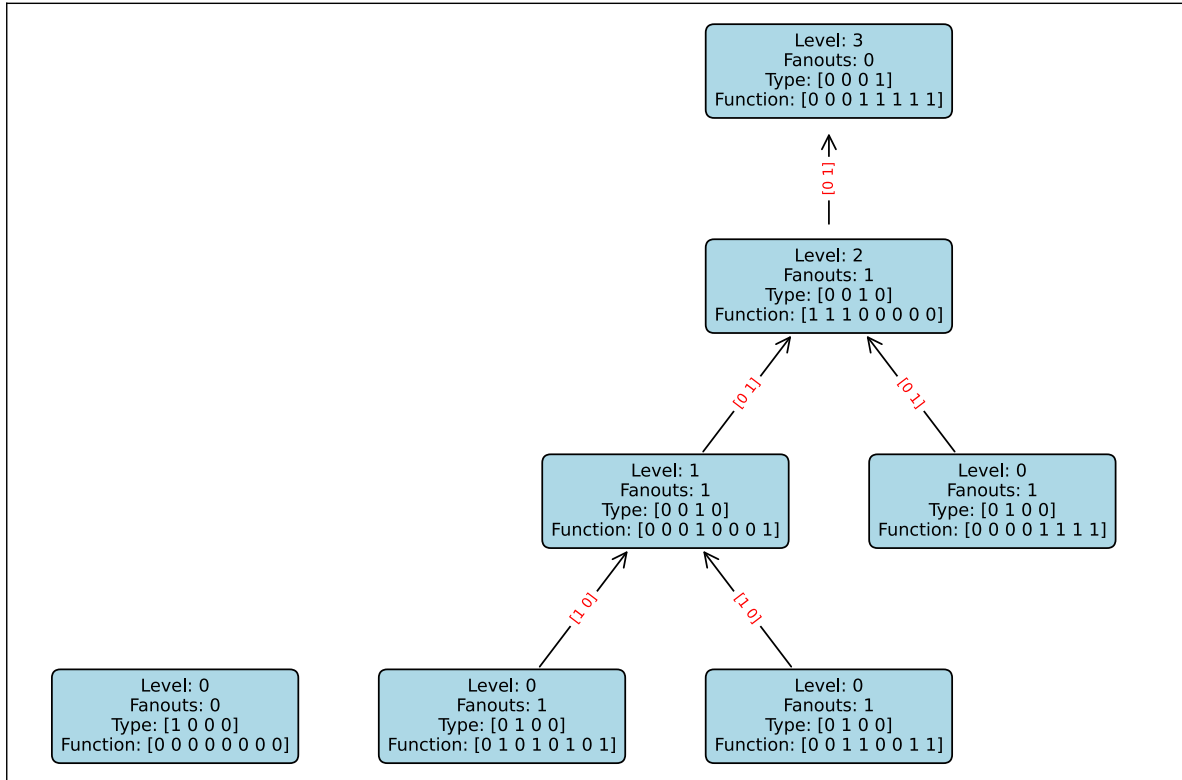
The [NetworkX](#) adapter allows you to convert an AIG into a [DiGraph](#) object. This enables use of graph-analysis and graph-learning tooling that operates on NetworkX graphs. Once converted, you can extract node and edge features, visualize the structure (e.g., with [Matplotlib](#)), or feed the graph into downstream model pipelines.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3 from networkx.drawing.nx_agraph import graphviz_layout
4 import numpy as np
5
6 from aigverse.networks import Aig
7 import aigverse.adapters
8
9 # Create a sample AIG
10 aig = Aig()
11 a = aig.create_pi()
12 b = aig.create_pi()
13 c = aig.create_pi()
14 f1 = aig.create_and(a, b)
15 f2 = aig.create_or(f1, c)
16 aig.create_po(f2)
17
18 # Convert the AIG to a NetworkX graph
19 G = aig.to_networkx(levels=True, fanouts=True, node_tts=True, dtype=np.int32)
20
21 # Generate the initial layout using Graphviz's 'dot' program
22 pos = graphviz_layout(G, prog="dot")
23
24 # Invert the y-axis to flip the layout upside down
25 # This places the primary inputs (level 0) at the bottom
26 for node, position in pos.items():
27     pos[node] = (position[0], -position[1])
28
29 # Prepare the labels for nodes and edges from graph attributes
30 node_labels = {
31     node: f"Level: {data['level']}\nFanouts: {data['fanouts']}\nType: {data['type']}\nFunction: {data['function']}"
32     for node, data in G.nodes(data=True)
33 }
34 edge_labels = {(u, v): data["type"] for u, v, data in G.edges(data=True)}
35
36 # Plot the graph
37 plt.figure(figsize=(12, 8))
38 plt.title("AIG with Attribute Labels")
39
40 # Draw the graph structure (just the edges and arrows)
41 nx.draw_networkx_nodes(G, pos, node_size=0)
42
43 # Draw the node labels with a bounding box
44 nx.draw_networkx_labels(
45     G,
46     pos,
47     labels=node_labels,
48     font_size=10,
49     bbox={"facecolor": "lightblue", "edgecolor": "black", "boxstyle": "round,pad=0.5"},
50 )
51
52 # Draw the network edges
53 nx.draw_networkx_edges(
54     G,
55     pos,
56     node_size=5000,
57     arrows=True,
58     arrowstyle="->",
59     arrowsize=20,
60 )
61
62 # Draw the edge labels to show edge attributes
63 nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color="red", font_size=8)
64
65 # Pad and show the plot
66 plt.margins(x=0.2)
67 plt.show()

```

AIG with Attribute Labels



VI-C Truth Tables

Truth tables can be easily converted to Python lists or NumPy arrays, making them compatible with standard ML libraries such as [scikit-learn](#), [PyTorch](#), or [TensorFlow](#). Since `TruthTable` objects are iterable, this conversion is direct and intuitive. You can use these arrays as labels or features in supervised learning tasks, or as part of a dataset for training and evaluating models.

```

1 from aigverse.utils import TruthTable
2 import numpy as np
3
4 # Create a simple truth table, e.g., a 3-input majority function
5 tt = TruthTable(3)
6 tt.create_from_hex_string("e8")
7
8 # Export to a list
9 tt_list = list(tt)
10 print(f"As list: {tt_list}")
11
12 # Export to NumPy arrays of different types
13 tt_np_bool = np.array(tt)
14 print(f"As NumPy bool array: {tt_np_bool}")
15 tt_np_int = np.array(tt, dtype=np.int32)
16 print(f"As NumPy int array: {tt_np_int}")
17 tt_np_float = np.array(tt, dtype=np.float64)
18 print(f"As NumPy float array: {tt_np_float}")
19
20
21 # These arrays can now be used as labels for an ML model.
22 # For example, let's generate the corresponding feature matrix:
23 def generate_inputs(num_vars):
24     inputs = []
25     for i in range(2**num_vars):
26         # Convert i to binary and pad with zeros
27         binary = bin(i)[2:].zfill(num_vars)
28         inputs.append([int(bit) for bit in binary])
29     return np.array(inputs)
30
31
32 feature_matrix = generate_inputs(tt.num_vars())
33 labels = tt_np_int # Using the integer array as labels
34
35 print("\nFeature matrix (X) and labels (y) for ML:")
36 print("X:\n", feature_matrix)
37 print("y:\n", labels)

```

```

As list: [False, False, False, True, False, True, True, True]
As NumPy bool array: [False False False True False True True True]
As NumPy int array: [0 0 0 1 0 1 1 1]
As NumPy float array: [0. 0. 0. 1. 0. 1. 1. 1.]

Feature matrix (X) and labels (y) for ML:
X:
[[0 0 0]
 [0 0 1]
 [0 1 0]
 [0 1 1]
 [1 0 0]
 [1 0 1]
 [1 1 0]
 [1 1 1]]
y:
[0 0 0 1 0 1 1 1]

```

VII aigverse

aigverse: A Python Library for Logic Networks, Synthesis, and Optimization.

VII-A Submodules

aigverse.adapters

aigverse adapters for ML tasks.

Submodules

aigverse.adapters.networkx AIG to NetworkX adapter.

Module Contents

to_networkx(self: Aig, * (Keyword-only parameters separator (PEP 3102)), levels: bool = False, fanouts: bool = False, node_tts: bool = False, graph_tts: bool = False, dtype: type[generic] = np.int8) → DiGraph

Converts an Aig to a DiGraph.

This function transforms the AIG into a directed graph representation using the NetworkX library. It allows for the inclusion of various attributes for the graph, its nodes, and edges, making it suitable for graph-based machine learning tasks.

Note that the constant-0 node is always included in the graph, as index 0, even if it is not referenced by any edges.

Parameters

- **self** – The AIG object to convert.
- **levels** – If True, computes and adds level information for each node and the total number of levels to the graph, as attributes `level` and `levels`, respectively. Defaults to False.
- **fanouts** – If True, adds the fanout count for each node as an integer `fanouts` attribute (0 for synthetic PO nodes). Defaults to False.
- **node_tts** – If True, computes and adds a truth table for each node as a `function` attribute. Defaults to False.
- **graph_tts** – If True, computes and adds the graph’s overall truth table as a `function` attribute to the graph. Defaults to False.
- **dtype** – The data type for truth tables and all one-hot encodings. Defaults to `int8`. For machine learning tasks, a floating-point type such as `float32` or `float64` may be more appropriate, as it allows for gradient-based optimization.

Returns A DiGraph representing the AIG.

Graph Attributes:

- `type` (str): "AIG".
- `num_pis` (int): Number of primary inputs.
- `num_pos` (int): Number of primary outputs.
- `num_gates` (int): Number of AND gates.
- `levels` (int, optional): Total number of levels in the AIG.
- `function` (list[ndarray], optional): Graph’s truth tables.
- `name` (str, optional): Network name (only for NamedAig).

Node Attributes:

- `index` (int): The node’s identifier.
- `level` (int, optional): The level of the node in the AIG.
- `fanouts` (int, optional): Fanouts of the node.
- `function` (ndarray, optional): The node’s truth table.
- `type` (ndarray, optional): The node type (0 for AND, 1 for OR, 2 for NOT, 3 for XOR, 4 for MUX, 5 for MUX2, 6 for MUX3, 7 for MUX4, 8 for MUX5, 9 for MUX6, 10 for MUX7, 11 for MUX8, 12 for MUX9, 13 for MUX10, 14 for MUX11, 15 for MUX12, 16 for MUX13, 17 for MUX14, 18 for MUX15, 19 for MUX16, 20 for MUX17, 21 for MUX18, 22 for MUX19, 23 for MUX20, 24 for MUX21, 25 for MUX22, 26 for MUX23, 27 for MUX24, 28 for MUX25, 29 for MUX26, 30 for MUX27, 31 for MUX28, 32 for MUX29, 33 for MUX30, 34 for MUX31, 35 for MUX32, 36 for MUX33, 37 for MUX34, 38 for MUX35, 39 for MUX36, 40 for MUX37, 41 for MUX38, 42 for MUX39, 43 for MUX40, 44 for MUX41, 45 for MUX42, 46 for MUX43, 47 for MUX44, 48 for MUX45, 49 for MUX46, 50 for MUX47, 51 for MUX48, 52 for MUX49, 53 for MUX50, 54 for MUX51, 55 for MUX52, 56 for MUX53, 57 for MUX54, 58 for MUX55, 59 for MUX56, 60 for MUX57, 61 for MUX58, 62 for MUX59, 63 for MUX60, 64 for MUX61, 65 for MUX62, 66 for MUX63, 67 for MUX64, 68 for MUX65, 69 for MUX66, 70 for MUX67, 71 for MUX68, 72 for MUX69, 73 for MUX70, 74 for MUX71, 75 for MUX72, 76 for MUX73, 77 for MUX74, 78 for MUX75, 79 for MUX76, 80 for MUX77, 81 for MUX78, 82 for MUX79, 83 for MUX80, 84 for MUX81, 85 for MUX82, 86 for MUX83, 87 for MUX84, 88 for MUX85, 89 for MUX86, 90 for MUX87, 91 for MUX88, 92 for MUX89, 93 for MUX90, 94 for MUX91, 95 for MUX92, 96 for MUX93, 97 for MUX94, 98 for MUX95, 99 for MUX96, 100 for MUX97, 101 for MUX98, 102 for MUX99, 103 for MUX100, 104 for MUX101, 105 for MUX102, 106 for MUX103, 107 for MUX104, 108 for MUX105, 109 for MUX106, 110 for MUX107, 111 for MUX108, 112 for MUX109, 113 for MUX110, 114 for MUX111, 115 for MUX112, 116 for MUX113, 117 for MUX114, 118 for MUX115, 119 for MUX116, 120 for MUX117, 121 for MUX118, 122 for MUX119, 123 for MUX120, 124 for MUX121, 125 for MUX122, 126 for MUX123, 127 for MUX124, 128 for MUX125, 129 for MUX126, 130 for MUX127, 131 for MUX128, 132 for MUX129, 133 for MUX130, 134 for MUX131, 135 for MUX132, 136 for MUX133, 137 for MUX134, 138 for MUX135, 139 for MUX136, 140 for MUX137, 141 for MUX138, 142 for MUX139, 143 for MUX140, 144 for MUX141, 145 for MUX142, 146 for MUX143, 147 for MUX144, 148 for MUX145, 149 for MUX146, 150 for MUX147, 151 for MUX148, 152 for MUX149, 153 for MUX150, 154 for MUX151, 155 for MUX152, 156 for MUX153, 157 for MUX154, 158 for MUX155, 159 for MUX156, 160 for MUX157, 161 for MUX158, 162 for MUX159, 163 for MUX160, 164 for MUX161, 165 for MUX162, 166 for MUX163, 167 for MUX164, 168 for MUX165, 169 for MUX166, 170 for MUX167, 171 for MUX168, 172 for MUX169, 173 for MUX170, 174 for MUX171, 175 for MUX172, 176 for MUX173, 177 for MUX174, 178 for MUX175, 179 for MUX176, 180 for MUX177, 181 for MUX178, 182 for MUX179, 183 for MUX180, 184 for MUX181, 185 for MUX182, 186 for MUX183, 187 for MUX184, 188 for MUX185, 189 for MUX186, 190 for MUX187, 191 for MUX188, 192 for MUX189, 193 for MUX190, 194 for MUX191, 195 for MUX192, 196 for MUX193, 197 for MUX194, 198 for MUX195, 199 for MUX196, 200 for MUX197, 201 for MUX198, 202 for MUX199, 203 for MUX200, 204 for MUX201, 205 for MUX202, 206 for MUX203, 207 for MUX204, 208 for MUX205, 209 for MUX206, 210 for MUX207, 211 for MUX208, 212 for MUX209, 213 for MUX210, 214 for MUX211, 215 for MUX212, 216 for MUX213, 217 for MUX214, 218 for MUX215, 219 for MUX216, 220 for MUX217, 221 for MUX218, 222 for MUX219, 223 for MUX220, 224 for MUX221, 225 for MUX222, 226 for MUX223, 227 for MUX224, 228 for MUX225, 229 for MUX226, 230 for MUX227, 231 for MUX228, 232 for MUX229, 233 for MUX230, 234 for MUX231, 235 for MUX232, 236 for MUX233, 237 for MUX234, 238 for MUX235, 239 for MUX236, 240 for MUX237, 241 for MUX238, 242 for MUX239, 243 for MUX240, 244 for MUX241, 245 for MUX242, 246 for MUX243, 247 for MUX244, 248 for MUX245, 249 for MUX246, 250 for MUX247, 251 for MUX248, 252 for MUX249, 253 for MUX250, 254 for MUX251, 255 for MUX252, 256 for MUX253, 257 for MUX254, 258 for MUX255, 259 for MUX256, 260 for MUX257, 261 for MUX258, 262 for MUX259, 263 for MUX260, 264 for MUX261, 265 for MUX262, 266 for MUX263, 267 for MUX264, 268 for MUX265, 269 for MUX266, 270 for MUX267, 271 for MUX268, 272 for MUX269, 273 for MUX270, 274 for MUX271, 275 for MUX272, 276 for MUX273, 277 for MUX274, 278 for MUX275, 279 for MUX276, 280 for MUX277, 281 for MUX278, 282 for MUX279, 283 for MUX280, 284 for MUX281, 285 for MUX282, 286 for MUX283, 287 for MUX284, 288 for MUX285, 289 for MUX286, 290 for MUX287, 291 for MUX288, 292 for MUX289, 293 for MUX290, 294 for MUX291, 295 for MUX292, 296 for MUX293, 297 for MUX294, 298 for MUX295, 299 for MUX296, 300 for MUX297, 301 for MUX298, 302 for MUX299, 303 for MUX300, 304 for MUX301, 305 for MUX302, 306 for MUX303, 307 for MUX304, 308 for MUX305, 309 for MUX306, 310 for MUX307, 311 for MUX308, 312 for MUX309, 313 for MUX310, 314 for MUX311, 315 for MUX312, 316 for MUX313, 317 for MUX314, 318 for MUX315, 319 for MUX316, 320 for MUX317, 321 for MUX318, 322 for MUX319, 323 for MUX320, 324 for MUX321, 325 for MUX322, 326 for MUX323, 327 for MUX324, 328 for MUX325, 329 for MUX326, 330 for MUX327, 331 for MUX328, 332 for MUX329, 333 for MUX330, 334 for MUX331, 335 for MUX332, 336 for MUX333, 337 for MUX334, 338 for MUX335, 339 for MUX336, 340 for MUX337, 341 for MUX338, 342 for MUX339, 343 for MUX340, 344 for MUX341, 345 for MUX342, 346 for MUX343, 347 for MUX344, 348 for MUX345, 349 for MUX346, 350 for MUX347, 351 for MUX348, 352 for MUX349, 353 for MUX350, 354 for MUX351, 355 for MUX352, 356 for MUX353, 357 for MUX354, 358 for MUX355, 359 for MUX356, 360 for MUX357, 361 for MUX358, 362 for MUX359, 363 for MUX360, 364 for MUX361, 365 for MUX362, 366 for MUX363, 367 for MUX364, 368 for MUX365, 369 for MUX366, 370 for MUX367, 371 for MUX368, 372 for MUX369, 373 for MUX370, 374 for MUX371, 375 for MUX372, 376 for MUX373, 377 for MUX374, 378 for MUX375, 379 for MUX376, 380 for MUX377, 381 for MUX378, 382 for MUX379, 383 for MUX380, 384 for MUX381, 385 for MUX382, 386 for MUX383, 387 for MUX384, 388 for MUX385, 389 for MUX386, 390 for MUX387, 391 for MUX388, 392 for MUX389, 393 for MUX390, 394 for MUX391, 395 for MUX392, 396 for MUX393, 397 for MUX394, 398 for MUX395, 399 for MUX396, 400 for MUX397, 401 for MUX398, 402 for MUX399, 403 for MUX400, 404 for MUX401, 405 for MUX402, 406 for MUX403, 407 for MUX404, 408 for MUX405, 409 for MUX406, 410 for MUX407, 411 for MUX408, 412 for MUX409, 413 for MUX410, 414 for MUX411, 415 for MUX412, 416 for MUX413, 417 for MUX414, 418 for MUX415, 419 for MUX416, 420 for MUX417, 421 for MUX418, 422 for MUX419, 423 for MUX420, 424 for MUX421, 425 for MUX422, 426 for MUX423, 427 for MUX424, 428 for MUX425, 429 for MUX426, 430 for MUX427, 431 for MUX428, 432 for MUX429, 433 for MUX430, 434 for MUX431, 435 for MUX432, 436 for MUX433, 437 for MUX434, 438 for MUX435, 439 for MUX436, 440 for MUX437, 441 for MUX438, 442 for MUX439, 443 for MUX440, 444 for MUX441, 445 for MUX442, 446 for MUX443, 447 for MUX444, 448 for MUX445, 449 for MUX446, 450 for MUX447, 451 for MUX448, 452 for MUX449, 453 for MUX450, 454 for MUX451, 455 for MUX452, 456 for MUX453, 457 for MUX454, 458 for MUX455, 459 for MUX456, 460 for MUX457, 461 for MUX458, 462 for MUX459, 463 for MUX460, 464 for MUX461, 465 for MUX462, 466 for MUX463, 467 for MUX464, 468 for MUX465, 469 for MUX466, 470 for MUX467, 471 for MUX468, 472 for MUX469, 473 for MUX470, 474 for MUX471, 475 for MUX472, 476 for MUX473, 477 for MUX474, 478 for MUX475, 479 for MUX476, 480 for MUX477, 481 for MUX478, 482 for MUX479, 483 for MUX480, 484 for MUX481, 485 for MUX482, 486 for MUX483, 487 for MUX484, 488 for MUX485, 489 for MUX486, 490 for MUX487, 491 for MUX488, 492 for MUX489, 493 for MUX490, 494 for MUX491, 495 for MUX492, 496 for MUX493, 497 for MUX494, 498 for MUX495, 499 for MUX496, 500 for MUX497, 501 for MUX498, 502 for MUX499, 503 for MUX500, 504 for MUX501, 505 for MUX502, 506 for MUX503, 507 for MUX504, 508 for MUX505, 509 for MUX506, 510 for MUX507, 511 for MUX508, 512 for MUX509, 513 for MUX510, 514 for MUX511, 515 for MUX512, 516 for MUX513, 517 for MUX514, 518 for MUX515, 519 for MUX516, 520 for MUX517, 521 for MUX518, 522 for MUX519, 523 for MUX520, 524 for MUX521, 525 for MUX522, 526 for MUX523, 527 for MUX524, 528 for MUX525, 529 for MUX526, 530 for MUX527, 531 for MUX528, 532 for MUX529, 533 for MUX530, 534 for MUX531, 535 for MUX532, 536 for MUX533, 537 for MUX534, 538 for MUX535, 539 for MUX536, 540 for MUX537, 541 for MUX538, 542 for MUX539, 543 for MUX540, 544 for MUX541, 545 for MUX542, 546 for MUX543, 547 for MUX544, 548 for MUX545, 549 for MUX546, 550 for MUX547, 551 for MUX548, 552 for MUX549, 553 for MUX550, 554 for MUX551, 555 for MUX552, 556 for MUX553, 557 for MUX554, 558 for MUX555, 559 for MUX556, 560 for MUX557, 561 for MUX558, 562 for MUX559, 563 for MUX560, 564 for MUX561, 565 for MUX562, 566 for MUX563, 567 for MUX564, 568 for MUX565, 569 for MUX566, 570 for MUX567, 571 for MUX568, 572 for MUX569, 573 for MUX570, 574 for MUX571, 575 for MUX572, 576 for MUX573, 577 for MUX574, 578 for MUX575, 579 for MUX576, 580 for MUX577, 581 for MUX578, 582 for MUX579, 583 for MUX580, 584 for MUX581, 585 for MUX582, 586 for MUX583, 587 for MUX584, 588 for MUX585, 589 for MUX586, 590 for MUX587, 591 for MUX588, 592 for MUX589, 593 for MUX590, 594 for MUX591, 595 for MUX592, 596 for MUX593, 597 for MUX594, 598 for MUX595, 599 for MUX596, 600 for MUX597, 601 for MUX598, 602 for MUX599, 603 for MUX600, 604 for MUX601, 605 for MUX602, 606 for MUX603, 607 for MUX604, 608 for MUX605, 609 for MUX606, 610 for MUX607, 611 for MUX608, 612 for MUX609, 613 for MUX610, 614 for MUX611, 615 for MUX612, 616 for MUX613, 617 for MUX614, 618 for MUX615, 619 for MUX616, 620 for MUX617, 621 for MUX618, 622 for MUX619, 623 for MUX620, 624 for MUX621, 625 for MUX622, 626 for MUX623, 627 for MUX624, 628 for MUX625, 629 for MUX626, 630 for MUX627, 631 for MUX628, 632 for MUX629, 633 for MUX630, 634 for MUX631, 635 for MUX632, 636 for MUX633, 637 for MUX634, 638 for MUX635, 639 for MUX636, 640 for MUX637, 641 for MUX638, 642 for MUX639, 643 for MUX640, 644 for MUX641, 645 for MUX642, 646 for MUX643, 647 for MUX644, 648 for MUX645, 649 for MUX646, 650 for MUX647, 651 for MUX648, 652 for MUX649, 653 for MUX650, 654 for MUX651, 655 for MUX652, 656 for MUX653, 657 for MUX654, 658 for MUX655, 659 for MUX656, 660 for MUX657, 661 for MUX658, 662 for MUX659, 663 for MUX660, 664 for MUX661, 665 for MUX662, 666 for MUX663, 667 for MUX664, 668 for MUX665, 669 for MUX666, 670 for MUX667, 671 for MUX668, 672 for MUX669, 673 for MUX670, 674 for MUX671, 675 for MUX672, 676 for MUX673, 677 for MUX674, 678 for MUX675, 679 for MUX676, 680 for MUX677, 681 for MUX678, 682 for MUX679, 683 for MUX680, 684 for MUX681, 685 for MUX682, 686 for MUX683, 687 for MUX684, 688 for MUX685, 689 for MUX686, 690 for MUX687, 691 for MUX688, 692 for MUX689, 693 for MUX690, 694 for MUX691, 695 for MUX692, 696 for MUX693, 697 for MUX694, 698 for MUX695, 699 for MUX696, 700 for MUX697, 701 for MUX698, 702 for MUX699, 703 for MUX700, 704 for MUX701, 705 for MUX702, 706 for MUX703, 707 for MUX704, 708 for MUX705, 709 for MUX706, 710 for MUX707, 711 for MUX708, 712 for MUX709, 713 for MUX710, 714 for MUX711, 715 for MUX712, 716 for MUX713, 717 for MUX714, 718 for MUX715, 719 for MUX716, 720 for MUX717, 721 for MUX718, 722 for MUX719, 723 for MUX720, 724 for MUX721, 725 for MUX722, 726 for MUX723, 727 for MUX724, 728 for MUX725, 729 for MUX726, 730 for MUX727, 731 for MUX728, 732 for MUX729, 733 for MUX730, 734 for MUX731, 735 for MUX732, 736 for MUX733, 737 for MUX734, 738 for MUX735, 739 for MUX736, 740 for MUX737, 741 for MUX738, 742 for MUX739, 743 for MUX740, 744 for MUX741, 745 for MUX742, 746 for MUX743, 747 for MUX744, 748 for MUX745, 749 for MUX746, 750 for MUX747, 751 for MUX748, 752 for MUX749, 753 for MUX750, 754 for MUX751, 755 for MUX752, 756 for MUX753, 757 for MUX754, 758 for MUX755, 759 for MUX756, 760 for MUX757, 761 for MUX758, 762 for MUX759, 763 for MUX760, 764 for MUX761, 765 for MUX762, 766 for MUX763, 767 for MUX764, 768 for MUX765, 769 for MUX766, 770 for MUX767, 771 for MUX768, 772 for MUX769, 773 for MUX770, 774 for MUX771, 775 for MUX772, 776 for MUX773, 777 for MUX774, 778 for MUX775, 779 for MUX776, 780 for MUX777, 781 for MUX778, 782 for MUX779, 783 for MUX780, 784 for MUX781, 785 for MUX782, 786 for MUX783, 787 for MUX784, 788 for MUX785, 789 for MUX786, 790 for MUX787, 791 for MUX788, 792 for MUX789, 793 for MUX790, 794 for MUX791, 795 for MUX792, 796 for MUX793, 797 for MUX794, 798 for MUX795, 799 for MUX796, 800 for MUX797, 801 for MUX798, 802 for MUX799, 803 for MUX800, 804 for MUX801, 805 for MUX802, 806 for MUX803, 807 for MUX804, 808 for MUX805, 809 for MUX806, 810 for MUX807, 811 for MUX808, 812 for MUX809, 813 for MUX810, 814 for MUX811, 815 for MUX812, 816 for MUX813, 817 for MUX814, 818 for MUX815, 819 for MUX816, 820 for MUX817, 821 for MUX818, 822 for MUX819, 823 for MUX820, 824 for MUX821, 825 for MUX822, 826 for MUX823, 827 for MUX824, 828 for MUX825, 829 for MUX826, 830 for MUX827, 831 for MUX828, 832 for MUX829, 833 for MUX830, 834 for MUX831, 835 for MUX832, 836 for MUX833, 837 for MUX834, 838 for MUX835, 839 for MUX836, 840 for MUX837, 841 for MUX838, 842 for MUX839, 843 for MUX840, 844 for MUX841, 845 for MUX842, 846 for MUX843, 847 for MUX844, 848 for MUX845, 849 for MUX846, 850 for MUX847, 851 for MUX848, 852 for MUX849, 853 for MUX850, 854 for MUX851, 855 for MUX852, 856 for MUX853, 857 for MUX854, 858 for MUX855, 859 for MUX856, 860 for MUX857, 861 for MUX858, 862 for MUX859, 863 for MUX860, 864 for MUX861, 865 for MUX862, 866 for MUX863, 867 for MUX864, 868 for MUX865, 869 for MUX866, 870 for MUX867, 871 for MUX868, 872 for MUX869, 873 for MUX870, 874 for MUX871, 875 for MUX872, 876 for MUX873, 877 for MUX874, 878 for MUX875, 879 for MUX876, 880 for MUX877, 881 for MUX878, 882 for MUX879, 883 for MUX880, 884 for MUX881, 885 for MUX882, 886 for MUX883, 887 for MUX884, 888 for MUX885, 889 for MUX886, 890 for MUX887, 891 for MUX888, 892 for MUX889, 893 for MUX890, 894 for MUX891, 895 for MUX892, 896 for MUX893, 897 for MUX894, 898 for MUX895, 899 for MUX896, 900 for MUX897, 901 for MUX898, 902 for MUX899, 903 for MUX900, 904 for MUX901, 905 for MUX902, 906 for MUX903, 907 for MUX904, 908 for MUX905, 909 for MUX906, 910 for MUX907, 911 for MUX908, 912 for MUX909, 913 for MUX910, 914 for MUX911, 915 for MUX912, 916 for MUX913, 917 for MUX914, 918 for MUX915, 919 for MUX916, 920 for MUX917, 921 for MUX918, 922 for MUX919, 923 for MUX920, 924 for MUX921, 925 for MUX922, 926 for MUX923, 927 for MUX924, 928 for MUX925, 929 for MUX926, 930 for MUX927, 931 for MUX928, 932 for MUX929, 933 for MUX930, 934 for MUX931, 935 for MUX932, 936 for MUX933, 937 for MUX934, 938 for MUX935, 939 for MUX936, 940 for MUX937, 941 for MUX938, 942 for MUX939, 943 for MUX940, 944 for MUX941, 945 for MUX942, 946 for MUX943, 947 for MUX944, 948 for MUX945, 949 for MUX946, 950 for MUX947, 951 for MUX948, 952 for MUX949, 953 for MUX950, 954 for MUX951, 955 for MUX952, 956 for MUX953, 957 for MUX954, 958 for MUX955, 959 for MUX956, 960 for MUX957, 961 for MUX958, 962 for MUX959, 963 for MUX960, 964 for MUX961, 965 for MUX962, 966 for MUX963, 967 for MUX964, 968 for MUX965, 969 for MUX966, 970 for MUX967, 971 for MUX968, 972 for MUX969, 973 for MUX970, 974 for MUX971, 975 for MUX972, 976 for MUX973, 977 for MUX974, 978 for MUX975, 979 for MUX976, 980 for MUX977, 981 for MUX978, 982 for MUX979, 983 for MUX980, 984 for MUX981, 985 for MUX982, 986 for MUX983, 987 for MUX984, 988 for MUX985, 989 for MUX986, 990 for MUX987, 991 for MUX988, 992 for MUX989, 993 for MUX990, 994 for MUX991, 995 for MUX992, 996 for MUX993, 997 for MUX994, 998 for MUX995, 999 for MUX996, 1000 for MUX997, 1001 for MUX998, 1002 for MUX999, 1003 for MUX1000, 1004 for MUX1001, 1005 for MUX1002, 1006 for MUX1003, 1007 for MUX1004, 1008 for MUX1005, 1009 for MUX1006, 1010 for MUX1007, 1011 for MUX1008, 1012 for MUX1009, 1013 for MUX1010, 1014 for MUX1011, 1015 for MUX1012, 1016 for MUX1013, 1017 for MUX1014, 1018 for MUX1015, 1019 for MUX1016, 1020 for MUX1017, 1021 for MUX1018, 1022 for MUX1019, 1023 for MUX1020, 1024 for MUX1021, 1025 for MUX1022, 1026 for MUX1023, 1027 for MUX1024, 1028 for MUX1025, 1029 for MUX1026, 103

- **impl** – The implementation network.
- **conflict_limit** – SAT conflict limit. A value of 0 means no limit.
- **functional_reduction** – Whether to perform functional reduction of the miter before checking.
- **verbose** – Whether to print verbose progress output.

Returns True if equivalent, False if not equivalent, or None if the procedure did not finish before the configured limit.

Raises **RuntimeError** – If miter construction fails due to incompatible interfaces (PI/PO count mismatch).

cleanup_dangling(*ntk: Aig, *, remove_dangling_pis: bool = False, remove_redundant_pos: bool = False*)
→ *Aig*

Removes dangling logic (dead nodes) from a network.

Parameters

- **ntk** – The input logic network.
- **remove_dangling_pis** – Whether to also remove dangling primary inputs.
- **remove_redundant_pos** – Whether to remove redundant primary outputs.

Returns A cleaned network with dangling structures removed.

sop_refactoring(*ntk: Aig, *, max_pis: int = 6, allow_zero_gain: bool = False, use_reconvergence_cut: bool = False, use_dont_cares: bool = False, use_quick_factoring: bool = True, try_both_polarities: bool = True, consider_inverter_cost: bool = False, verbose: bool = False, inplace: bool = False*) → *Aig | None*

Performs SOP-based network refactoring.

Parameters

- **ntk** – The input logic network.
- **max_pis** – Maximum number of leaves used in local windows.
- **allow_zero_gain** – Whether substitutions with zero gain are allowed.
- **use_reconvergence_cut** – Whether to use reconvergence-driven cuts.
- **use_dont_cares** – Whether to use don't-care information.
- **use_quick_factoring** – Whether to use the quick SOP factoring heuristic.
- **try_both_polarities** – Whether both output polarities are explored.
- **consider_inverter_cost** – Whether inverter cost is included in optimization.
- **verbose** – Whether to print verbose progress output.
- **inplace** – Whether to mutate ntk in place.

Returns The refactored network if **inplace** is False. Otherwise None.

Raises **RuntimeError** – If refactoring fails in the underlying synthesis engine.

aig_resubstitution(*ntk: Aig, *, max_pis: int = 8, max_divisors: int = 150, max_inserts: int = 2, skip_fanout_limit_for_roots: int = 1000, skip_fanout_limit_for_divisors: int = 100, verbose: bool = False, use_dont_cares: bool = False, window_size: int = 12, preserve_depth: bool = False, inplace: bool = False*) → *Aig | None*

Performs AIG resubstitution-based optimization.

Parameters

- **ntk** – The input logic network.
- **max_pis** – Maximum number of leaves in a local window.
- **max_divisors** – Maximum number of candidate divisors.
- **max_inserts** – Maximum number of inserted nodes per replacement.
- **skip_fanout_limit_for_roots** – Fanout threshold to skip root candidates.
- **skip_fanout_limit_for_divisors** – Fanout threshold to skip divisor candidates.
- **verbose** – Whether to print verbose progress output.
- **use_dont_cares** – Whether to use don't-care information.
- **window_size** – Window size used for don't-care computation.
- **preserve_depth** – Whether replacements must preserve depth.
- **inplace** – Whether to mutate ntk in place.

Returns The optimized network if **inplace** is False. Otherwise None.

aig_cut_rewriting(*ntk*: Aig, *, *cut_size*: int = 4, *cut_limit*: int = 8, *minimize_truth_table*: bool = True, *allow_zero_gain*: bool = False, *use_dont_cares*: bool = False, *min_cand_cut_size*: int = 3, *min_cand_cut_size_override*: int | None = None, *preserve_depth*: bool = False, *verbose*: bool = False, *very_verbose*: bool = False) → Aig

Rewrites an AIG network using cut-based NPN resynthesis.

Parameters

- **ntk** – The input logic network.
- **cut_size** – Maximum cut size used during cut enumeration.
- **cut_limit** – Maximum number of cuts retained per node.
- **minimize_truth_table** – Whether to minimize cut truth tables.
- **allow_zero_gain** – Whether replacements with zero gain are allowed.
- **use_dont_cares** – Whether to use don't-care information.
- **min_cand_cut_size** – Minimum candidate cut size.
- **min_cand_cut_size_override** – Optional override for minimum candidate cut size.
- **preserve_depth** – Whether replacements must preserve network depth.
- **verbose** – Whether to print verbose progress output.
- **very_verbose** – Whether to print highly detailed progress output.

Returns A rewritten network.

balancing(*ntk*: Aig, *, *cut_size*: int = 4, *cut_limit*: int = 8, *minimize_truth_table*: bool = True, *only_on_critical_path*: bool = False, *rebalance_function*: Literal['sop', 'esop'] = 'sop', *sop_both_phases*: bool = True, *verbose*: bool = False) → Aig

Balances a network using SOP or ESOP-based local restructuring.

Parameters

- **ntk** – The input logic network.
- **cut_size** – Maximum cut size used during cut enumeration.
- **cut_limit** – Maximum number of cuts retained per node.
- **minimize_truth_table** – Whether to minimize cut truth tables during enumeration.
- **only_on_critical_path** – Whether to balance only nodes on the critical path.
- **rebalance_function** – Rebalancing engine to use. Supported values are "sop" and "esop".
- **sop_both_phases** – Whether to consider both phases in SOP/ESOP balancing.
- **verbose** – Whether to print verbose progress output.

Returns A new balanced network.

Raises **ValueError** – If *rebalance_function* is not one of the supported values.

simulate(*ntk*: Aig) → list[TruthTable]

Simulates all primary outputs of a network as truth tables.

Parameter **ntk** – The input logic network.

Returns A list containing one truth table per primary output.

Raises **MemoryError** – If the truth tables cannot be allocated due to memory limits.

simulate_nodes(*ntk*: Aig) → dict[int, TruthTable]

Simulates all nodes of a network as truth tables.

Parameter **ntk** – The input logic network.

Returns A dictionary that maps node identifiers to truth tables.

Raises **MemoryError** – If the truth tables cannot be allocated due to memory limits.

aigverse.generators

Provides generators for random and structured benchmark construction.

Module Contents

random_aig(*, *num_pis*: int, *num_gates*: int, *seed*: int = 3405688830) → Aig

Generates a single random AIG with a fixed size.

Parameters

- **num_pis** – Number of primary inputs.
- **num_gates** – Number of logic gates.

- **seed** – Seed controlling random choices.

Returns A randomly generated AIG.

Raises

- **ValueError** – If `num_pis` or `num_gates` is 0.
- **TypeError** – If `num_pis` or `num_gates` cannot be converted to `uint32`.

ripple_carry_adder(*bitwidth: int*) → *Aig*

Creates a complete ripple-carry adder benchmark network.

Parameter **bitwidth** – Number of bits per operand.

Returns An Aig with $2 * \text{bitwidth}$ primary inputs and $\text{bitwidth} + 1$ primary outputs (sum plus carry-out).

Raises **ValueError** – If `bitwidth` is not greater than 0.

carry_lookahead_adder(*bitwidth: int*) → *Aig*

Creates a complete carry-lookahead adder benchmark network.

Parameter **bitwidth** – Number of bits per operand.

Returns An Aig with $2 * \text{bitwidth}$ primary inputs and $\text{bitwidth} + 1$ primary outputs (sum plus carry-out).

Raises **ValueError** – If `bitwidth` is not greater than 0.

ripple_carry_multiplier(*bitwidth: int*) → *Aig*

Creates a complete ripple-carry multiplier benchmark network.

Parameter **bitwidth** – Number of bits per operand.

Returns An Aig with $2 * \text{bitwidth}$ primary inputs and $2 * \text{bitwidth}$ primary outputs representing the product bits.

Raises **ValueError** – If `bitwidth` is not greater than 0.

sideways_sum_adder(*bitwidth: int*) → *Aig*

Creates a complete sideways sum adder benchmark network.

Parameter **bitwidth** – Number of input bits.

Returns An Aig with `bitwidth` primary inputs and output bits encoding the population count of the input word.

Raises **ValueError** – If `bitwidth` is not greater than 0.

multiplexer(*bitwidth: int*) → *Aig*

Creates a complete word-level n-bit 2:1 MUX network.

Parameter **bitwidth** – Number of bits in each data input word.

Returns An Aig with $1 + 2 * \text{bitwidth}$ primary inputs and `bitwidth` primary outputs.

Raises **ValueError** – If `bitwidth` is not greater than 0.

binary_decoder(*num_select_bits: int*) → *Aig*

Creates a complete binary-decoder network.

Parameter **num_select_bits** – Number of select input bits.

Returns An Aig with `num_select_bits` primary inputs and $2 ** \text{num_select_bits}$ primary outputs.

Raises **ValueError** – If `num_select_bits` is not greater than 0.

aigverse.io

Provides file import and export functions for logic networks.

The module contains readers and writers for common file formats in the domain.

Module Contents

read_aiger_into_aig(*filename: str | PathLike*) → *NamedAig*

Reads a binary AIGER file into a logic network.

Parameter **filename** – Path to the AIGER file.

Returns The parsed network instance.

Raises **RuntimeError** – If parsing the AIGER file fails.

read_ascii_aiger_into_aig(filename: str | PathLike) → NamedAig

Reads an ASCII AIGER file into a logic network.

Parameter **filename** – Path to the ASCII AIGER file.

Returns The parsed network instance.

Raises **RuntimeError** – If parsing the ASCII AIGER file fails.

read_aiger_into_sequential_aig(filename: str | PathLike) → SequentialAig

Reads a binary AIGER file into a logic network.

Parameter **filename** – Path to the AIGER file.

Returns The parsed network instance.

Raises **RuntimeError** – If parsing the AIGER file fails.

read_ascii_aiger_into_sequential_aig(filename: str | PathLike) → SequentialAig

Reads an ASCII AIGER file into a logic network.

Parameter **filename** – Path to the ASCII AIGER file.

Returns The parsed network instance.

Raises **RuntimeError** – If parsing the ASCII AIGER file fails.

write_aiger(ntk: Aig, filename: str | PathLike) → None

Writes a logic network to a binary AIGER file.

Parameters

- **ntk** – The network to serialize.
- **filename** – Destination path for the AIGER file.

read_pla_into_aig(filename: str | PathLike) → Aig

Reads a PLA file into a logic network.

Parameter **filename** – Path to the PLA file.

Returns The parsed network instance.

Raises **RuntimeError** – If parsing the PLA file fails.

read_verilog_into_aig(filename: str | PathLike) → NamedAig

Reads a synthesized gate-level Verilog netlist into a logic network.

Parameter **filename** – Path to the Verilog file.

Returns The parsed network instance.

Raises **RuntimeError** – If parsing the Verilog file fails.

write_verilog(ntk: Aig, filename: str | PathLike) → None

Writes a logic network to a Verilog netlist.

Parameters

- **ntk** – The network to serialize.
- **filename** – Destination path for the Verilog file.

write_dot(ntk: Aig, filename: str | PathLike) → None

Writes a logic network to a Graphviz DOT file for visualization.

Parameters

- **ntk** – The network to serialize.
- **filename** – Destination path for the DOT file.

aigverse.networks

Provides logic network data structures and derived views.

This module includes network types, edge and index list utilities, and helper objects for structural manipulation.

Module Contents

class AigSignal(*index: int, complement: bool*)

Represents a signal in an AIG.

Signals point to nodes and may be complemented.

property index: int

Node index referenced by the signal.

property complement: bool

Whether this signal is complemented.

property data: int

Raw packed signal representation.

__hash__() → int

Returns a hash value for dictionary/set usage.

__eq__(*other: object*) → bool

Returns whether two signals are equal.

Parameter **other** – Object to compare.

Returns True if equal, otherwise False.

__ne__(*other: object*) → bool

Returns whether two signals are not equal.

Parameter **other** – Object to compare.

Returns True if not equal, otherwise False.

__lt__(*other: AigSignal*) → bool

Returns whether this signal is ordered before another signal.

__invert__() → *AigSignal*

Returns the complemented signal.

__pos__() → *AigSignal*

Returns a normalized positive-phase signal.

__neg__() → *AigSignal*

Returns a normalized negative-phase signal.

__xor__(*complement: bool*) → *AigSignal*

XORs the signal phase with a Boolean complement bit.

Parameter **complement** – Complement bit to XOR with the current signal phase.

Returns A phase-adjusted signal.

class Aig

Represents an AIG and its structural operations.

Note

`to_index_list()` keeps combinational structure only. Augmented view metadata is not preserved.

clone() → *Aig*

Creates a structural copy of the network.

property size: int

Number of nodes in the network.

property num_gates: int
 Number of logic gates in the network.

property num_pis: int
 Number of primary inputs.

property num_pos: int
 Number of primary outputs.

get_node(*s: AigSignal*) → int
 Returns the node referenced by a signal.

make_signal(*n: int*) → *AigSignal*
 Creates a signal from a node.

is_complemented(*s: AigSignal*) → bool
 Returns whether a signal is complemented.

node_to_index(*n: int*) → int
 Returns the integer index of a node.

index_to_node(*index: int*) → int
 Returns the node for an index.

pi_index(*n: int*) → int
 Returns the primary-input position of a node.

pi_at(*index: int*) → int
 Returns the primary input node at *index*.

po_index(*s: AigSignal*) → int
 Returns the primary-output position of a signal.

po_at(*index: int*) → *AigSignal*
 Returns the primary output signal at *index*.

get_constant(*value: bool*) → *AigSignal*
 Returns the constant signal for a Boolean value.

create_pi() → *AigSignal*
 Creates and returns a new primary input signal.

create_po(*f: AigSignal*) → int
 Creates a primary output from signal *f*.

property is_combinational: bool
 Whether the network is combinational.

create_buf(*a: AigSignal*) → *AigSignal*
 Creates a buffer.

create_not(*a: AigSignal*) → *AigSignal*
 Creates an inversion.

create_and(*a: AigSignal, b: AigSignal*) → *AigSignal*
 Creates an AND.

create_nand(*a: AigSignal, b: AigSignal*) → *AigSignal*
 Creates a NAND.

create_or(*a: AigSignal, b: AigSignal*) → *AigSignal*
 Creates an OR.

create_nor(*a: AigSignal, b: AigSignal*) → *AigSignal*
 Creates a NOR.

create_xor(*a: AigSignal, b: AigSignal*) → *AigSignal*
 Creates an XOR.

create_xnor(*a: AigSignal, b: AigSignal*) → *AigSignal*
 Creates an XNOR.

create_lt(*a: AigSignal, b: AigSignal*) → *AigSignal*
 Creates a less-than comparator.

create_le(*a: AigSignal, b: AigSignal*) → *AigSignal*
 Creates a less-or-equal comparator.

create_maj(*a: AigSignal, b: AigSignal, c: AigSignal*) → *AigSignal*
 Creates a majority.

create_ite(*cond: AigSignal, f_then: AigSignal, f_else: AigSignal*) → *AigSignal*
 Creates an if-then-else.

create_xor3(*a: AigSignal, b: AigSignal, c: AigSignal*) → *AigSignal*
 Creates a 3-input XOR.

create_nary_and(*fs: Sequence[AigSignal]*) → *AigSignal*
 Creates an n-ary AND.

create_nary_or(*fs: Sequence[AigSignal]*) → *AigSignal*
 Creates an n-ary OR.

create_nary_xor(*fs: Sequence[AigSignal]*) → *AigSignal*
 Creates an n-ary XOR.

clone_node(*other: Aig, source: int, children: Sequence[AigSignal]*) → *AigSignal*
 Clones one node from *other* into this network.

nodes() → *list[int]*
 Returns a list of all nodes in order of creation.

gates() → *list[int]*
 Returns a list of all non-constant and non-PI nodes in order of creation.

pis() → *list[int]*
 Returns a list of all primary input nodes in order of creation.

pos() → *list[AigSignal]*
 Returns a list of all primary output signals in order of creation.

fanins(*n: int*) → *list[AigSignal]*
 Returns fanin signals of node *n*.

fanin_size(*n: int*) → *int*
 Returns the number of fanins of node *n*.

fanout_size(*n: int*) → *int*
 Returns the number of fanouts of node *n*.

is_constant(*n: int*) → *bool*
 Returns whether *n* is a constant node.

is_pi(*n: int*) → *bool*
 Returns whether *n* is a primary input.

has_and(*a*: AigSignal, *b*: AigSignal) → *AigSignal* | None

Returns whether an AND with fanins *a* and *b* already exists.

is_and(*n*: int) → bool

Returns whether *n* is an AND node.

is_or(*n*: int) → bool

Returns whether *n* is an OR node.

is_xor(*n*: int) → bool

Returns whether *n* is an XOR node.

is_maj(*n*: int) → bool

Returns whether *n* is a majority node.

is_ite(*n*: int) → bool

Returns whether *n* is an if-then-else node.

is_xor3(*n*: int) → bool

Returns whether *n* is a 3-input XOR node.

is_nary_and(*n*: int) → bool

Returns whether *n* is an n-ary AND node.

is_nary_or(*n*: int) → bool

Returns whether *n* is an n-ary OR node.

to_edge_list(*regular_weight*: int = 0, *inverted_weight*: int = 1) → *AigEdgeList*

Converts the network to an edge list.

Parameters

- **regular_weight** – Weight assigned to non-inverted edges.
- **inverted_weight** – Weight assigned to inverted edges.

Returns The corresponding edge-list representation.

to_index_list() → *AigIndexList*

Converts the network to an index-list encoding.

Returns The corresponding index-list representation.

to_networkx(* , *levels*: bool = False, *fanouts*: bool = False, *node_tts*: bool = False, *graph_tts*: bool = False, *dtype*: type[*generic*] = ...) → *DiGraph*

Converts an *Aig* to a *DiGraph*.

This function transforms the AIG into a directed graph representation using the NetworkX library. It allows for the inclusion of various attributes for the graph, its nodes, and edges, making it suitable for graph-based machine learning tasks.

Note that the constant-0 node is always included in the graph, as index 0, even if it is not referenced by any edges.

Parameters

- **self** – The AIG object to convert.
- **levels** – If True, computes and adds level information for each node and the total number of levels to the graph, as attributes *level* and *levels*, respectively. Defaults to False.
- **fanouts** – If True, adds the fanout count for each node as an integer *fanouts* attribute (0 for synthetic PO nodes). Defaults to False.
- **node_tts** – If True, computes and adds a truth table for each node as a *function* attribute. Defaults to False.
- **graph_tts** – If True, computes and adds the graph's overall truth table as a *function* attribute to the graph. Defaults to False.

- **dtype** – The data type for truth tables and all one-hot encodings. Defaults to `int8`. For machine learning tasks, a floating-point type such as `float32` or `float64` may be more appropriate, as it allows for gradient-based optimization.

Returns A `DiGraph` representing the AIG.

Graph Attributes:

- `type` (str): "AIG".
- `num_pis` (int): Number of primary inputs.
- `num_pos` (int): Number of primary outputs.
- `num_gates` (int): Number of AND gates.
- `levels` (int, optional): Total number of levels in the AIG.
- `function` (list[`ndarray`], optional): Graph's truth tables.
- `name` (str, optional): Network name (only for `NamedAig`).

Node Attributes:

- `index` (int): The node's identifier.
- `level` (int, optional): The level of the node in the AIG.
- `fanouts` (int, optional): Fanouts of the node.
- `function` (`ndarray`, optional): The node's truth table.
- `type` (ndarray, optional): The node type (constant, vector, pos). The data type is determined by the `dtype` argument, defaulting to `int8`.

Edge Attributes:

- `type` (`ndarray`, optional): The data type is determined by the `dtype` argument, defaulting to `int8`.
- `name` (str, optional): Primary output name for edges to synthetic

`__len__()` → int

Returns the number of nodes.

`__iter__()` → `Iterator[int]`

Returns an iterator over all nodes.

`__contains__(value: object)` → bool

Returns whether a node index is contained in the network.

`__bool__()` → bool

Returns True for non-trivial networks.

`__copy__()` → `Aig`

Returns a shallow structural copy.

`__deepcopy__(memo: dict)` → `Aig`

Returns a deep structural copy.

`__setstate__(state: object)` → `None`

Restores a network from a pickled state tuple.

Parameter `state` – Tuple containing one index-list payload.

Raises `ValueError` – If the state shape or payload is invalid.

`__getstate__()` → tuple

Returns pickle state as an index-list tuple.

Preserves only combinational structure and does not capture augmented view metadata.

class `NamedAig`

class `NamedAig(ntk: NamedAig)`

class `NamedAig(ntk: Aig)`

Bases: `Aig`

Extends a network with input/output and node names.

clone() → *NamedAig*
 Creates a structural copy including names.

__copy__() → *NamedAig*
 Returns a shallow copy of the named view.

__deepcopy__(memo: dict) → *NamedAig*
 Returns a deep copy of the named view.

create_pi(name: str = "") → *AigSignal*
 Creates a primary input with an optional name.

create_po(f: AigSignal, name: str = "") → int
 Creates a primary output with an optional name.

set_network_name(name: str) → None
 Sets the network name.

get_network_name() → str
 Returns the network name.

has_name(s: AigSignal) → bool
 Returns whether signal *s* has a name.

set_name(s: AigSignal, name: str) → None
 Assigns a name to signal *s*.

get_name(s: AigSignal) → str
 Returns the name of signal *s*.

has_output_name(index: int) → bool
 Returns whether output *index* has a name.

set_output_name(index: int, name: str) → None
 Sets the name of output *index*.

get_output_name(index: int) → str
 Returns the name of output *index*.

class DepthAig

class DepthAig(ntk: DepthAig)

class DepthAig(ntk: Aig)

Bases: *Aig*

Extends a network with depth information.

clone() → *DepthAig*

Creates a copy of the depth view.

__copy__() → *DepthAig*

Returns a shallow copy of the depth view.

__deepcopy__(memo: dict) → *DepthAig*

Returns a deep copy of the depth view.

property num_levels: int

Current network depth in levels.

level(n: int) → int

Returns the level of node *n*.

is_on_critical_path(*n: int*) → bool
Returns whether node *n* is on at least one critical path.

update_levels() → None
Recomputes level information.

create_po(*f: AigSignal*) → int
Creates an output and updates depth information.

class FanoutAig
class FanoutAig(*ntk: Aig*)
class FanoutAig(*ntk: FanoutAig*)
Bases: *Aig*
Extends a network with fanout information.

clone() → *FanoutAig*
Creates a structural copy of the fanout view.

__copy__() → *FanoutAig*
Returns a shallow copy of the fanout view.

__deepcopy__(*memo: dict*) → *FanoutAig*
Returns a deep copy of the fanout view.

fanouts(*n: int*) → list[int]
Returns fanout nodes of node *n*.

class AigRegister
class AigRegister(*register: AigRegister*)
Represents metadata for one sequential register.

property control: str
Optional control signal.

property init: int
Initial value of the register.

property type: str
Register type/category.

class SequentialAig
Bases: *Aig*
Represents a sequential network with register interfaces.

clone() → *SequentialAig*
Creates a structural copy of the sequential network.

__copy__() → *SequentialAig*
Returns a shallow copy of the sequential network.

__deepcopy__(*memo: dict*) → *SequentialAig*
Returns a deep copy of the sequential network.

create_pi() → *AigSignal*
Creates a primary input.

create_po(*f: AigSignal*) → int
Creates a primary output.

create_ro() → *AigSignal*
Creates a register output node.

create_ri(f: AigSignal) → *int*
Creates a register input signal.

property is_combinational: bool
Whether the network is combinational.

is_ci(n: int) → *bool*
Returns whether n is a combinational input.

is_pi(n: int) → *bool*
Returns whether n is a primary input.

is_ro(n: int) → *bool*
Returns whether n is a register output.

property num_pis: int
Number of primary inputs.

property num_pos: int
Number of primary outputs.

property num_cis: int
Number of combinational inputs.

property num_cos: int
Number of combinational outputs.

property num_registers: int
Number of registers.

pi_at(index: int) → *int*
Returns primary input at *index*.

po_at(index: int) → *AigSignal*
Returns primary output at *index*.

ci_at(index: int) → *int*
Returns combinational input at *index*.

co_at(index: int) → *AigSignal*
Returns combinational output at *index*.

ro_at(index: int) → *int*
Returns register output at *index*.

ri_at(index: int) → *AigSignal*
Returns register input at *index*.

set_register(index: int, reg: AigRegister) → *None*
Sets metadata for register *index*.

register_at(index: int) → *AigRegister*
Returns metadata for register *index*.

pi_index(n: int) → *int*
Returns PI index of node n.

ci_index(n: int) → *int*
Returns CI index of node n.

co_index(*s*: *AigSignal*) → *int*
Returns CO index of signal *s*.

ro_index(*n*: *int*) → *int*
Returns RO index of node *n*.

ri_index(*s*: *AigSignal*) → *int*
Returns RI index of signal *s*.

ro_to_ri(*s*: *AigSignal*) → *AigSignal*
Maps a register output signal to its input.

ri_to_ro(*s*: *AigSignal*) → *int*
Maps a register input signal to its output.

to_index_list() → *NoReturn*
Sequential networks cannot be encoded as combinational index lists.

__getstate__() → *NoReturn*
Sequential networks are not pickleable via combinational index-list state.

__setstate__(*state*: *object*) → *NoReturn*
Sequential networks cannot be restored from combinational index-list state.

to_edge_list(*regular_weight*: *int* = 0, *inverted_weight*: *int* = 1) → *AigEdgeList*
Converts the sequential network to an edge list.

pis() → *list[int]*
Returns all primary input nodes.

pos() → *list[AigSignal]*
Returns all primary output signals.

cis() → *list[int]*
Returns all combinational input nodes.

cos() → *list[AigSignal]*
Returns all combinational output signals.

ros() → *list[int]*
Returns all register output nodes.

ris() → *list[AigSignal]*
Returns all register input signals.

registers() → *list[tuple[AigSignal, int]]*
Returns all register pairs as (*ri_signal*, *ro_node*) tuples.

class AigEdge

class AigEdge(*source*: *int*, *target*: *int*, *weight*: *int* = 0)
Represents a directed edge in a logic network graph. A weight attribute may encode inversion.

property source: *int*
Source node identifier.

property target: *int*
Target node identifier.

property weight: *int*
Edge weight value.

`__eq__(other: object) → bool`

Checks whether two edges are equal.

Parameter **other** – Object to compare.

Returns True if **other** is an edge with equal fields, otherwise False.

`__ne__(other: object) → bool`

Checks whether two edges are not equal.

Parameter **other** – Object to compare.

Returns True if **other** is not equal to this edge.

class AigEdgeList

class AigEdgeList(*ntk: Aig*)

class AigEdgeList(*edges: Sequence[AigEdge]*)

class AigEdgeList(*ntk: Aig, edges: Sequence[AigEdge]*)

Represents a list of edges associated with a network.

property ntk: Aig

Underlying network associated with this list.

property edges: list[AigEdge]

Stored edges in insertion order.

append(*edge: AigEdge*) → None

Appends an edge to the list.

Parameter **edge** – Edge to append.

clear() → None

Removes all edges from the list.

__iter__() → Iterator[AigEdge]

Returns an iterator over stored edges.

__len__() → int

Returns the number of edges.

__getitem__(*index: int*) → AigEdge

Returns the edge at a given position.

Parameter **index** – Edge index. Negative indices are supported.

Returns The edge at the requested position.

Raises **IndexError** – If **index** is out of bounds.

__setitem__(*index: int, edge: AigEdge*) → None

Replaces the edge at a given position.

Parameters

- **index** – Edge index. Negative indices are supported.
- **edge** – Replacement edge.

Raises **IndexError** – If **index** is out of bounds.

class AigIndexList(*num_pis: int = 0*)

class AigIndexList(*values: Sequence[int]*)

Represents an index-list encoding of an AIG network.

raw() → list[int]

Returns the raw integer encoding.

Returns A list of encoded index-list values.

property size: int
 Number of raw entries in the encoding.

property num_gates: int
 Number of encoded gates.

property num_pis: int
 Number of encoded primary inputs.

property num_pos: int
 Number of encoded primary outputs.

add_inputs(*n: int = 1*) → None
 Appends primary inputs to the encoding.
 Parameters
n – Number of inputs to append.

add_and(*lit0: int, lit1: int*) → int
 Appends an AND gate to the encoding.
 Parameters

- **lit0** – First fanin literal.
- **lit1** – Second fanin literal.

add_xor(*lit0: int, lit1: int*) → int
 Appends an XOR gate to the encoding.
 Parameters

- **lit0** – First fanin literal.
- **lit1** – Second fanin literal.

add_output(*lit: int*) → None
 Appends a primary output literal.
 Parameters
lit – Output literal.

clear() → None
 Clears all encoded data.

to_aig() → Aig
 Decodes the index list into an AIG network.
 Returns A decoded AIG network.

gates() → list[tuple[int, int]]
 Returns encoded gate fanins as literal pairs.
 Returns A list of (**lit0**, **lit1**) tuples for each gate.

pos() → list[int]
 Returns encoded primary output literals.
 Returns A list of output literals.

__iter__() → Iterator[int]
 Returns an iterator over the raw encoding values.

__getitem__(*index: int*) → int
 Returns one raw encoding value by index.
 Parameters
index – Position in the raw encoding.
 Returns The raw value at **index**.
 Raises **IndexError** – If **index** is out of range.

`__setitem__(index: int, value: int) → None`

Sets one raw encoding value by index.

Parameters

- **index** – Position in the raw encoding.
- **value** – Replacement raw value.

Raises **IndexError** – If **index** is out of range.

`__len__() → int`

Returns the number of raw encoding entries.

aigverse.utils

Provides utility data structures and functions.

Module Contents

class TruthTable(*num_vars: int*)

Represents a dynamic Boolean truth table.

`__eq__(other: object) → bool`

Checks equality with another truth table.

`__ne__(other: object) → bool`

Checks inequality with another truth table.

`__lt__(other: TruthTable) → bool`

Lexicographically compares two truth tables.

`__and__(other: TruthTable) → TruthTable`

Computes bitwise AND with another truth table.

`__or__(other: TruthTable) → TruthTable`

Computes bitwise OR with another truth table.

`__xor__(other: TruthTable) → TruthTable`

Computes bitwise XOR with another truth table.

`__invert__() → TruthTable`

Computes bitwise NOT.

`__len__() → int`

Returns the number of bits.

`__getitem__(index: int) → bool`

Returns one bit by index.

Parameter **index** – Bit index.

Returns The bit value.

Raises **IndexError** – If **index** is out of range.

`__setitem__(index: int, value: bool) → None`

Sets one bit by index.

Parameters

- **index** – Bit index.
- **value** – New bit value.

Raises **IndexError** – If **index** is out of range.

`__iter__() → Iterator[bool]`

Returns an iterator over all bits.

`num_vars() → int`

Returns the number of variables.

num_blocks() → int
Returns the number of storage blocks.

num_bits() → int
Returns the number of truth table bits.

__copy__() → *TruthTable*
Returns a shallow copy of the truth table.

__deepcopy__(arg: dict, / (Positional-only parameter separator (PEP 570))) → *TruthTable*
Returns a deep copy of the truth table.

__assign__(other: TruthTable) → *TruthTable*
Assigns from another truth table with a compatible shape.
Parameter **other** – Source truth table.
Returns The updated truth table.

__hash__() → int
Returns a hash value for dictionary/set usage.

__getstate__() → tuple
Returns pickle state as (num_vars, words).

__setstate__(state: tuple) → None
Restores a truth table from pickle state.
Parameter **state** – Tuple (num_vars, words).
Raises

- **RuntimeError** – If the serialized state is malformed.
- **TypeError** – If nanobind cannot convert the pickle payload to the expected C++ types.

set_bit(index: int) → None
Sets one bit to 1.
Parameter **index** – Bit index.
Raises **IndexError** – If **index** is out of range.

get_bit(index: int) → bool
Returns one bit value.
Parameter **index** – Bit index.
Returns The bit value.
Raises **IndexError** – If **index** is out of range.

clear_bit(index: int) → None
Clears one bit to 0.
Parameter **index** – Bit index.
Raises **IndexError** – If **index** is out of range.

flip_bit(index: int) → None
Toggles one bit value.
Parameter **index** – Bit index.
Raises **IndexError** – If **index** is out of range.

get_block(block_index: int) → int
Returns one 64-bit storage block.
Parameter **block_index** – Block index.
Returns The block value.
Raises **IndexError** – If **block_index** is out of range.

create_nth_var(*var_index: int, complement: bool = False*) → *None*

Creates the projection function for one variable.

Parameters

- **var_index** – Variable index to project.
- **complement** – Whether to complement the projection.

Raises **IndexError** – If *var_index* is out of range.

create_from_binary_string(*binary: str*) → *None*

Loads bit values from a binary string.

Parameter **binary** – Binary string of length *num_bits*.

Raises **ValueError** – If the string length does not match *num_bits*.

create_from_hex_string(*hexadecimal: str*) → *None*

Loads bit values from a hexadecimal string.

Parameter **hexadecimal** – Hex string matching the truth table size.

Raises **ValueError** – If the string length does not match the expected size.

create_random() → *None*

Fills the truth table with random bits.

create_majority() → *None*

Fills the truth table with the majority function.

clear() → *None*

Clears all bits to 0.

count_ones() → *int*

Returns the number of set bits.

count_zeroes() → *int*

Returns the number of cleared bits.

is_const0() → *bool*

Returns whether all bits are 0.

is_const1() → *bool*

Returns whether all bits are 1.

to_binary() → *str*

Returns the truth table as a binary string.

to_hex() → *str*

Returns the truth table as a hexadecimal string.

ternary_majority(*a: TruthTable, b: TruthTable, c: TruthTable*) → *TruthTable*

Computes the ternary majority of three truth tables.

Parameters

- **a** – First truth table.
- **b** – Second truth table.
- **c** – Third truth table.

Returns The bitwise majority truth table.

cofactor0(*tt: TruthTable, var_index: int*) → *TruthTable*

Computes the cofactor with respect to assigning one variable to 0.

Parameters

- **tt** – Input truth table.
- **var_index** – Index of the variable to cofactor.

Returns The cofactored truth table with *var_index* fixed to 0.

Raises **ValueError** – If `var_index` is out of range.

cofactor1(*tt: TruthTable, var_index: int*) → *TruthTable*

Computes the cofactor with respect to assigning one variable to 1.

Parameters

- **tt** – Input truth table.
- **var_index** – Index of the variable to cofactor.

Returns The cofactored truth table with `var_index` fixed to 1.

Raises **ValueError** – If `var_index` is out of range.

VII-B Package Contents

__version__: `str`

Index

Symbols

`__and__` () (*TruthTable method*), 40
`__assign__` () (*TruthTable method*), 41
`__bool__` () (*Aig method*), 33
`__contains__` () (*Aig method*), 33
`__copy__` () (*Aig method*), 33
`__copy__` () (*DepthAig method*), 34
`__copy__` () (*FanoutAig method*), 35
`__copy__` () (*NamedAig method*), 34
`__copy__` () (*SequentialAig method*), 35
`__copy__` () (*TruthTable method*), 41
`__deepcopy__` () (*Aig method*), 33
`__deepcopy__` () (*DepthAig method*), 34
`__deepcopy__` () (*FanoutAig method*), 35
`__deepcopy__` () (*NamedAig method*), 34
`__deepcopy__` () (*SequentialAig method*), 35
`__deepcopy__` () (*TruthTable method*), 41
`__eq__` () (*AigEdge method*), 37
`__eq__` () (*AigSignal method*), 29
`__eq__` () (*TruthTable method*), 40
`__getitem__` () (*AigEdgeList method*), 38
`__getitem__` () (*AigIndexList method*), 39
`__getitem__` () (*TruthTable method*), 40
`__getstate__` () (*Aig method*), 33
`__getstate__` () (*SequentialAig method*), 37
`__getstate__` () (*TruthTable method*), 41
`__hash__` () (*AigSignal method*), 29
`__hash__` () (*TruthTable method*), 41
`__invert__` () (*AigSignal method*), 29
`__invert__` () (*TruthTable method*), 40
`__iter__` () (*Aig method*), 33
`__iter__` () (*AigEdgeList method*), 38
`__iter__` () (*AigIndexList method*), 39
`__iter__` () (*TruthTable method*), 40
`__len__` () (*Aig method*), 33
`__len__` () (*AigEdgeList method*), 38
`__len__` () (*AigIndexList method*), 40
`__len__` () (*TruthTable method*), 40
`__lt__` () (*AigSignal method*), 29
`__lt__` () (*TruthTable method*), 40
`__ne__` () (*AigEdge method*), 38
`__ne__` () (*AigSignal method*), 29
`__ne__` () (*TruthTable method*), 40
`__neg__` () (*AigSignal method*), 29
`__or__` () (*TruthTable method*), 40
`__pos__` () (*AigSignal method*), 29
`__setitem__` () (*AigEdgeList method*), 38
`__setitem__` () (*AigIndexList method*), 39
`__setitem__` () (*TruthTable method*), 40
`__setstate__` () (*Aig method*), 33
`__setstate__` () (*SequentialAig method*), 37
`__setstate__` () (*TruthTable method*), 41
`__version__` (*in module aigverse*), 43
`__xor__` () (*AigSignal method*), 29
`__xor__` () (*TruthTable method*), 40

A

`add_and` () (*AigIndexList method*), 39
`add_inputs` () (*AigIndexList method*), 39
`add_output` () (*AigIndexList method*), 39
`add_xor` () (*AigIndexList method*), 39
`Aig` (*class in aigverse.networks*), 29
`aig_cut_rewriting` () (*in module aigverse.algorithms*), 25
`aig_resubstitution` () (*in module aigverse.algorithms*), 25
`AigEdge` (*class in aigverse.networks*), 37
`AigEdgeList` (*class in aigverse.networks*), 38
`AigIndexList` (*class in aigverse.networks*), 38
`AigRegister` (*class in aigverse.networks*), 35
`AigSignal` (*class in aigverse.networks*), 29
`aigverse`
 module, 23
`aigverse.adapters`

module, 23
`aigverse.adapters.networkx`
 module, 24
`aigverse.algorithms`
 module, 24
`aigverse.generators`
 module, 26
`aigverse.io`
 module, 27
`aigverse.networks`
 module, 28
`aigverse.utils`
 module, 40
`append` () (*AigEdgeList method*), 38

B

`balancing` () (*in module aigverse.algorithms*), 26
`binary_decoder` () (*in module aigverse.generators*), 27

C

`carry_lookahead_adder` () (*in module aigverse.generators*), 27
`ci_at` () (*SequentialAig method*), 36
`ci_index` () (*SequentialAig method*), 36
`cis` () (*SequentialAig method*), 37
`cleanup_dangling` () (*in module aigverse.algorithms*), 25
`clear` () (*AigEdgeList method*), 38
`clear` () (*AigIndexList method*), 39
`clear` () (*TruthTable method*), 42
`clear_bit` () (*TruthTable method*), 41
`clone` () (*Aig method*), 29
`clone` () (*DepthAig method*), 34
`clone` () (*FanoutAig method*), 35
`clone` () (*NamedAig method*), 33
`clone` () (*SequentialAig method*), 35
`clone_node` () (*Aig method*), 31
`co_at` () (*SequentialAig method*), 36
`co_index` () (*SequentialAig method*), 36
`cofactor0` () (*in module aigverse.utils*), 42
`cofactor1` () (*in module aigverse.utils*), 43
`complement` (*AigSignal property*), 29
`control` (*AigRegister property*), 35
`cos` () (*SequentialAig method*), 37
`count_ones` () (*TruthTable method*), 42
`count_zeroes` () (*TruthTable method*), 42
`create_and` () (*Aig method*), 30
`create_buf` () (*Aig method*), 30
`create_from_binary_string` () (*TruthTable method*), 42
`create_from_hex_string` () (*TruthTable method*), 42
`create_ite` () (*Aig method*), 31
`create_le` () (*Aig method*), 31
`create_lt` () (*Aig method*), 31
`create_maj` () (*Aig method*), 31
`create_majority` () (*TruthTable method*), 42
`create_nand` () (*Aig method*), 30
`create_nary_and` () (*Aig method*), 31
`create_nary_or` () (*Aig method*), 31
`create_nary_xor` () (*Aig method*), 31
`create_nor` () (*Aig method*), 30
`create_not` () (*Aig method*), 30
`create_nth_var` () (*TruthTable method*), 41
`create_or` () (*Aig method*), 30
`create_pi` () (*Aig method*), 30
`create_pi` () (*NamedAig method*), 34
`create_pi` () (*SequentialAig method*), 35
`create_po` () (*Aig method*), 30
`create_po` () (*DepthAig method*), 35
`create_po` () (*NamedAig method*), 34
`create_po` () (*SequentialAig method*), 35
`create_random` () (*TruthTable method*), 42
`create_ri` () (*SequentialAig method*), 36
`create_ro` () (*SequentialAig method*), 35
`create_xnor` () (*Aig method*), 31

create_xor() (Aig method), 31
create_xor3() (Aig method), 31

D

data (AigSignal property), 29
DepthAig (class in aigverse.networks), 34

E

edges (AigEdgeList property), 38
equivalence_checking() (in module aigverse.algorithms), 24

F

fanin_size() (Aig method), 31
fanins() (Aig method), 31
fanout_size() (Aig method), 31
FanoutAig (class in aigverse.networks), 35
fanouts() (FanoutAig method), 35
flip_bit() (TruthTable method), 41

G

gates() (Aig method), 31
gates() (AigIndexList method), 39
get_bit() (TruthTable method), 41
get_block() (TruthTable method), 41
get_constant() (Aig method), 30
get_name() (NamedAig method), 34
get_network_name() (NamedAig method), 34
get_node() (Aig method), 30
get_output_name() (NamedAig method), 34

H

has_and() (Aig method), 31
has_name() (NamedAig method), 34
has_output_name() (NamedAig method), 34

I

index (AigSignal property), 29
index_to_node() (Aig method), 30
init (AigRegister property), 35
is_and() (Aig method), 32
is_ci() (SequentialAig method), 36
is_combinational (Aig property), 30
is_combinational (SequentialAig property), 36
is_complemented() (Aig method), 30
is_const0() (TruthTable method), 42
is_const1() (TruthTable method), 42
is_constant() (Aig method), 31
is_ite() (Aig method), 32
is_maj() (Aig method), 32
is_nary_and() (Aig method), 32
is_nary_or() (Aig method), 32
is_on_critical_path() (DepthAig method), 34
is_or() (Aig method), 32
is_pi() (Aig method), 31
is_pi() (SequentialAig method), 36
is_ro() (SequentialAig method), 36
is_xor() (Aig method), 32
is_xor3() (Aig method), 32

L

level() (DepthAig method), 34

M

make_signal() (Aig method), 30
module
 aigverse, 23
 aigverse.adapters, 23
 aigverse.adapters.networkx, 24
 aigverse.algorithms, 24
 aigverse.generators, 26
 aigverse.io, 27

aigverse.networks, 28
aigverse.utils, 40
multiplexer() (in module aigverse.generators), 27

N

NamedAig (class in aigverse.networks), 33
node_to_index() (Aig method), 30
nodes() (Aig method), 31
ntk (AigEdgeList property), 38
num_bits() (TruthTable method), 41
num_blocks() (TruthTable method), 40
num_cis (SequentialAig property), 36
num_cos (SequentialAig property), 36
num_gates (Aig property), 29
num_gates (AigIndexList property), 39
num_levels (DepthAig property), 34
num_pis (Aig property), 30
num_pis (AigIndexList property), 39
num_pis (SequentialAig property), 36
num_pos (Aig property), 30
num_pos (AigIndexList property), 39
num_pos (SequentialAig property), 36
num_registers (SequentialAig property), 36
num_vars() (TruthTable method), 40

P

pi_at() (Aig method), 30
pi_at() (SequentialAig method), 36
pi_index() (Aig method), 30
pi_index() (SequentialAig method), 36
pis() (Aig method), 31
pis() (SequentialAig method), 37
po_at() (Aig method), 30
po_at() (SequentialAig method), 36
po_index() (Aig method), 30
pos() (Aig method), 31
pos() (AigIndexList method), 39
pos() (SequentialAig method), 37

R

random_aig() (in module aigverse.generators), 26
raw() (AigIndexList method), 38
read_aiger_into_aig() (in module aigverse.io), 27
read_aiger_into_sequential_aig() (in module aigverse.io),
 28
read_ascii_aiger_into_aig() (in module aigverse.io), 27
read_ascii_aiger_into_sequential_aig() (in module
 aigverse.io), 28
read_pla_into_aig() (in module aigverse.io), 28
read_verilog_into_aig() (in module aigverse.io), 28
register_at() (SequentialAig method), 36
registers() (SequentialAig method), 37
ri_at() (SequentialAig method), 36
ri_index() (SequentialAig method), 37
ri_to_ro() (SequentialAig method), 37
ripple_carry_adder() (in module aigverse.generators), 27
ripple_carry_multiplier() (in module aigverse.generators),
 27
ris() (SequentialAig method), 37
ro_at() (SequentialAig method), 36
ro_index() (SequentialAig method), 37
ro_to_ri() (SequentialAig method), 37
ros() (SequentialAig method), 37

S

SequentialAig (class in aigverse.networks), 35
set_bit() (TruthTable method), 41
set_name() (NamedAig method), 34
set_network_name() (NamedAig method), 34
set_output_name() (NamedAig method), 34
set_register() (SequentialAig method), 36
sideways_sum_adder() (in module aigverse.generators), 27
simulate() (in module aigverse.algorithms), 26
simulate_nodes() (in module aigverse.algorithms), 26

size (*Aig* property), 29
size (*AigIndexList* property), 38
sop_refactoring() (in module *aigverse.algorithms*), 25
source (*AigEdge* property), 37

T

target (*AigEdge* property), 37
ternary_majority() (in module *aigverse.utils*), 42
to_aig() (*AigIndexList* method), 39
to_binary() (*TruthTable* method), 42
to_edge_list() (*Aig* method), 32
to_edge_list() (*SequentialAig* method), 37
to_hex() (*TruthTable* method), 42
to_index_list() (*Aig* method), 32
to_index_list() (*SequentialAig* method), 37
to_networkx() (*Aig* method), 32
to_networkx() (in module *aigverse.adapters.networkx*), 24
TruthTable (class in *aigverse.utils*), 40
type (*AigRegister* property), 35

U

update_levels() (*DepthAig* method), 35

W

weight (*AigEdge* property), 37
write_aiger() (in module *aigverse.io*), 28
write_dot() (in module *aigverse.io*), 28
write_verilog() (in module *aigverse.io*), 28