Contents

1 Working with Boolean Functions

Boolean functions are the building blocks of Boolean network models used to represent gene regulatory networks, signaling pathways, and other biological control systems. Understanding how to create and analyze individual Boolean functions is essential before studying network-level dynamics.

In this tutorial, we explore the `BooleanFunction` class — the foundation of BoolForge. Boolean functions form the regulatory rules in Boolean network models of gene regulation, so understanding their structure is essential before studying networks.

1.1 What you will learn

In this tutorial you will:

- create Boolean functions from truth tables and from textual expressions,
- inspect core attributes such as degree, variable names, and stored properties,
- compute basic structural properties (essential variables, Hamming weight, bias),
- convert Boolean functions into logical and polynomial representations,
- and interface with CANA objects.

1.2 Setup

```
import boolforge
```

1.3 Create a Boolean function

Boolean functions can be described in logical form, as polynomials, or as truth tables. BoolForge treats Boolean functions as binary vectors of length 2^n, where n is the number of inputs. The vectors describe the *right side* of the truth table. The left side of the truth table is not stored because it is the same for any function with n inputs. For example, the function

$$f(A, B) = A \wedge B$$

is stored as `[0, 0, 0, 1]`, corresponding to:

A	B	f(A,B)
0	0	0
0	1	0
1	0	0
1	1	1

1.3.1 Create Boolean functions from a truth table

An instance of `BooleanFunction` can be generated by specifying the right side of the truth table, i.e., by providing a binary vector of length 2^n for any $n \geq 0$. For example, to create the AND function above, we can write

```
f = boolforge.BooleanFunction([0, 0, 0, 1], name="f_AND") #name is optional
print("f:", f)
print("Truth table of f:\n", f.to_truth_table().to_string())
```

```
f: [0 0 0 1]
Truth table of f:
    x0  x1  f_AND
0   0   0       0
1   0   1       0
```

```
2   1   0      0
3   1   1      1
```

Any Boolean function is stored as right side of the truth table. That is, the outputs are ordered by the binary representation of inputs:

- Position 0 → (A,B) = (0,0)
- Position 1 → (A,B) = (0,1)

- Position 2 → (A,B) = (1,0)
- Position 3 → (A,B) = (1,1)

1.3.2 Create Boolean functions from text

Boolean functions can also be created from textual expressions. For example, to define the same function as f, we can write

```
f2 = boolforge.BooleanFunction("A and B")
print("f2:", f2)
```

```
f2: [0 0 0 1]
```

The text processor is fairly versatile. For example, we can define the same function as f also by writing

```
f3 = boolforge.BooleanFunction("A + B > 1")
print("f3:", f3)
```

```
f3: [0 0 0 0]
```

Some examples of more complicated functions include:

```
g = boolforge.BooleanFunction("(A AND B) OR (NOT A AND C)")
h = boolforge.BooleanFunction("(x + y + z) % 2 == 0")
k = boolforge.BooleanFunction("(-1) * x + y + z > 0")

labels = ["g", "h", "k"]
boolforge.display_truth_table(g, h, k, labels=labels)
```

```
x1  x2  x3  |   g   h   k
-----------------------------------------------
0   0   0   |   0   1   0
0   0   1   |   1   0   1
0   1   0   |   0   0   1
0   1   1   |   1   0   1
1   0   0   |   0   0   0
1   0   1   |   0   0   0
1   1   0   |   1   0   0
1   1   1   |   1   0   1
```

1.3.3 Combining BooleanFunction objects

New Boolean functions can be constructed by combining existing ones using Boolean algebra operations. This is useful when building larger rules from simpler components.

Supported operations include:

- ~ NOT
- & AND
- | OR
- ^ XOR

```
a = boolforge.BooleanFunction("X + Y == 1")
b = boolforge.BooleanFunction("X OR Y")

not_a = ~a
a_and_b = a & b
a_or_b = a | b
a_xor_b = a ^ b

labels = ["a", "b", "NOT a", "a AND b", "a OR b", "a XOR b"]
boolforge.display_truth_table(a, b, not_a, a_and_b, a_or_b, a_xor_b, labels=labels)
```

```
x1  x2  |   a   b   NOT a   a AND b a OR b  a XOR b
------------------------------------------------------------------------
0   0   |   0   0   1   0   0   0
0   1   |   1   1   0   1   1   0
1   0   |   1   1   0   1   1   0
1   1   |   1   1   0   1   1   0
```

1.4 Attributes of BooleanFunction

Each `BooleanFunction` instance has the following attributes:

attribute	type	description
`f`	`np.ndarray`	truth table (right side)
`n`	`int`	number of variables
`variables`	`np.ndarray`	variable names
`name`	`str`	optional name
`properties`	`dict`	cached properties

```
print("f.f:", f.f)
print("f.n:", f.n)
print("f.variables:", f.variables)
print("f.name:", f.name)
print("f.properties:", f.properties)
```

```
f.f: [0 0 0 1]
f.n: 2
f.variables: ['x0' 'x1']
f.name: f_AND
f.properties: {}
```

When a function is created from a truth table, variable names default to `x0, x1, ...`. When created from text, variable names are inferred.

```
print("f2.variables:", f2.variables)
print("f3.variables:", f3.variables)
print("g.variables:", g.variables)
print("h.variables:", h.variables)
```

```
f2.variables: ['A' 'B']
f3.variables: ['A' 'B']
g.variables: ['A' 'B' 'C']
h.variables: ['x' 'y' 'z']
```

The variable order is determined by first occurrence in the expression. See e.g.,

```
print(boolforge.BooleanFunction("(x + y + z) % 2 == 0").variables)
print(boolforge.BooleanFunction("(y + z + x) % 2 == 0").variables)
```

```
['x' 'y' 'z']
['y' 'z' 'x']
```

The variable order determines how the truth table is indexed. For example, for variables [x,y,z], the entry in position i corresponds to the binary expansion of i over (x,y,z). Therefore, the same expression with a different variable order results in a different right-side truth table ordering. This becomes important when combining functions inside networks or importing networks from text files.

1.5 Basic properties of Boolean functions

We can inspect various properties of a Boolean function. The degree, i.e., the number of inputs, is readily available via 'f.n'. Other properties can be computed.

- '.is_constant()' checks if the function is constant,
- '.is_degenerate()' checks if the function contains non-essential variables,
- '.get_essential_variables()' provides the indices (Python: starting at 0!) of the essential variables,
- '.get_type_of_inputs()' describes the type of each input ('positive', 'negative', 'conditional', or 'non-essential').
- The Hamming weight is the number of 1s in the right side of the truth table.
- The absolute bias is $|\#\text{ones} - \#\text{zeros}|/2^n$. It equals 1 for constant functions and 0 for unbiased functions.

```
print("Number of variables:", f.n)
print("Is constant?", f.is_constant())
print("Is degenerate?", f.is_degenerate())
print("Essential variables:", f.get_essential_variables())
print("Type of inputs:", f.get_type_of_inputs())
print("Hamming weight:", f.get_hamming_weight())
print("Absolute bias:", f.get_absolute_bias())
```

```
Number of variables: 2
Is constant? False


Is degenerate? False
Essential variables: [0, 1]
Type of inputs: ['positive' 'positive']
Hamming weight: 1
Absolute bias: 0.5
```

Repeating this for `g` illustrates how properties differ.

```
print("Number of variables:", g.n)
print("Is constant?", g.is_constant())
print("Is degenerate?", g.is_degenerate())
print("Essential variables:", g.get_essential_variables())
print("Type of inputs:", g.get_type_of_inputs())
print("Hamming weight:", g.get_hamming_weight())
print("Absolute bias:", g.get_absolute_bias())
```

```
Number of variables: 3
Is constant? False
Is degenerate? False
Essential variables: [0, 1, 2]
```

```
Type of inputs: ['conditional' 'positive' 'positive']
Hamming weight: 4
Absolute bias: 0.0
```

The `.summary()` method prints a human-readable overview of basic properties. If more advanced properties have already been computed, e.g., by `.get_layer_structure()` or `get_type_of_inputs()`, they are also displayed (or if the optional keyword `COMPUTE_ALL` is set to True, default False).

```
f = boolforge.BooleanFunction("(A and B) OR NOT C")
print(f.summary())
print()

# Trigger computation of more advanced properties
print(f.summary(compute_all=True))
```

```
BooleanFunction summary
----------------------------------------
Number of variables:        3
Hamming Weight:             5
Bias:                       0.625
Absolute bias:              0.250
Variables:                  ['A' 'B' 'C']

BooleanFunction summary
----------------------------------------
Number of variables:        3
Hamming Weight:             5
Bias:                       0.625
Absolute bias:              0.250
Variables:                  ['A' 'B' 'C']
CanalizingDepth:            3
NumberOfLayers:             2
CanalizingInputs:           [0 0 0]
CanalizedOutputs:           [1 0 0]
CoreFunction:               [1]
OrderOfCanalizingVariables: [2 0 1]
LayerStructure:             [1, 2]
InputTypes:                 ['negative' 'positive' 'positive']
```

1.6 Logical and polynomial representations

While Boolean functions are stored as truth tables, they can be expressed in logical and polynomial format.

```
print(f"Logical form of {f.name}:", f.to_logical(and_op=" ∧ ", or_op=" ∨ ", not_op=" ¬"))
print(f"Polynomial form of {f.name}:", f.to_polynomial())
```

```
Logical form of : (( ¬C)) ∨ (A ∧ B)
Polynomial form of : (1 - A) * (1 - B) * (1 - C) + (1 - A) * B * (1 - C) + A * (1 - B) * (1 - C) + A * B * (1 - C) + A * B * C
```

In addition, an instance of `BooleanFunction` can be turned into an instance of `BooleanNode` from the CANA package. This requires the optional CANA package to be installed.

```
cana_object = f.to_cana()
print(type(cana_object))
```

```
<class 'cana.boolean_node.BooleanNode'>
```

1.7 Summary of Key Concepts

Before moving on to more advanced topics, here is a short summary of the fundamental ideas introduced in this tutorial:

1.7.1 Boolean functions

A Boolean function maps a set of binary inputs (0/1) to a single binary output. BoolForge represents Boolean functions internally by their truth table, i.e., the list of outputs in lexicographic order of the input combinations.

1.7.2 Representations of Boolean functions

Boolean functions can be created from:

- a truth table (a sequence of 0s and 1s of length 2^n for some n),
- a logical expression written in Python syntax,
- algebraic combinations of existing BooleanFunction objects using operations such as `+` (OR), $*$ (AND), `^` (XOR), and other supported Boolean operations.

Each representation produces an equivalent internal truth-table-based object.

1.7.3 Variable names and ordering

BoolForge automatically infers variable names from the order of first appearance in expressions.
This order determines the indexing of the truth table and therefore affects how the function interacts with larger Boolean networks.

1.7.4 Basic properties of Boolean functions

BoolForge can compute structural properties, including:

- the number of variables (`n`),
- the Hamming weight (number of 1s in the truth table),
- absolute bias (imbalance between 0s and 1s),
- essential and non-essential variables,
- positive/negative influence of each input.

These properties help characterize the function's behavior and are used throughout later tutorials.

1.7.5 Conversions and interoperability

BoolForge supports conversion between representations (truth table, polynomial, and logical form) and is compatible with external packages such as **CANA** for advanced analysis.
This makes it easy to move between analytical frameworks and reuse models.

Together, these concepts provide the foundation for understanding canalization, random Boolean function generation, and eventually the construction and analysis of full Boolean networks.

1.8 Frequently Asked Questions

1.8.1 Why does the order of variables matter?

The order in which variables appear determines the ordering of the truth table. For a function with variables `[A, B, C]`, the entry at position $i \in \{0, 1, \ldots, 2^n - 1$ corresponds to the binary representation of i over `(A, B, C)`. For example, row 4 (i.e., the fifth row since Python starts indexing at 0) corresponds to $A = 1, B = 0, C = 0$.

If two equivalent expressions list variables in different orders, their truth tables will be indexed differently. See, for example,

```
print(boolforge.BooleanFunction('A and not B'))
print(boolforge.BooleanFunction('not B and A'))
```

```
[0 0 1 0]
[0 1 0 0]
```

To ensure reproducibility, always use consistent variable names and ordering.

1.8.2 How do I choose between defining a function via a truth table or via an expression?

Short answer: It does not matter. Both methods produce identical internal representations.

Slightly longer answer: Use a **textual expression** if:

- you know the natural logical description of your function (e.g., `A and B`),
- the function is part of a Boolean network stored in some text file.

Use a **truth table** if:

- you generated the table programmatically (e.g., using `boolforge.random_function`).

1.8.3 What is the difference between `get_type_of_inputs()` and monotonicity?

The method `get_type_of_inputs()` classifies each input variable individually according to how it influences the output:

- positively increasing,
- negatively increasing,
- conditional,
- or non-essential.

Monotonicity, by contrast, is a **global property** of the Boolean function. A function is monotone if **all** essential variables influence the output in a consistent direction.

A function can therefore be non-monotone even if individual inputs have a well-defined influence type.

1.8.4 Quick Reference

Task	Example
Create from truth table	`BooleanFunction([0, 0, 0, 1])`
Create from expression	`BooleanFunction("A and B")`
Combine with operations	`f & g, f | g, ~f, f ^ g`
Check properties	`f.n`, `f.is_constant()`, `f.is_degenerate()`
Get variable names	`f.variables`
Convert representations	`f.to_logical()`, `f.to_polynomial()`

2 Advanced Concepts for Boolean Functions

Understanding the structure of a Boolean function is essential for analyzing the behavior of the Boolean networks they define. In this tutorial, we move beyond the basics of `BooleanFunction` and explore three core concepts:

- **Symmetries** among inputs
- **Activities** of inputs
- **Average sensitivity** of a Boolean function

These quantities are tied to redundancy, robustness, and dynamical behavior – concepts that will play a central role in later tutorials on canalization and network dynamics.

2.1 What you will learn

In this tutorial you will learn how to:

- identify symmetry groups of Boolean functions,
- compute activities and sensitivities,
- choose between exact and Monte Carlo computation,
- interpret these quantities in terms of robustness and redundancy.

2.2 Setup

```
import boolforge
import numpy as np
```

2.3 Symmetries in Boolean Functions

In gene regulation, symmetric variables might represent redundant transcription factor binding sites or functionally equivalent repressors. Identifying symmetries can:

- Reduce model complexity
- Suggest evolutionary mechanisms (gene duplication)
- Identify potential drug targets (symmetric inputs may compensate)

A symmetry of a Boolean function is a permutation of input variables that does **not** change its output.

- Inputs in the same symmetry group can be swapped freely.
- Inputs in different groups cannot.

The following three Boolean functions exhibit full, partial, and no symmetry.

```
# Fully symmetric (parity / XOR)
f = boolforge.BooleanFunction("(x0 + x1 + x2) % 2")

# Partially symmetric
g = boolforge.BooleanFunction("x0 | (x1 & x2)")

# No symmetry
h = boolforge.BooleanFunction("x0 | (x1 & ~x2)")

labels = ["f", "g", "h"]
boolforge.display_truth_table(f, g, h, labels=labels)
```

```
x0  x1  x2  |   f   g   h
-----------------------------------------------
0   0   0   |   0   0   0
0   0   1   |   1   0   0
```

```
0   1   0   |   1   0   1
0   1   1   |   1   1   0
1   0   0   |   1   1   1
1   0   1   |   1   1   1
1   1   0   |   1   1   1
1   1   1   |   1   1   1
```

```
for func, label in zip([f, g, h], labels):
    print(f"Symmetry groups of {label}:")
    for group in func.get_symmetry_groups():
        print("  ", func.variables[np.array(group)])
    print()
```

```
Symmetry groups of f:
   ['x0' 'x1' 'x2']

Symmetry groups of g:
   ['x0']
   ['x1' 'x2']

Symmetry groups of h:
   ['x0']
   ['x1']
   ['x2']
```

Interpretation

- `f` is fully symmetric: all variables are interchangeable.
- `g` has partial symmetry: `x1` and `x2` are equivalent but `x0` is distinct.
- `h` has no symmetries: all inputs play unique roles.

These patterns foreshadow the concepts of canalization, and specifically canalizing layers, explored in later tutorials.

2.4 Degenerate functions

A function is **degenerate** if one or more inputs do not matter at all.

```
print("f.is_degenerate()", f.is_degenerate())
k = boolforge.BooleanFunction("(x AND y) OR x")
print("k.is_degenerate()", k.is_degenerate())
```

```
f.is_degenerate() False
k.is_degenerate() True
```

Detecting degeneracy is NP-hard in general. However even at relatively low degree, such functions are extremely rare unless intentionally created.

BoolForge therefore:

- allows degenerate functions by default,
- avoids expensive essential-variable checks unless requested.

2.5 Activities and Sensitivities

Activities and sensitivity quantify how much each input affects the output of a Boolean function.

2.5.1 Activity

The activity of input x_i is the probability that flipping x_i changes the function's output:

$$a(f, x_i) = \Pr[f(\mathbf{x}) \neq f(\mathbf{x} \oplus e_i)],$$

where $e_i = (0, \ldots, 0, 1, 0, \ldots, 0)$ is the ith unit vector.

- If $a = 1$: the variable always matters.
- If $a = 0$: the variable is irrelevant (degenerate).
- In large random Boolean functions, $a \approx 0.5$ for all variables.

2.5.2 Average sensitivity

The *average sensitivity* of a Boolean function describes how sensitive its output is to changes in its inputs, specifically to a random single-bit flip. The (unnormalized) average sensitivity is the sum of all its activities:

$$S(f) = \sum_i a(f, x_i).$$

Division by n yields the *normalized average sensitivity* $s(f)$, which can be readily compared between functions of different degree n:

$$s(f) = \frac{S(f)}{n}.$$

Interpretation

In Boolean network theory, the mean normalized average sensitivity $s(f)$ determines how perturbations tend to propagate through the system.

- If $s(f) < 1$, perturbations tend to die out (*ordered regime*).
- If $s(f) > 1$, perturbations typically amplify (*chaotic regime*).
- The boundary $s(f) = 1$ defines the *critical regime*.

The critical regime is believed to characterize many biological networks (see later tutorials). It represents a balance between order and chaos. Operating at this "edge of chaos" may optimize information processing and evolvability.

2.5.3 Exact vs Monte Carlo computation

- Exact (`exact=True`) computation enumerates all 2^n states; feasible for small n.
- Monte Carlo (`exact=False`, default) simulation approximates using random samples; scalable to large n.

Computational cost guide:

- Exact methods: $O(2^n)$ time and space, where n = number of inputs.
- Monte Carlo: $O(k)$ time, where k = number of samples.

Recommendation:

- $n \leq 10$: Use exact methods (fast, deterministic)
- $10 < n \leq 20$: Use exact if possible, Monte Carlo if repeated computation needed
- n > 20: Use Monte Carlo (exact is infeasible)

2.5.4 Computing activities and sensitivities

To investigate how to compute the activities and the average sensitivity in `BoolForge`, we work with the linear function `f` from above, as well as with the function `g`.

```
exact = True
normalized = True

print("Activities of f:", f.get_activities(exact=exact))
print("Activities of g:", g.get_activities(exact=exact))

print("Normalized average sensitivity of f:", f.get_average_sensitivity(exact=exact, normalized=normalized))
print("Normalized average sensitivity of g:", g.get_average_sensitivity(exact=exact, normalized=normalized))
```

```
Activities of f: [0.25 0.25 0.25]
Activities of g: [0.75 0.25 0.25]
Normalized average sensitivity of f: 0.25
Normalized average sensitivity of g: 0.4166666666666667
```

Interpretation

- For `f` (XOR), flipping any input always flips the output, so $s(f) = 1$.
- For `g`, x_0 influences the output more often than x_1 or x_2. 75% of x_0 flips and 25% of x_1 or x_2 flips change the output of `g`. Thus, the normalized average sensitivity of `g` is $\frac{1}{3} * 75\% + \frac{2}{3} 25\% = \frac{5}{12}$.

This unequal influence is a precursor to canalization, a property investigated in depth in the next tutorial.

Exact computation is infeasible for large n, so Monte Carlo simulation must be used.

When generating such a large function randomly (see Tutorial 4) it is not recommended to require that all inputs are essential, as (i) this is almost certainly the case anyways (the probability that an n-input function does not depend on input x_i is given $1/2^{n-1}$), and (ii) checking for input degeneracy is NP-hard (i.e., very computationally expensive). We thus suggest setting `allow_degenerate_functions=True`. You find more on this and the `random_function` method in Tutorial 4.

```
exact = False
n = 25

h = boolforge.random_function(n=n, allow_degenerate_functions=True)

activities = h.get_activities(exact=exact)
print(f"Mean activity: {np.mean(activities):.4f}")
print(
    f"Normalized average sensitivity: "
    f"{h.get_average_sensitivity(exact=exact):.4f}"
)
```

```
Mean activity: 0.5008
Normalized average sensitivity: 0.5001
```

Interpretation

Random Boolean functions satisfy:

- mean activity ≈ 0.5,
- normalized average sensitivity ≈ 0.5.

Thus, the results for `h` align with known theoretical results. More generally, random Boolean function results define the typical behavior against which biological functions can be compared (see Tutorial 5).

2.6 Summary

In this tutorial you learned:

- how to compute symmetry groups,
- how to test for input degeneracy,
- how to compute activities and sensitivities,
- how these quantities relate to robustness and structure.

These concepts provide essential foundations for understanding

- canalization, the core concept of Tutorial 3,
- and the robustness of Boolean networks, explored in Tutorial 8.

3 Canalization

Canalization is a key property of biological Boolean functions that confers robustness: when a canalizing variable takes its canalizing value, the output is determined regardless of other inputs. This "buffering" mechanism is thought to protect organisms from genetic and environmental perturbations.

Discovered by C.H. Waddington in 1942 in developmental biology, canalization has since been formalized in Boolean network theory and found to be prevalent in empirically-derived gene regulatory networks.

3.1 What you will learn

In this tutorial you will:

- determine if a Boolean function is canalizing, k-canalizing, and nested canalizing,
- compute the canalizing layer structure of any Boolean function,
- compute properties related to collective canalization, such as canalizing strength, effective degree and input redundancy.

3.2 Setup

```
import boolforge
import matplotlib.pyplot as plt
```

3.3 Canalizing variables and layers

A Boolean function $f(x_1, \dots, x_n)$ is *canalizing* if there exists at least one *canalizing variable* x_i and a *canalizing input value* $a \in \{0, 1\}$ such that

$$f(x_1, \dots, x_i = a, \dots, x_n) = b,$$

where $b \in \{0, 1\}$ is a constant, the *canalized output*.

A Boolean function is *k-canalizing* if it has at least k conditionally canalizing variables. This is checked recursively: after fixing a canalizing variable x_i to its non-canalizing input value $\bar{a}$, the subfunction $f(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$ must itself contain another canalizing variable, and so on. For a given function, the maximal possible value of k is defined as its *canalizing depth*. If all variables are conditionally canalizing (i.e., if the canalizing depth is n), the function is called a *nested canalizing* function (*NCF*). Biological networks are heavily enriched for NCFs as we explore in a later tutorial.

Per (He and Macauley, Physica D, 2016), any Boolean function can be decomposed into a unique standard monomial form by recursively identifying and removing all conditionally canalizing variables (this set of variables is called a *canalizing layer*). Each variable of a Boolean function appears in exactly one layer, or (if it is not conditionally canalizing) it is part of the non-canalizing core function that has to be evaluated only if all conditionally canalizing variables receive their non-canalizing input value. The *canalizing layer structure* $[k_1, \dots, k_r]$ describes the number of variables in each canalizing layer. We thus have $r \geq 0$, $k_i \geq 1$ and $k_1 + \cdots + k_r$.

In the following code, we define four 3-input functions with different canalizing properties.

```
# Non-canalizing XOR function
f = boolforge.BooleanFunction("(x0 + x1 + x2) % 2")

# 1-canalizing function
g = boolforge.BooleanFunction("(x0 | (x1 & x2 | ~x1 & ~x2)) % 2")

# Nested canalizing function with all variables in one layer
h = boolforge.BooleanFunction("~x0 & x1 & x2")
```

```
# Nested canalizing function with two canalizing layers
k = boolforge.BooleanFunction("x0 | (x1 & x2)")

labels = ["f", "g", "h", "k"]
boolforge.display_truth_table(f, g, h, k, labels=labels)
```

```
x0  x1  x2  |   f   g   h   k
-------------------------------------------------------
0   0   0   |   0   1   0   0
0   0   1   |   1   0   0   0
0   1   0   |   1   0   0   0
0   1   1   |   1   1   1   1
1   0   0   |   1   1   0   1
1   0   1   |   1   1   0   1
1   1   0   |   1   1   0   1
1   1   1   |   1   1   0   1
```

3.3.1 Canalizing depth and nested canalization

For each function, we can determine whether it is canalizing and/or nested canalizing. This is determined by the canalizing depth (the number of conditionally canalizing variables), which we can also directly compute. As a reminder, an n-input function is canalizing if its canalizing depth is non-zero and nested canalizing if its canalizing depth equals n.

```
for func, label in zip([f, g, h, k], labels):
    depth = func.get_canalizing_depth()
    print(f"Canalizing depth of {label}: {depth}")

    print(f"{label} is canalizing:", func.is_canalizing())
    print(f"{label} is nested canalizing:", func.is_k_canalizing(k=func.n))
    print()
```

```
Canalizing depth of f: 3
f is canalizing: True
f is nested canalizing: True

Canalizing depth of g: 1
g is canalizing: True
g is nested canalizing: False

Canalizing depth of h: 3
h is canalizing: True
h is nested canalizing: True

Canalizing depth of k: 3
k is canalizing: True
k is nested canalizing: True
```

3.3.2 Canalizing layer structure

The full canalizing layer structure includes canalizing input values, canalized output values, the order of canalizing variables, the layer structure, and the remaining non-canalizing core function.

```
for func, label in zip([f, g, h, k], labels):
    info = func.get_layer_structure()
    print(f"Canalizing input values of {label}: {info['CanalizingInputs']}")
```

```
    print(f"Canalized output values of {label}: {info['CanalizedOutputs']}")
    print(f"Order of canalizing variables of {label}: {info['OrderOfCanalizingVariables']}")
    print(f"Layer structure of {label}: {info['LayerStructure']}")
    print(f"Number of layers of {label}: {info['NumberOfLayers']}")
    print(f"Core function of {label}: {info['CoreFunction']}")
    print()
```

```
Canalizing input values of f: [1 1 1]
Canalized output values of f: [1 1 1]
Order of canalizing variables of f: [0 1 2]
Layer structure of f: [3]
Number of layers of f: 1
Core function of f: [0]

Canalizing input values of g: [1]
Canalized output values of g: [1]
Order of canalizing variables of g: [0]
Layer structure of g: [1]
Number of layers of g: 1
Core function of g: [1 0 0 1]

Canalizing input values of h: [1 0 0]
Canalized output values of h: [0 0 0]
Order of canalizing variables of h: [0 1 2]
Layer structure of h: [3]
Number of layers of h: 1
Core function of h: [1]

Canalizing input values of k: [1 0 0]
Canalized output values of k: [1 0 0]
Order of canalizing variables of k: [0 1 2]
Layer structure of k: [1, 2]
Number of layers of k: 2
Core function of k: [1]
```

Consider, for example, the output for `k`. The canalizing input values corresponding to x_0, x_1, x_2 are $1, 0, 0$, respectively, with the same canalized outputs. That is,

- Layer 1: x_0 (if $x_0 = 1$, then $k = 1$, regardless of x_1 and x_2)
- Layer 2: x_1, x_2 (if $x_0 = 0$ and $x_1 = 0$ or $x_2 = 0$, then $k = 0$)

3.4 Collective canalization

Collective canalization treats canalization as a property of the function rather than individual variables (Reichhardt & Bassler, J. Phys. A, 2007). Individual canalization asks: "Which *single* variables can determine output?" Collective canalization asks: "Which *sets* of variables can determine output?"

A Boolean function is *k-set canalizing* if there exists a set of k variables whose fixed values determine the output irrespective of the remaining inputs.

Consider, for example, the function $k(x_0, x_1, x_2) = x_0 \vee (x_1 \wedge x_2)$. This function is 2-set canalizing because

- $\{x_0, x_1\}$ can determine the output: if $(x_0, x_1) = (1, 0)$, $k = 1$ (x_2 irrelevant), or
- $\{x_1, x_2\}$ can determine the output: if $(x_1, x_2) = (1, 1)$, $k = 1$ (x_0 irrelevant)

The proportion of such k-sets, the k-set canalizing proportion denoted $P_k(f)$, is used to define the canalizing strength. It is fairly obvious that

- nested canalizing functions of a single layer such as `h` are the non-degenerate functions with highest k-set canalizing proportion $P_k(f) = 1 - 1/2^k$, and
- $P_{k-1}(f) \leq P_k(f)$, i.e., more knowledge about a function's inputs cannot result in less knowledge about its output,
- the $n-1$-set canalizing proportion $P_{n-1}(f)$ is 1 minus the function's normalized average sensitivity.

We can compute the k-set canalizing proportions for the four 3-input functions:

```
for func, label in zip([f, g, h, k], labels):
    print(f"1-set canalizing proportion of {label}: {func.get_kset_canalizing_proportion(k=1)}")
    print(f"2-set canalizing proportion of {label}: {func.get_kset_canalizing_proportion(k=2)}")
    print(f"Normalized average sensitivity of {label}: {func.get_average_sensitivity(exact=True, normalized=True)}
    print(f"3-set canalizing proportion of {label}: {func.get_kset_canalizing_proportion(k=3)}")
    print()
```

```
1-set canalizing proportion of f: 0.5
2-set canalizing proportion of f: 0.75
Normalized average sensitivity of f: 0.25
3-set canalizing proportion of f: 1.0

1-set canalizing proportion of g: 0.16666666666666666
2-set canalizing proportion of g: 0.5
Normalized average sensitivity of g: 0.5
3-set canalizing proportion of g: 1.0

1-set canalizing proportion of h: 0.5
2-set canalizing proportion of h: 0.75
Normalized average sensitivity of h: 0.25
3-set canalizing proportion of h: 1.0

1-set canalizing proportion of k: 0.16666666666666666
2-set canalizing proportion of k: 0.5833333333333334
Normalized average sensitivity of k: 0.4166666666666667
3-set canalizing proportion of k: 1.0
```

3.4.1 Canalizing strength

The *canalizing strength* summarizes collective canalization as a weighted average of the k-set canalizing proportions (Kadelka et al., Adv Appl Math, 2023). It ranges from:

- 1 for maximally canalizing non-degenerate functions (namely, nested canalizing functions of a single canalizing layer such as `h`),
- 0 for linear functions such as `f`,

For all other non-degenerate Boolean functions it is within $(0, 1)$.

It helps to consider the canalizing strength as a probability: Given that I know a random number of function inputs (drawn uniformly at random from $1, \dots, n-1$), how likely am I to already know the function output?

```
for func, label in zip([f, g, h, k], labels):
    strength = func.get_canalizing_strength()
    print(f"Canalizing strength of {label}: {strength}")
    print()
```

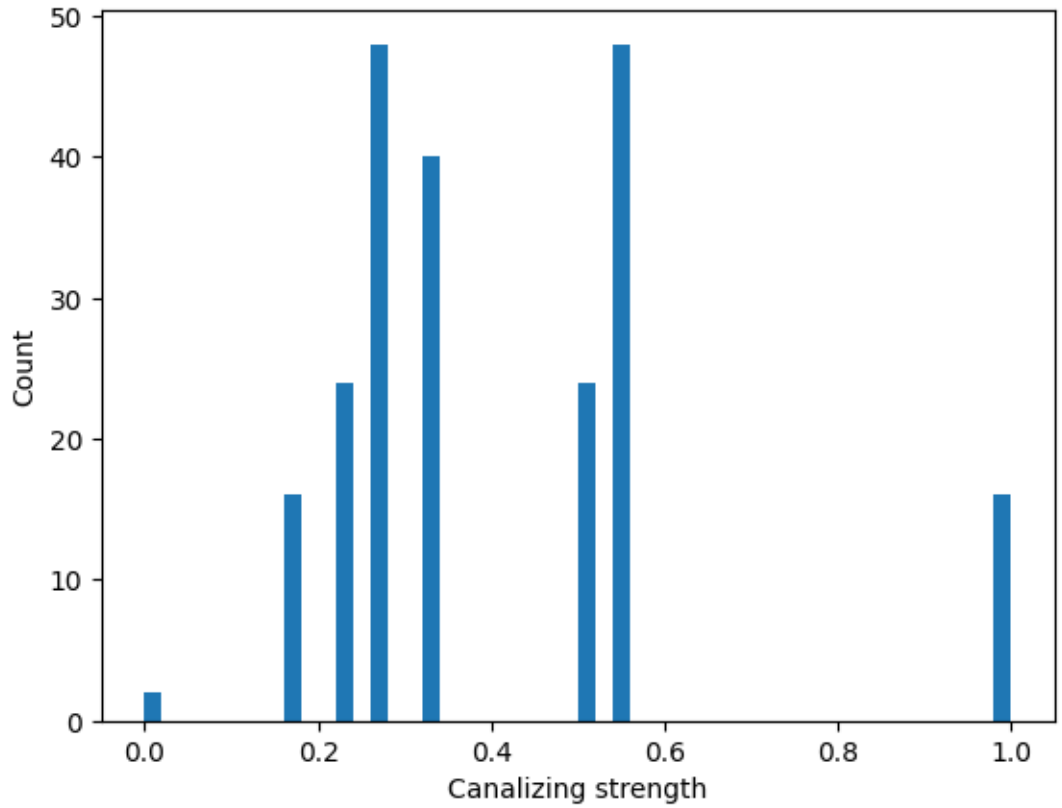
```
Canalizing strength of f: 1.0

Canalizing strength of g: 0.5
```

```
Canalizing strength of h: 1.0

Canalizing strength of k: 0.5555555555555556
```

3.4.2 Distribution of canalizing strength

An enumeration of all non-degenerate 3-input Boolean functions reveals the distribution of the canalizing strength. Note that this brute-force code can also run (in less than a minute) for all $2^{2^4} = 2^{16} = 65,536$ 4-input functions but will take days for all $2^{2^5} = 2^{32} = 4,294,967,296$ 5-input functions.

```
n = 3
all_functions = boolforge.get_left_side_of_truth_table(2**n)

canalizing_strengths = []
for binary_vector in all_functions:
    func = boolforge.BooleanFunction(f=binary_vector)
    if not func.is_degenerate():
        canalizing_strengths.append(func.get_canalizing_strength())

fig, ax = plt.subplots()
ax.hist(canalizing_strengths, bins=50)
ax.set_xlabel("Canalizing strength")
ax.set_ylabel("Count")
plt.show()
```



3.5 Canalization as a measure of input redundancy

Canalization, symmetry and redundancy are related concepts. A highly symmetry Boolean function with few (e.g., one) symmetry groups exhibits high input redundancy and is on average more canalizing, irrespective of the measure of canalization. Recently, it was shown that almost all Boolean functions (except the linear functions) exhibit some level of *input redundancy* (Gates et al., PNAS, 2021). The input redundancy of a variable is defined as 1 minus its *edge effectiveness*, which describes the proportion of times that this variable is needed to determine the output of the function. Edge effectiveness is very similar to the activity of a variable but is not the same (the difference is defined as *excess canalization*). The sum of all edge effectiveness values of the inputs of a function is known as its *effective degree*. The average input redundancy serves as a measure of the canalization in a function.

`BoolForge` can compute all these quantities. To use this functionality, the optional `CANA` package must be installed (`pip install cana` or `pip install boolforge[cana]`). To exemplify this, reconsider the four 3-input functions from above.

```
for func, label in zip([f, g, h, k], labels):
    edge_eff = func.get_edge_effectiveness()
    activities = func.get_activities()
    effective_degree = func.get_effective_degree()
    input_redundancy = func.get_input_redundancy()

    print(f"Edge effectiveness of {label}: {edge_eff}")
    print(f"Activities of {label}: {activities}")
    print(f"Excess canalization of {label}: {edge_eff - activities}")
    print(f"Effective degree of {label}: {effective_degree}")
    print(f"Average edge effectiveness of {label}: {effective_degree / func.n}")
    print(f"Normalized input redundancy of {label}: {input_redundancy}")
    print()
```

```
Edge effectiveness of f: [0.41666666666666663, 0.41666666666666663, 0.41666666666666663]
Activities of f: [0.2504 0.2444 0.2515]
Excess canalization of f: [0.16626667 0.17226667 0.16516667]
Effective degree of f: 1.25
Average edge effectiveness of f: 0.4166666666666667
Normalized input redundancy of f: 0.5833333333333334

Edge effectiveness of g: [0.625, 0.625, 0.625]
Activities of g: [0.5015 0.4986 0.4986]
Excess canalization of g: [0.1235 0.1264 0.1264]
Effective degree of g: 1.875
Average edge effectiveness of g: 0.625
Normalized input redundancy of g: 0.375

Edge effectiveness of h: [0.41666666666666663, 0.41666666666666663, 0.41666666666666663]
Activities of h: [0.2565 0.2529 0.2505]
Excess canalization of h: [0.16016667 0.16376667 0.16616667]
Effective degree of h: 1.25
Average edge effectiveness of h: 0.4166666666666667
Normalized input redundancy of h: 0.5833333333333334

Edge effectiveness of k: [0.8125, 0.375, 0.375]
Activities of k: [0.7551 0.2473 0.2525]
Excess canalization of k: [0.0574 0.1277 0.1225]
Effective degree of k: 1.5625
Average edge effectiveness of k: 0.5208333333333334
```

```
Normalized input redundancy of k: 0.4791666666666667
```

3.6 Summary and next steps

In this tutorial you learned how to:

- compute canalizing depth and identify nested canalizing functions,
- compute the canalizing layer structure and interpret layers and core functions,
- quantify collective canalization via k-set canalizing proportions,
- summarize canalization via canalizing strength,
- relate canalization to redundancy-based measures such as edge effectiveness.

Canalization provides a structural explanation for why many biological Boolean rules are robust to perturbations.

Next steps: Subsequent tutorials will explore random Boolean functions with prescribed canalization properties and the impact of canalization on Boolean network dynamics and robustness.

4 Random Boolean Function Generation

This tutorial focuses on the random generation of Boolean functions with prescribed properties, enabling large-scale computational studies.

Random Boolean function generation enables: 1. Null model comparisons: Are biological networks special? 2. Ensemble studies: How do structural properties affect dynamics? 3. Robustness testing: Sample the space of equivalent models 4. Theoretical predictions: Derive expected values for network properties

4.1 What you will learn

In this tutorial you will learn how to generate random Boolean functions with:

- specified canalizing properties (depth, layer structure),
- bias, absolute bias, or a specific Hamming weight,
- linearity constraints,
- degeneracy constraints.

It is strongly recommended to complete the previous tutorials first.

4.2 Setup

```
import boolforge
import numpy as np
import matplotlib.pyplot as plt
```

4.3 Generating random Boolean functions

The function `boolforge.random_function(n, *args)` generates a random n-input Boolean function subject to optional constraints. By default, it generates a **non-degenerate** function, meaning that all variables are essential.

```
n = 3
f = boolforge.random_function(n)

boolforge.display_truth_table(f, labels="f_random_non_degenerate")

print("Is f degenerate?", f.is_degenerate())
print("Activities of f:", f.get_activities(exact=True))
print("Edge effectiveness of f:", f.get_edge_effectiveness())
```

```
x0  x1  x2  |   f_random_non_degenerate
-----------------------------------------------------------
0   0   0   |   0
0   0   1   |   1
0   1   0   |   1
0   1   1   |   0
1   0   0   |   1
1   0   1   |   1
1   1   0   |   0
1   1   1   |   1
Is f degenerate? False
Activities of f: [0.75 0.75 0.75]
Edge effectiveness of f: [0.8333333333333334, 0.8333333333333334, 0.8333333333333334]
```

The rest of this tutorial describes the various constraints. Each constraint defines a specific family of n-input Boolean functions, from which `boolforge.random_function(n,*args)` samples *uniformly at random*. That is, each function satisfying a given set of constraints is selected with equal probability.

4.4 Parity functions

Setting `parity=True` generates *parity* functions, also known as non-degenerate *linear* functions.

```
f = boolforge.random_function(n, parity=True)

boolforge.display_truth_table(f, labels="f_linear")

print("Activities:", f.get_activities(exact=True))
print("Edge effectiveness:", f.get_edge_effectiveness())
print("Normalized average sensitivity:", f.get_average_sensitivity(exact=True))
print("Canalizing strength:", f.get_canalizing_strength())

# Parity functions are the only Boolean functions with activity 1 (for all variables),
# normalized average sensitivity 1 and canalizing strength 0.
```

```
x0  x1  x2  |   f_linear
-----------------------------------------
0   0   0   |   0
0   0   1   |   1
0   1   0   |   1
0   1   1   |   0
1   0   0   |   1
1   0   1   |   0
1   1   0   |   0
1   1   1   |   1
Activities: [1. 1. 1.]
Edge effectiveness: [1.0, 1.0, 1.0]
Normalized average sensitivity: 1.0
Canalizing strength: 0.0
```

4.5 Functions with prescribed canalizing properties

If `parity=False` (default), the canalizing layer structure can be specified via `layer_structure`. This specifies the number of conditionally canalizing variables in each layer of the randomly generated function. If the optional argument `exact_depth=True` (default is False), then this describes the exact layer structure, i.e., the core function cannot be canalizing. If `exact_depth=False` (the default), it is possible that the core function is canalizing, meaning that the last described layer in `layer_structure` may have more conditionally canalizing variables, or that there are additional canalizing layers.

Before generating any random function, `random_function()` goes through a number of checks ensuring that the provided optional arguments make sense. For example, it checks that the provided layer structure $(k_1, \dots, k_r)$ satisfies - $k_i \geq 1$, - $k_1 + \cdots + k_r \leq n$, and - if $k_1 + \cdots + k_r = n$, then $k_r \geq 2$ because the last layer of a nested canalizing function must always contain two or more variables.

```
f = boolforge.random_function(n, layer_structure=[1])
g = boolforge.random_function(n, layer_structure=[1], exact_depth=True)
h = boolforge.random_function(n, layer_structure=[3])
k = boolforge.random_function(n, layer_structure=[1, 2])

labels = ["f", "g", "h", "k"]
boolforge.display_truth_table(f, g, h, k, labels=labels)

for func, label in zip([f, g, h, k], labels):
    info = func.get_layer_structure()
    print(f"Canalizing depth of {label}: {func.get_canalizing_depth()}")
    print(f"Layer structure of {label}: {info['LayerStructure']}")
```

```
    print(f"Number of layers of {label}: {info['NumberOfLayers']}")
    print(f"Core function of {label}: {info['CoreFunction']}")
    print()
```

```
x0  x1  x2  |   f   g   h   k
-------------------------------------------------------
0   0   0   |   1   0   1   0
0   0   1   |   0   1   1   1
0   1   0   |   1   1   1   0
0   1   1   |   0   1   1   0
1   0   0   |   1   1   1   0
1   0   1   |   0   1   1   1
1   1   0   |   0   0   0   0
1   1   1   |   0   1   1   1
Canalizing depth of f: 3
Layer structure of f: [1, 2]
Number of layers of f: 2
Core function of f: [0]

Canalizing depth of g: 1
Layer structure of g: [1]
Number of layers of g: 1
Core function of g: [0 1 1 0]

Canalizing depth of h: 3
Layer structure of h: [3]
Number of layers of h: 1
Core function of h: [0]

Canalizing depth of k: 3
Layer structure of k: [1, 2]
Number of layers of k: 2
Core function of k: [0]
```

Repeated evaluation of this block of code shows that the canalizing depth of `f` is either 1 or 3 (note that a canalizing depth of $n-1$ is never possible for a non-degenerate function). On the contrary, the canalizing depth of `g` is always 1 because we set `exact_depth=True`. The 2-input core function of `g` is one of the two parity functions, each with 50% probability. Likewise, the core function for the other functions is simply [0] or [1], each with 50% probability. Functions `h` and `k` are nested canalizing, i.e., their canalizing depth is 3. Their layer structure is exactly as specified.

If we do not care about the specific layer structure but only about the canalizing depth, we specify the optional argument `depth` instead of `layer_structure`.

```
# any function has at least canalizing depth 0 so this is the same as boolforge.random_function(n)
f = boolforge.random_function(n,depth=0)

# a random non-canalizing function
g = boolforge.random_function(n,depth=0,exact_depth=True)

# a random canalizing function
h = boolforge.random_function(n,depth=1)

# a random nested canalizing function
k = boolforge.random_function(n,depth=n)
```

```
labels = ["f", "g", "h", "k"]
boolforge.display_truth_table(f, g, h, k, labels=labels)

for func, label in zip([f, g, h, k], labels):
    print(f"Canalizing depth of {label}: {func.get_canalizing_depth()}")
    print()
```

```
x0  x1  x2  |   f   g   h   k
-------------------------------------------------------
0   0   0   |   1   0   1   0
0   0   1   |   1   0   1   0
0   1   0   |   0   0   0   0
0   1   1   |   1   1   1   0
1   0   0   |   1   1   1   0
1   0   1   |   0   1   1   1
1   1   0   |   1   1   1   1
1   1   1   |   0   0   1   1
Canalizing depth of f: 0

Canalizing depth of g: 0

Canalizing depth of h: 3

Canalizing depth of k: 3
```

Repeated evaluation of this block of code shows that the canalizing depth of `f` can be 0, 1, or 3. Note that specifying `depth=0` without `exact_depth=True` does not restrict the space of functions at all. On the contrary, the canalizing depth of `g` is always 0 (i.e., g does not contain any canalizing variables) because we set `exact_depth=True`. Function `h` is canalizing and may be nested canalizing (because we specified that the minimal canalizing depth is 1), and `k` is always nested canalizing (i.e., it has canalizing depth $n = 3$).

We remember: If `exact_depth=True`, `depth` is interpreted as exact canalizing depth. Otherwise (default), `depth` is interpreted as minimal canalizing depth. For example,

- `depth=1`: "At least 1-canalizing" (could be 2,3,...,n-canalizing)
- `depth=1, exact_depth=True`: "Exactly 1-canalizing" (not 2,3,...,n-canalizing)

4.6 Allowing degenerate functions

It is possible that an n-input Boolean function does not depend on all its variables. For example, the function $f(x, y) = x$ depends on x but not on y. By default, such degenerate functions are never generated by `boolforge.random_function()`. To enable the generation of possibly degenerate functions, we set `allow_degenerate_functions=True`. Although hardly of any practical value, we can even restrict the random generation to degenerate functions only, using `boolforge.random_degenerate_function(n,*args)`.

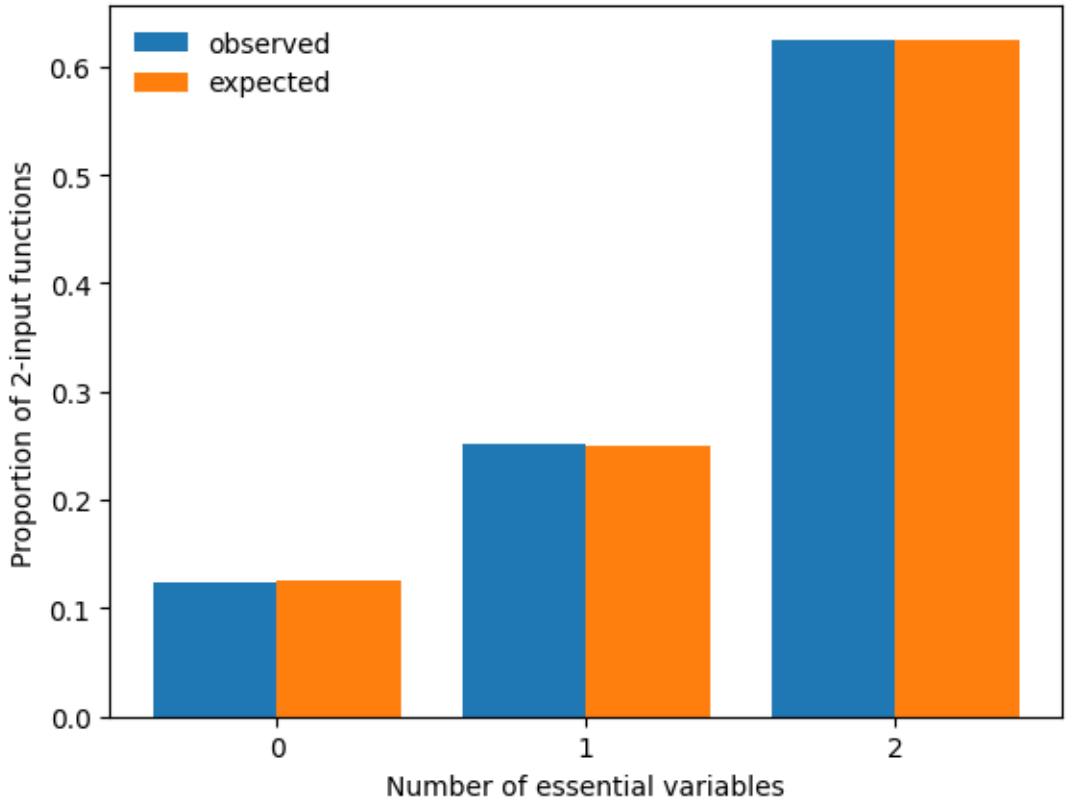
Note: When generating random canalizing functions, the value of `allow_degenerate_functions` is ignored. The non-canalizing core function is assumed to always depend on all its variables.

Since degenerate functions occur much more frequently at low degree, we set `n=2`, generate a large number of random, possibly degenerate functions and compare a histogram of the observed number of essential variables to the expected proportions.

```
n = 2
n_simulations = 10000

count_essential = np.zeros(n + 1, dtype=int)

for _ in range(n_simulations):
```

```
    f = boolforge.random_function(n, allow_degenerate_functions=True)
    count_essential[f.get_number_of_essential_variables()] += 1

expected = np.array([2 / 16, 4 / 16, 10 / 16])

x = np.arange(n + 1)
width = 0.4

fig, ax = plt.subplots()
ax.bar(x - width / 2, count_essential / n_simulations, width=width, label="observed")
ax.bar(x + width / 2, expected, width=width, label="expected")
ax.legend(frameon=False)
ax.set_xticks(x)
ax.set_xlabel("Number of essential variables")
ax.set_ylabel(f"Proportion of {n}-input functions")

print("Error:", count_essential / n_simulations - expected)
plt.show()
```

```
Error: [-0.0017  0.002  -0.0003]
```



4.7 Functions with prescribed Hamming weight

The Hamming weight of a Boolean function is the number of ones in its truth table. BoolForge allows for the generation of random n-input functions with a specific Hamming weight $w \in \{0, 1, \dots, 2^n\}$. The additional optional parameters `allow_degenerate_functions` and `exact_depth` specify whether degenerate and canalizing functions are allowed. By default, canalizing functions are allowed, while degenerate functions are not. Since all functions with Hamming weight $w \in \{0, 1, 2^n - 1, 2^n\}$ are canalizing, we require $2 \leq w \leq 2^n - 2$ whenever

canalizing functions are not permissible (i.e., whenever `exact_depth=True`).

```
n = 3

f = boolforge.random_function(n, hamming_weight=5)
g = boolforge.random_function(n, hamming_weight=5, exact_depth=True)
h = boolforge.random_function(n, hamming_weight=2, allow_degenerate_functions=True)

labels = ["f", "g", "h"]
boolforge.display_truth_table(f, g, h, labels=labels)

for func, label in zip([f, g, h], labels):
    print(f"Hamming weight of {label}: {func.get_hamming_weight()}")
    print(f"Canalizing depth of {label}: {func.get_canalizing_depth()}")
    print(f"Number of essential variables of {label}: {func.get_number_of_essential_variables()}")
    print()
```

```
x0  x1  x2  |   f   g   h
-----------------------------------------------
0   0   0   |   0   0   0
0   0   1   |   1   0   0
0   1   0   |   1   1   0
0   1   1   |   1   1   1
1   0   0   |   0   1   0
1   0   1   |   1   1   1
1   1   0   |   0   0   0
1   1   1   |   1   1   0
Hamming weight of f: 5
Canalizing depth of f: 3
Number of essential variables of f: 3

Hamming weight of g: 5
Canalizing depth of g: 0
Number of essential variables of g: 3

Hamming weight of h: 2
Canalizing depth of h: 1
Number of essential variables of h: 3
```

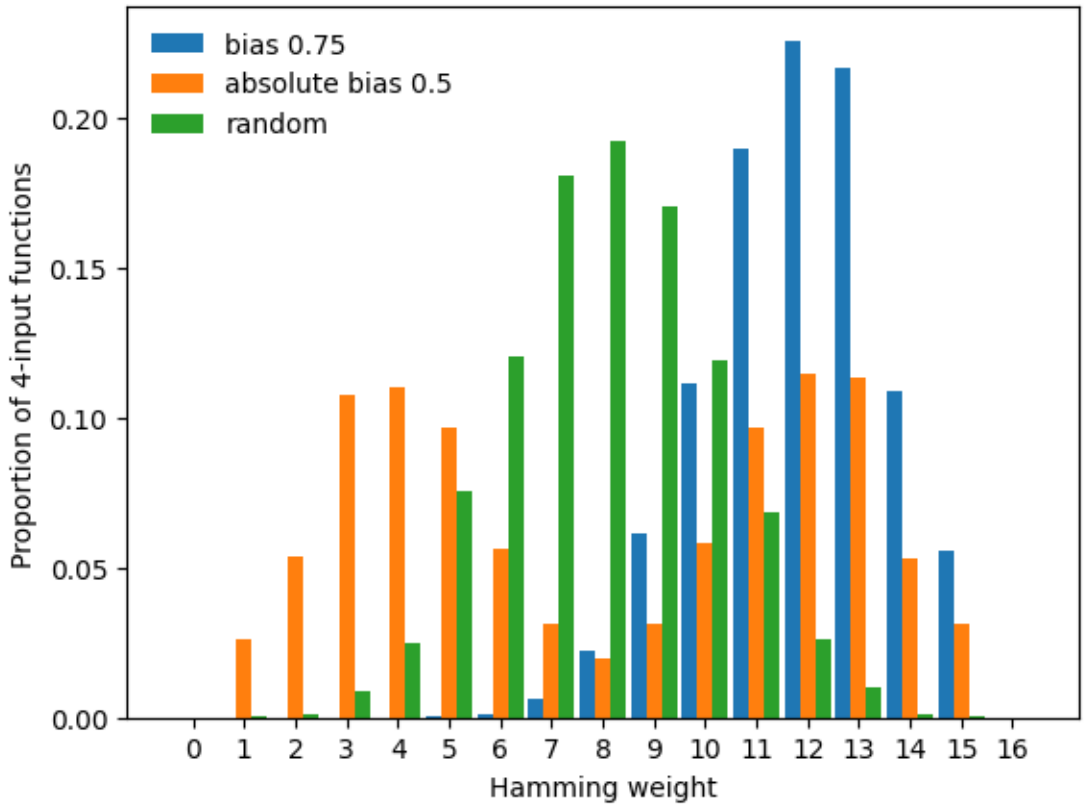
4.8 Biased and absolutely biased functions

While specifying the Hamming weight fixes the exact number of 1s in the truth table of a generated function, specifying the bias or absolute bias acts slightly differently. The bias p describes the probability of selecting a 1 at any position in the truth table and can be modified using the optional argument `bias`. Instead of specifying the bias, the absolute bias may also be specified. Unbiased functions generated using $p = 0.5$ have an absolute bias of 0, the default. If, for example, we set `absolute_bias=0.5` and specify to use absolute bias (`use_absolute_bias=True`, default is False), the bias used to generate the function is either 0.25 or 0.75, both with probability 50%. Generally, if we set `use_absolute_bias=True; absolute_bias=a` for $a \in [0, 1]$, the bias is either $(1 + a)/2$ or $(1 - a)/2$, both with probability 50%.

To display these different modes, we repeatedly generate random Boolean functions under three different constraints (`f` with bias $p = 0.75$, `g` with absolute bias 0.5, and `h` an unbiased function, i.e., with bias $p = 0.5$), and compare the empirical Hamming weight distribution of the three families of functions.

```
n = 4
n_simulations = 10000
```

```
counts = np.zeros((3, 2**n + 1), dtype=int)

for _ in range(n_simulations):
    f = boolforge.random_function(n, bias=0.75)
    g = boolforge.random_function(n, absolute_bias=0.5, use_absolute_bias=True)
    h = boolforge.random_function(n, absolute_bias=0.5)

    counts[0, f.get_hamming_weight()] += 1
    counts[1, g.get_hamming_weight()] += 1
    counts[2, h.get_hamming_weight()] += 1

labels = ["bias 0.75", "absolute bias 0.5", "random"]
x = np.arange(2**n + 1)
width = 0.3

fig, ax = plt.subplots()
for i, label in enumerate(labels):
    ax.bar(x - width + i * width, counts[i] / n_simulations, width=width, label=label)

ax.legend(frameon=False)
ax.set_xticks(x)
ax.set_xlabel("Hamming weight")
ax.set_ylabel(f"Proportion of {n}-input functions")
plt.show()
```



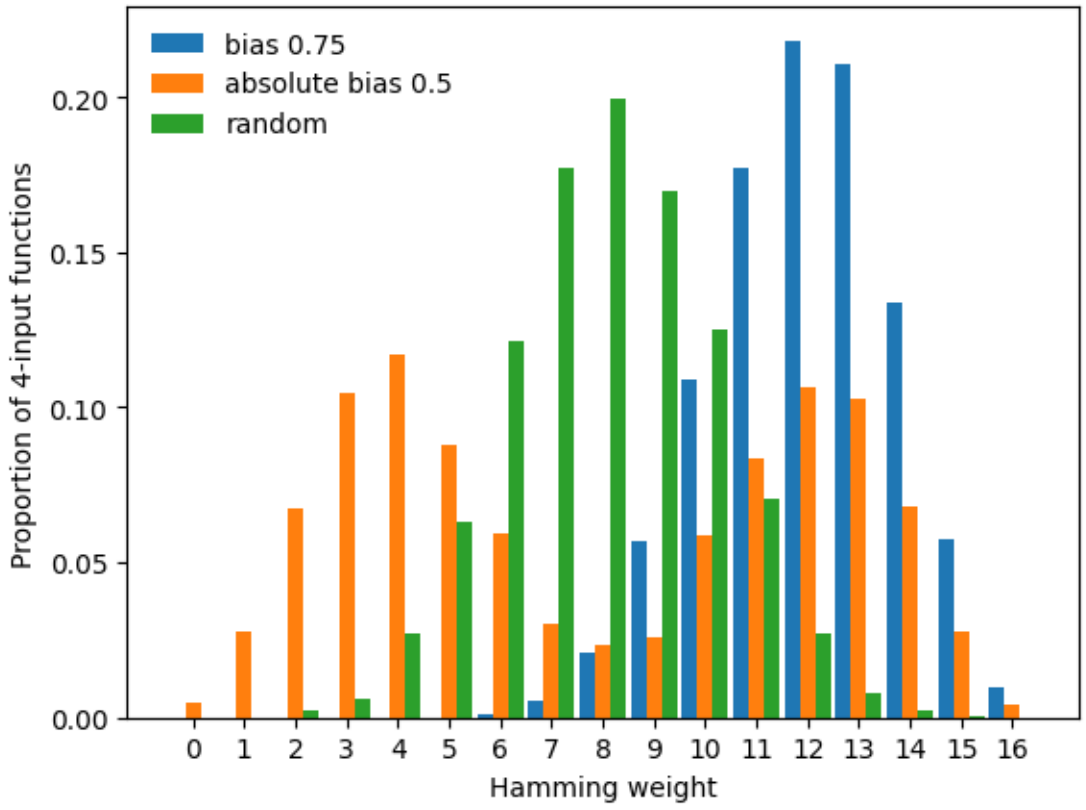
This plot exemplifies the difference between bias and absolute bias:

- Specifying the bias shifts the mode of the Hamming weight distribution to the value of `bias`.
- Specifying the absolute bias yields random functions with a bimodal Hamming weight distribution.

Note that `absolute_bias=0.5` is ignored in the generation of `h`. The desired use of absolute bias must be specified by `use_absolute_bias=True`.

In the above plot, we notice a lack of functions with Hamming weight 0 and $16 = 2^n$. These constant functions are degenerate and thus not generated unless we set `allow_degenerate_functions=True`, which as we see below slightly modifies the resulting Hamming weight distributions.

```
counts[:] = 0

for _ in range(n_simulations):
    f = boolforge.random_function(n, bias=0.75, allow_degenerate_functions=True)
    g = boolforge.random_function(
        n, absolute_bias=0.5, use_absolute_bias=True, allow_degenerate_functions=True
    )
    h = boolforge.random_function(n, absolute_bias=0.5, allow_degenerate_functions=True)

    counts[0, f.get_hamming_weight()] += 1
    counts[1, g.get_hamming_weight()] += 1
    counts[2, h.get_hamming_weight()] += 1

fig, ax = plt.subplots()
for i, label in enumerate(labels):
    ax.bar(x - width + i * width, counts[i] / n_simulations, width=width, label=label)

ax.legend(frameon=False)
ax.set_xticks(x)
ax.set_xlabel("Hamming weight")
ax.set_ylabel(f"Proportion of {n}-input functions")
plt.show()
```



4.9 Summary

This tutorial demonstrated how BoolForge enables uniform random generation of Boolean functions under flexible constraints. Different constraints define fundamentally different ensembles, and being explicit about these choices is essential for a correct generation and interpretation of computational results.

Next steps: The next tutorial exemplifies how these function-level ensembles can be used to uncover new insights into biological regulatory networks, as well as the relationship between network structure and dynamics.

4.10 Common pitfalls

- `absolute_bias` has no effect unless `use_absolute_bias=True`.
- `depth=0` without `exact_depth=True` does not restrict the function space since any Boolean function is at least 0-canalizing.
- Constant functions and other degenerate functions that do not depend on all inputs are only generated if `allow_degenerate_functions=True`. If `layer_structure` or `depth>0` are provided, canalizing functions are generated, which always depend on all inputs. That is, in these cases a possible parameter choice of `allow_degenerate_functions=True` is ignored.
- For larger n (e.g., $n > 5$), set `allow_degenerate_functions=True` to avoid expensive degeneracy tests. Almost all functions in many variables are non-degenerate.

5 Example Use Cases of the Random Function Generator

In this tutorial, we explore how BoolForge's random Boolean function generator can be used to generate large ensembles of Boolean functions with prescribed structural properties, whose statistical and dynamical characteristics can then be studied.

5.1 What you will learn

In this tutorial you will learn how to:

- compute the prevalence of canalization, k-canalization, and nested canalization,
- determine distributions of canalizing strength and normalized input redundancy,
- investigate correlations between absolute bias and canalization,
- generate and analyze dynamically distinct nested canalizing functions.

It is strongly recommended to complete the previous tutorials first.

5.2 Setup

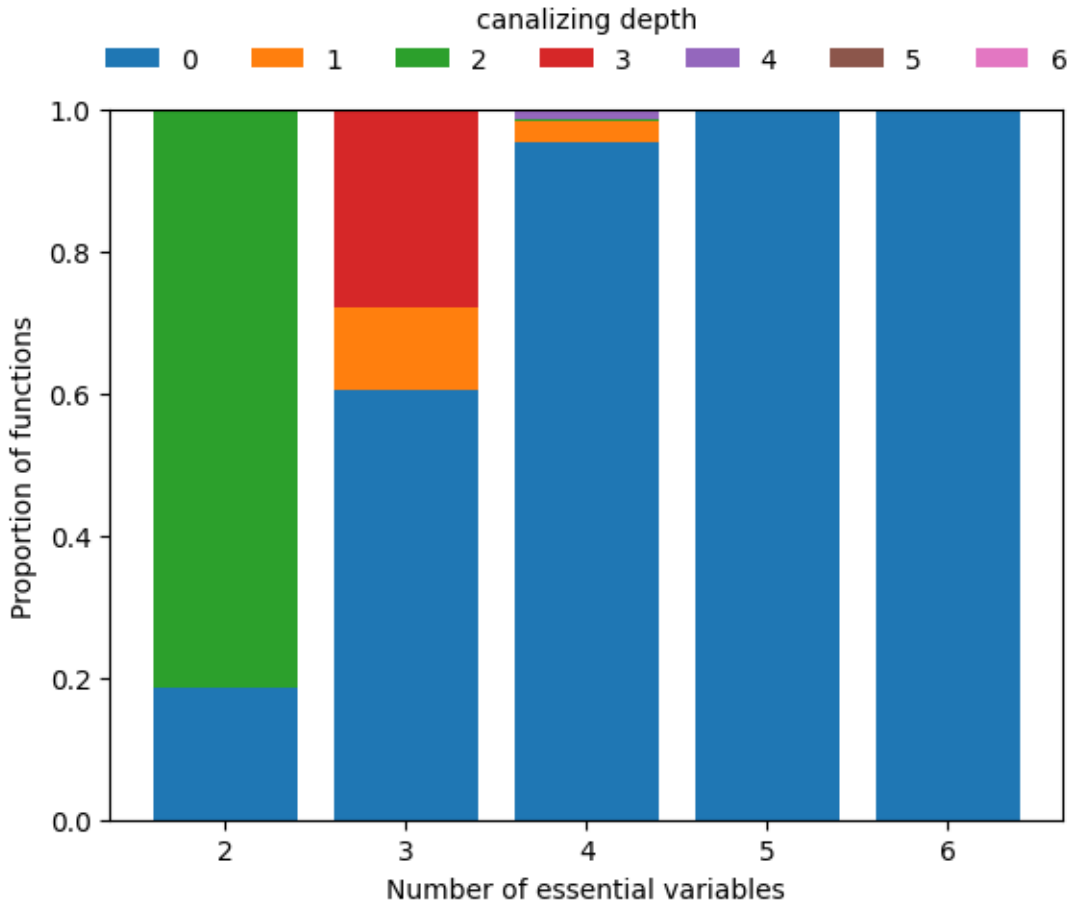
```
import boolforge
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy.stats as stats
```

5.3 Prevalence of canalization

Using random sampling, we estimate how frequently Boolean functions of degree n exhibit a given canalizing depth.

```
n_simulations = 1000
ns = np.arange(2, 7)
canalizing_depths = np.arange(max(ns) + 1)

count_depths = np.zeros((len(ns), max(ns) + 1))

for _ in range(n_simulations):
    for i, n in enumerate(ns):
        f = boolforge.random_function(n)
        count_depths[i, f.get_canalizing_depth()] += 1

count_depths /= n_simulations

fig, ax = plt.subplots()
for i, depth in enumerate(canalizing_depths):
    ax.bar(
        ns,
        count_depths[:, i],
        bottom=np.sum(count_depths[:, :i], axis=1),
        label=str(depth),
    )

ax.legend(
    frameon=False,
    loc="center",
    bbox_to_anchor=(0.5, 1.1),
```

```
    ncol=8,
    title="canalizing depth",
)
ax.set_xticks(ns)
ax.set_xlabel("Number of essential variables")
ax.set_ylabel("Proportion of functions")
plt.show()

out = pd.DataFrame(
    count_depths,
    index=["n=" + str(n) for n in ns],
    columns=["k=" + str(k) for k in canalizing_depths],
)

print(out.to_string())
```




```
      k=0    k=1    k=2    k=3    k=4  k=5  k=6
n=2  0.185  0.000  0.815  0.000  0.000  0.0  0.0
n=3  0.605  0.116  0.000  0.279  0.000  0.0  0.0
n=4  0.953  0.031  0.004  0.000  0.012  0.0  0.0
n=5  1.000  0.000  0.000  0.000  0.000  0.0  0.0
n=6  1.000  0.000  0.000  0.000  0.000  0.0  0.0
```

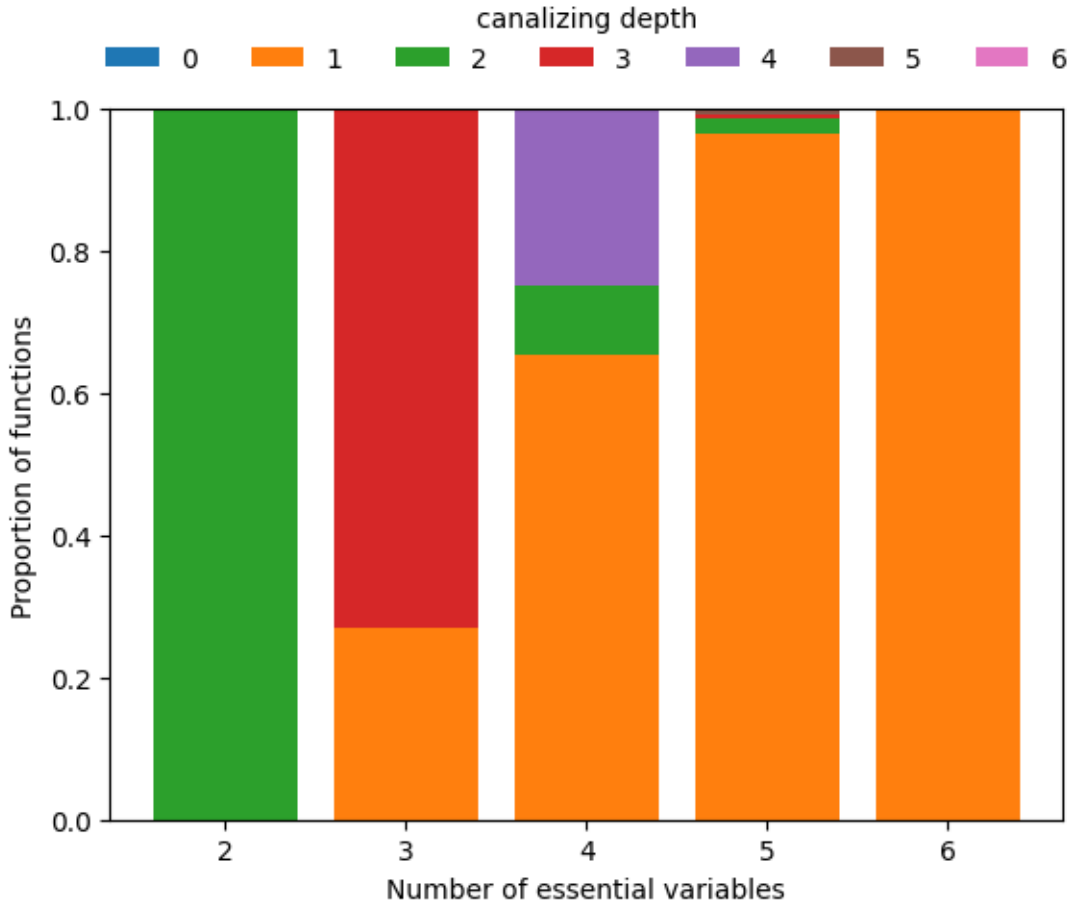
We see that hardly any Boolean function with $n \geq 5$ inputs is canalizing, let alone nested canalizing. This makes the finding that most Boolean functions in published Boolean gene regulatory network models are

nested canalizing very surprising (Kadelka et al., Science Advances, 2024).

5.3.1 Restricting to canalizing functions

To zoom in on the few functions that are canalizing for higher n, we can simply require `depth=1` and repeat the above analysis.

```
count_depths = np.zeros((len(ns), max(ns) + 1))

for _ in range(n_simulations):
    for i, n in enumerate(ns):
        f = boolforge.random_function(n, depth=1)
        count_depths[i, f.get_canalizing_depth()] += 1

count_depths /= n_simulations

fig, ax = plt.subplots()
for i, depth in enumerate(canalizing_depths):
    ax.bar(
        ns,
        count_depths[:, i],
        bottom=np.sum(count_depths[:, :i], axis=1),
        label=str(depth),
    )

ax.legend(
    frameon=False,
    loc="center",
    bbox_to_anchor=(0.5, 1.1),
    ncol=8,
    title="canalizing depth",
)
ax.set_xticks(ns)
ax.set_xlabel("Number of essential variables")
ax.set_ylabel("Proportion of functions")
plt.show()

out = pd.DataFrame(
    count_depths,
    index=["n=" + str(n) for n in ns],
    columns=["k=" + str(k) for k in canalizing_depths],
);

print(out.to_string())
```



	k=0	k=1	k=2	k=3	k=4	k=5	k=6
n=2	0.0	0.000	1.000	0.000	0.000	0.000	0.0
n=3	0.0	0.269	0.000	0.731	0.000	0.000	0.0
n=4	0.0	0.653	0.098	0.000	0.249	0.000	0.0
n=5	0.0	0.965	0.022	0.004	0.000	0.009	0.0
n=6	0.0	1.000	0.000	0.000	0.000	0.000	0.0

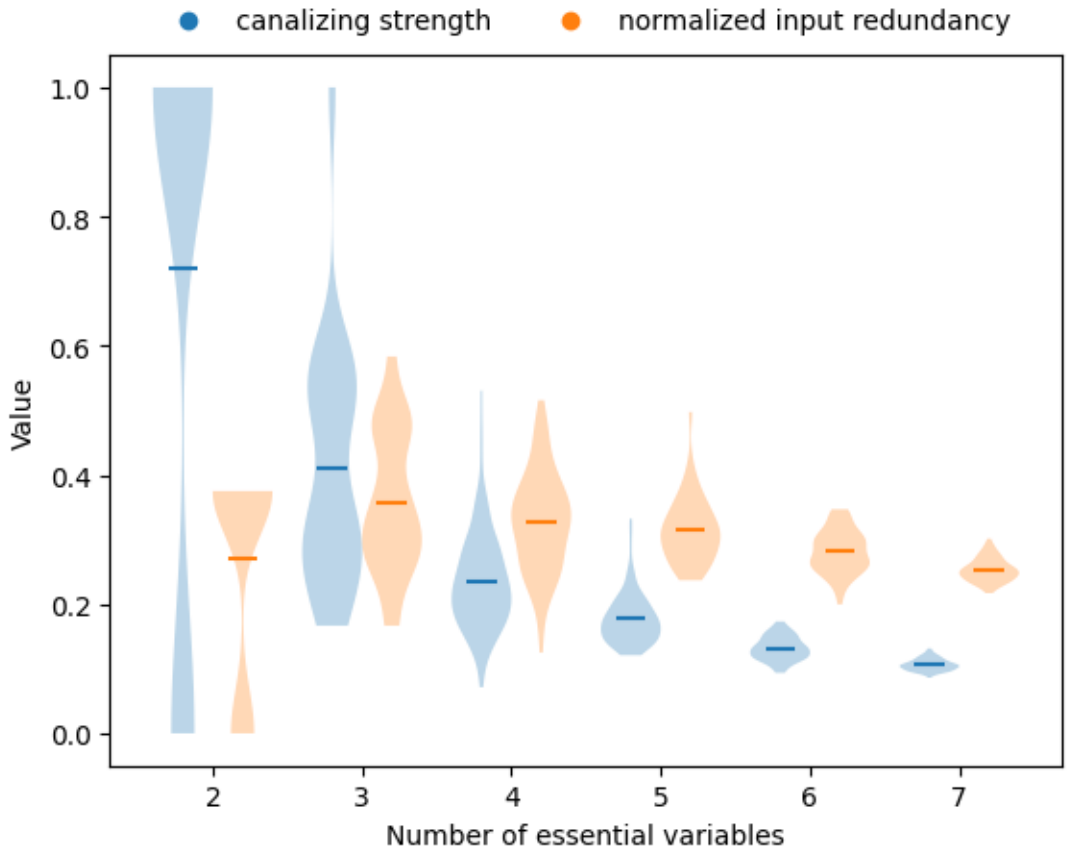
This analysis reveals that among Boolean functions of degree $n \geq 5$, functions with few conditionally canalizing variables are much more abundant than functions with more conditionally canalizing variables, which is mathematically obvious due to the recursive nature of the definition of k-canalization.

5.4 Collective canalization vs degree

Using a similar setup, we can investigate if and how the various measures of collective canalization, specifically canalizing strength (Kadelka et al., Adv in Appl Math, 2023) and the normalized input redundancy (Gates et al., PNAS, 2021), change when the degree of the functions changes.

```
n_simulations = 100
ns = np.arange(2, 8)

canalizing_strengths = np.zeros((len(ns), n_simulations))
input_redundancies = np.zeros((len(ns), n_simulations))

for j in range(n_simulations):
    for i, n in enumerate(ns):
```

```
        f = boolforge.random_function(n)
        canalizing_strengths[i, j] = f.get_canalizing_strength()
        input_redundancies[i, j] = f.get_input_redundancy()

width = 0.4
fig, ax = plt.subplots()

ax.violinplot(
    canalizing_strengths.T,
    positions=ns - width / 2,
    widths=width,
    showmeans=True,
    showextrema=False,
)
ax.scatter([], [], color="C0", label="canalizing strength")

ax.violinplot(
    input_redundancies.T,
    positions=ns + width / 2,
    widths=width,
    showmeans=True,
    showextrema=False,
)
ax.scatter([], [], color="C1", label="normalized input redundancy")

ax.legend(
    loc="center",
    bbox_to_anchor=(0.5, 1.05),
    frameon=False,
    ncol=2,
)
ax.set_xlabel("Number of essential variables")
ax.set_ylabel("Value")
plt.show()
```



Both measures decrease with increasing degree, but canalizing strength declines more sharply.

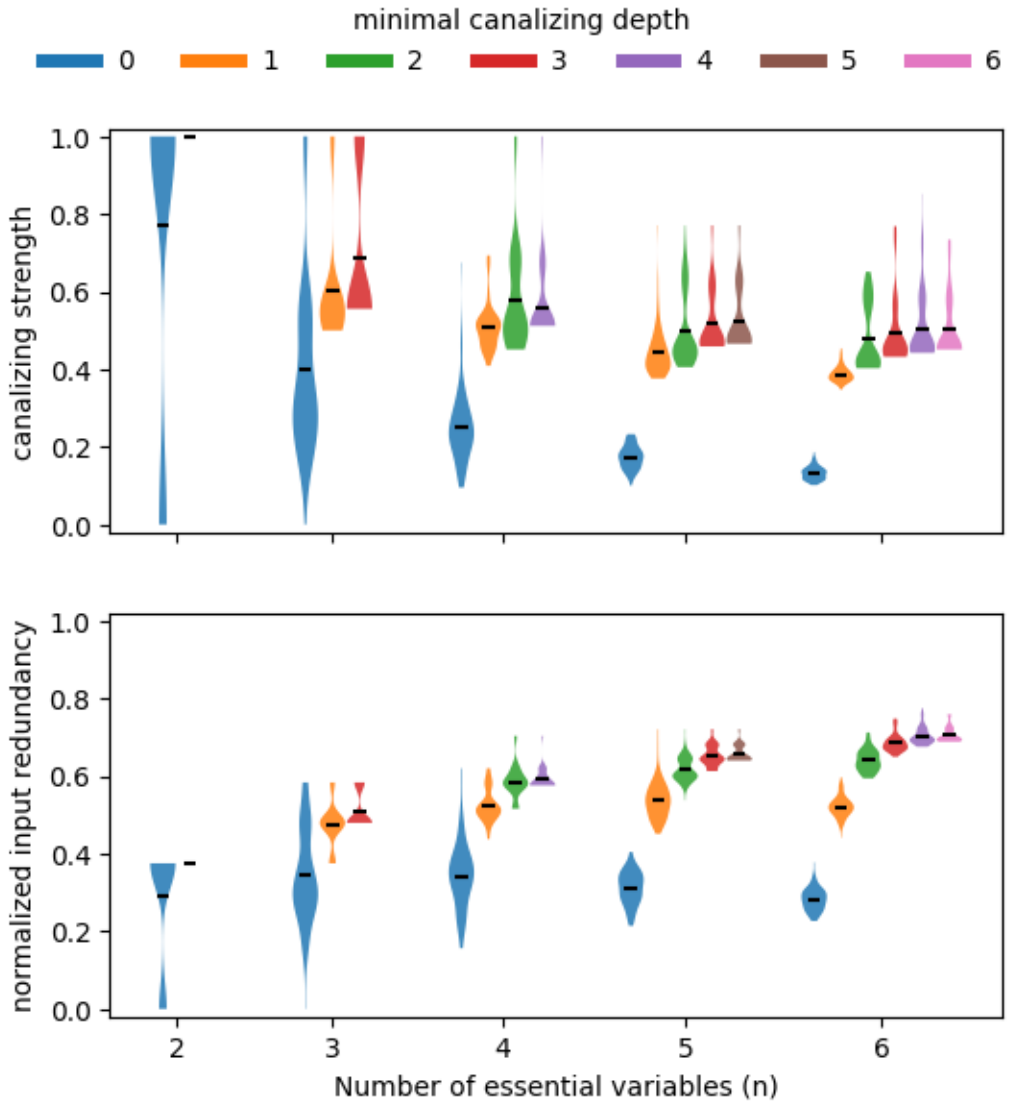
5.4.1 Stratification by canalizing depth

If we stratify this analysis by canalizing depth (exact canalizing depth using `exact_depth=True` or minimal canalizing depth using the default `exact_depth=False`), we can confirm that functions with more conditionally canalizing variables tend to also have higher average collective canalization, irrespective of how it is measured. In other words, the various measures of canalization are all highly correlated.

```
n_simulations = 100
exact_depth = False
ns = np.arange(2, 7)

max_depth = max(ns)

canalizing_strengths = np.zeros((len(ns), max_depth + 1, n_simulations))
input_redundancies = np.zeros((len(ns), max_depth + 1, n_simulations))

for k in range(n_simulations):
    for i, n in enumerate(ns):
        for depth in np.append(np.arange(n - 1), n):
            f = boolforge.random_function(n, depth=depth, exact_depth=exact_depth)
            canalizing_strengths[i, depth, k] = f.get_canalizing_strength()
            input_redundancies[i, depth, k] = f.get_input_redundancy()

fig, ax = plt.subplots(2, 1, figsize=(6, 6), sharex=True)
```

```
base_gap = 1.0
intra_gap = 0.3
width = 0.28

for ii, (data, label) in enumerate(
    zip(
        [canalizing_strengths, input_redundancies],
        ["canalizing strength", "normalized input redundancy"],
    )
):
    positions = []
    values = []
    colors = []
    group_centers = []

    current_x = 0.0
    for i, n in enumerate(ns):
        depths = np.append(np.arange(n - 1), n)
        offsets = np.linspace(
            -(len(depths) - 1) * intra_gap / 2,
            (len(depths) - 1) * intra_gap / 2,
            len(depths),
        )
        group_positions = current_x + offsets
        positions.extend(group_positions)
        group_centers.append(current_x)

        for d in depths:
            values.append(data[i, d, :])
            colors.append("C" + str(d))

        group_width = (len(depths) - 1) * intra_gap
        current_x += group_width / 2 + base_gap + width + intra_gap

    for pos, val, c in zip(positions, values, colors):
        vp = ax[ii].violinplot(val, positions=[pos], widths=width, showmeans=True, showextrema=False)
        for body in vp["bodies"]:
            body.set_facecolor(c)
            body.set_alpha(0.85)
        vp["cmeans"].set_color("k")

    ax[ii].set_ylabel(label)
    ax[ii].set_ylim([-0.02, 1.02])

ax[1].set_xlabel("Number of essential variables (n)")
ax[1].set_xticks(group_centers)
ax[1].set_xticklabels(ns)

depth_handles = [
    plt.Line2D([0], [0], color="C" + str(d), lw=5, label=str(d))
    for d in range(max_depth + 1)
]
```

```
fig.legend(
    handles=depth_handles,
    loc="upper center",
    ncol=7,
    frameon=False,
    title="exact canalizing depth" if exact_depth else "minimal canalizing depth",
)
plt.show()
```

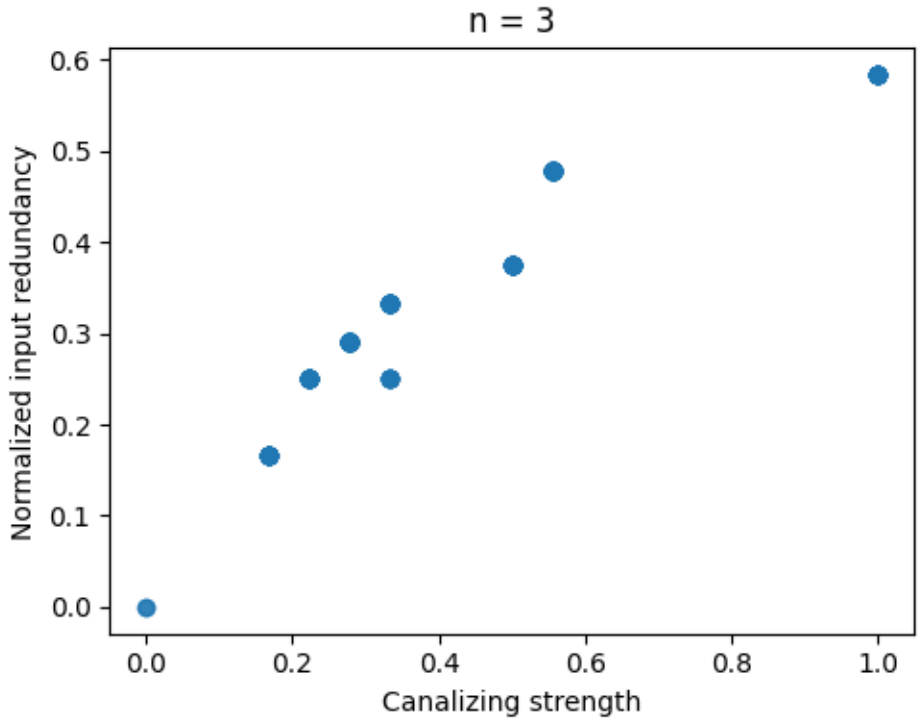


5.4.2 Correlation between canalizing strength and input redundancy

We can generate all (non-degenerate) Boolean functions of a certain degree n (only feasible up to $n = 4$) and compare canalizing strength and input redundancy.

```
n = 3
allow_degenerate_functions = False
degenerate = np.zeros(2 ** (2**n), dtype=bool)
strengths = np.zeros(2 ** (2**n))
```

```
redundancies = np.zeros(2 ** (2**n))

for i, fvec in enumerate(boolforge.get_left_side_of_truth_table(2**n)):
    bf = boolforge.BooleanFunction(fvec)
    strengths[i] = bf.get_canalizing_strength()
    redundancies[i] = bf.get_input_redundancy()
    if not allow_degenerate_functions:
        degenerate[i] = bf.is_degenerate()

if allow_degenerate_functions:
    which = np.ones(2 ** (2**n), dtype=bool)
else:
    which = ~degenerate


plt.figure(figsize=(5, 4))
plt.scatter(strengths[which], redundancies[which], alpha=0.7)
plt.xlabel("Canalizing strength")
plt.ylabel("Normalized input redundancy")
plt.title(f"n = {n}")
plt.tight_layout()
plt.show()

stats.spearmanr(strengths[which], redundancies[which])
```



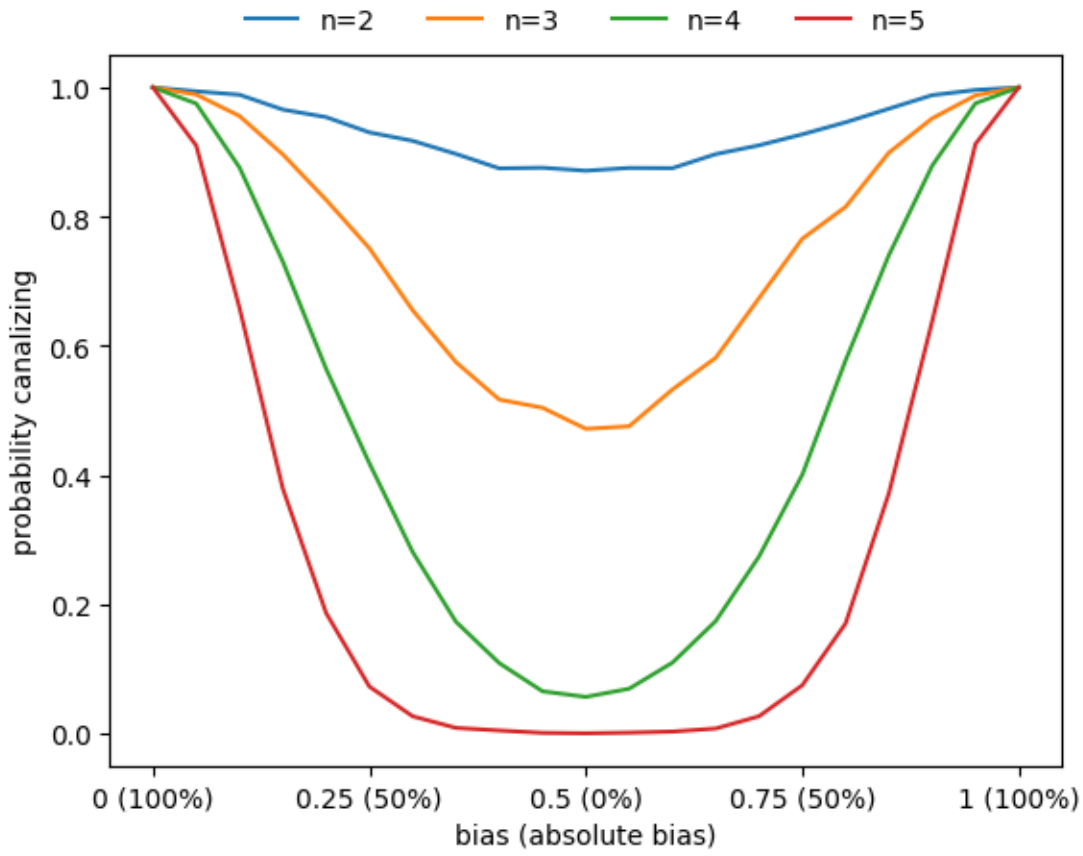

```
SignificanceResult(statistic=np.float64(0.9700304760700043), pvalue=np.float64(1.0759731607818433e-134))
```

Both measures are highly correlated but markedly not the same. Some functions possess relatively high canalizing strength but low input redundancy, and vice versa. It remains an open question what drives this behavior.

5.5 Correlation between canalization and bias

Basically all metrics used to assess the sensitivity of Boolean functions (canalization, absolute bias, average sensitivity) are correlated. For example, functions with higher absolute bias are more likely to be canalizing.

```
ns = np.arange(2, 6)
n_simulations = 3000
bias_values = np.linspace(0, 1, 21)

count_canalizing = np.zeros((len(ns), len(bias_values)), dtype=int)

for i, n in enumerate(ns):
    for _ in range(n_simulations):
        for j, bias in enumerate(bias_values):
            f = boolforge.random_function(n, bias=bias, allow_degenerate_functions=True)
            if f.is_canalizing():
                count_canalizing[i, j] += 1

fig, ax = plt.subplots()
for i, n in enumerate(ns):
    ax.plot(bias_values, count_canalizing[i] / n_simulations, label=f"n={n}")

xticks = [0, 0.25, 0.5, 0.75, 1]
ax.set_xticks(xticks)
ax.set_xticklabels([f"{p} ({round(200*abs(p-0.5))}%)" for p in xticks])
ax.set_xlabel("bias (absolute bias)")
ax.set_ylabel("probability canalizing")
ax.legend(
    loc="center",
    frameon=False,
    bbox_to_anchor=(0.5, 1.05),
    ncol=6,
)
plt.show()
```

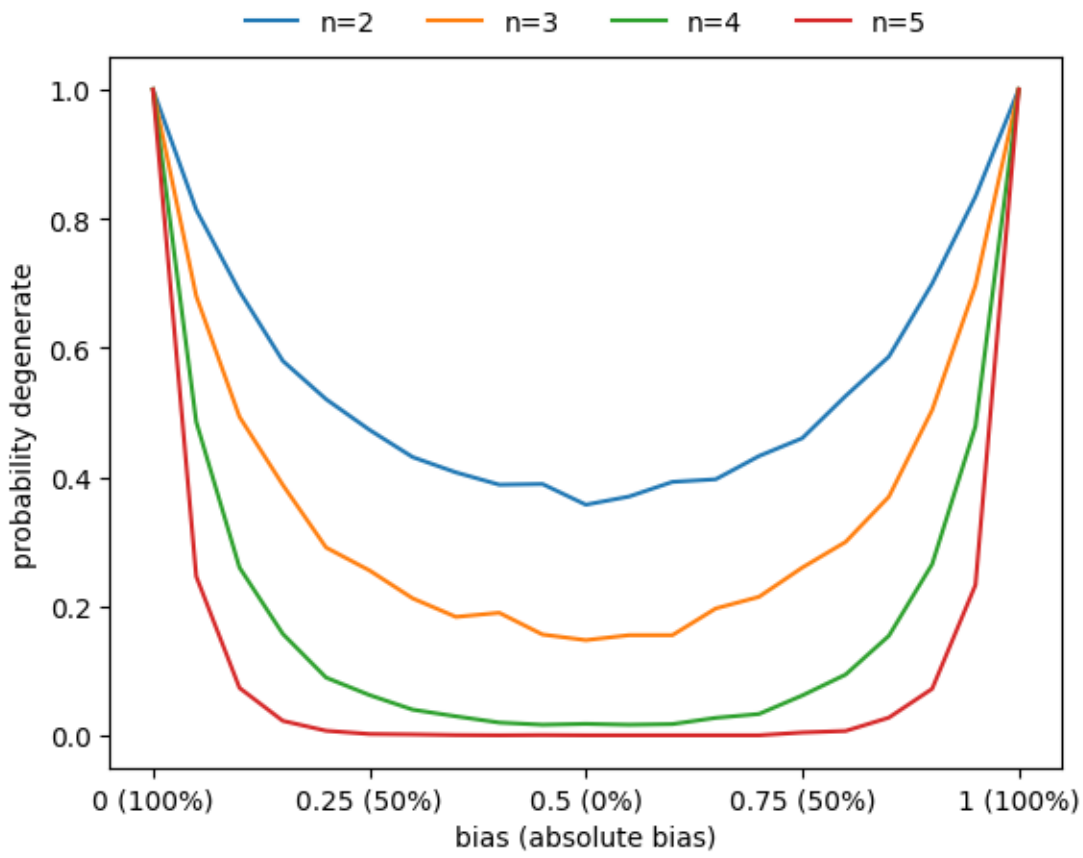


5.5.1 Degeneracy vs bias

Similarly, the probability that a function is degenerate (i.e., that it does not depend on all its variables) also increases as the absolute bias increases.

```
count_degenerate = np.zeros((len(ns), len(bias_values)), dtype=int)

for i, n in enumerate(ns):
    for _ in range(n_simulations):
        for j, bias in enumerate(bias_values):
            f = boolforge.random_function(n, bias=bias, allow_degenerate_functions=True)
            if f.is_degenerate():
                count_degenerate[i, j] += 1

fig, ax = plt.subplots()
for i, n in enumerate(ns):
    ax.plot(bias_values, count_degenerate[i] / n_simulations, label=f"n={n}")

ax.set_xticks(xticks)
ax.set_xticklabels([f"{p} ({round(200*abs(p-0.5))}%)" for p in xticks])
ax.set_xlabel("bias (absolute bias)")
ax.set_ylabel("probability degenerate")
ax.legend(
    loc="center",
    frameon=False,
    bbox_to_anchor=(0.5, 1.05),
```

```
    ncol=6,
)
plt.show()
```



5.6 Analyzing functions with specific canalizing layer structure

The average sensitivity of the Boolean functions governing the updates in a Boolean network, determines the stability of the network to perturbations. More generally, it determines the dynamical regime of the network (see Tutorial 8). The ability to generate canalizing functions with a specific canalizing layer structure enables us to investigate the link between layer structure and average sensitivity, as well as other properties, such as canalizing strength or effective degree.

For nested canalizing functions of a given degree n, there exists a bijection between their absolute bias and their canalizing layer structure (Kadelka et al., Physica D, 2017). The function `boolforge.utils.hamming_weight_to_ncf_layer_structure(degree,hamming_weight)` implements this. NCFs with the same layer structure have the same dynamical properties. That is, they have the same average sensitivity, canalizing strength and the same effective degree. Iterating over all possible absolute biases (parametrized by the possible Hamming weights), we can thus generate all dynamically different types of n-input NCFs and investigate their average sensitivity, which we can compute exactly for relatively low degree.

```
n = 5
all_hamming = np.arange(1, 2 ** (n - 1), 2)
all_abs_bias = all_hamming / 2 ** (n - 1)

avg_sens = np.zeros(2 ** (n - 2))
```

```
can_strength = np.zeros_like(avg_sens)
eff_degree = np.zeros_like(avg_sens)
layer_structures = []

for i, w in enumerate(all_hamming):
    layer = boolforge.utils.hamming_weight_to_ncf_layer_structure(n, w)
    layer_structures.append(layer)
    f = boolforge.random_function(n, layer_structure=layer)
    avg_sens[i] = f.get_average_sensitivity(exact=True, normalized=False)
    can_strength[i] = f.get_canalizing_strength()
    eff_degree[i] = f.get_effective_degree()

df = pd.DataFrame(
    {
        "Hamming weight": all_hamming,
        "Absolute bias": all_abs_bias,
        "Layer structure": list(map(str, layer_structures)),
        "Average sensitivity": avg_sens,
        "Canalizing strength": np.round(can_strength, 4),
        "Effective degree": np.round(eff_degree, 4),
    }
)

print(df.to_string())
```

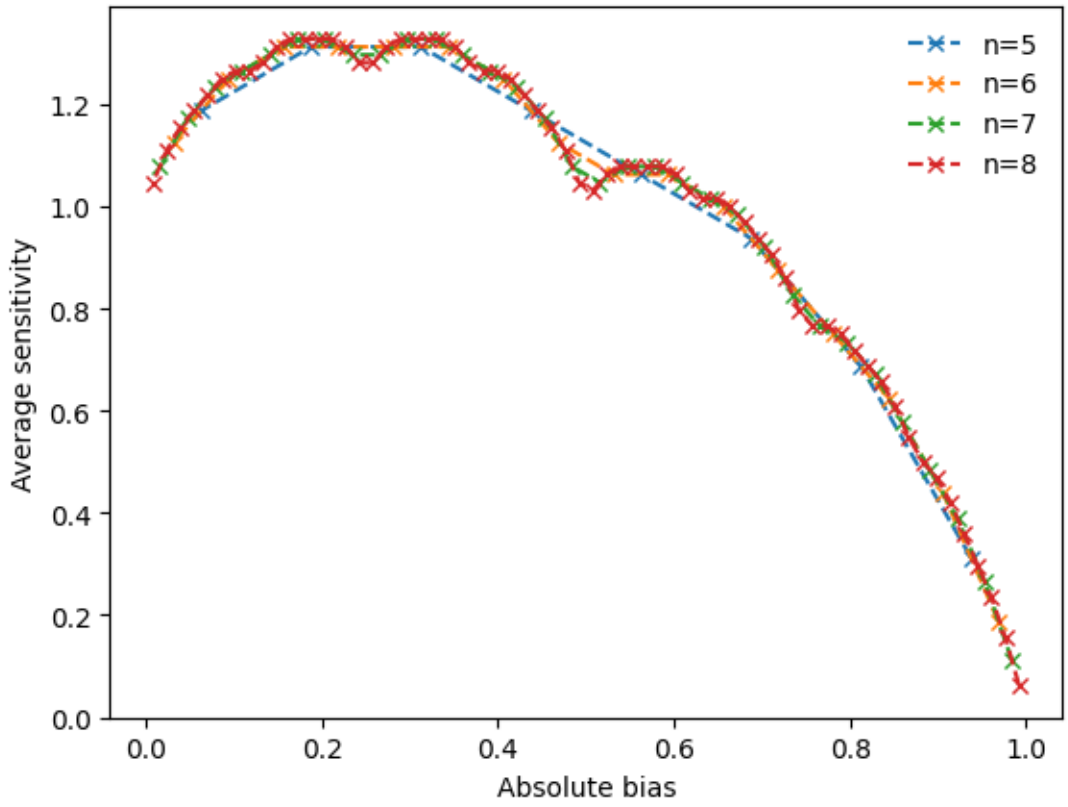
	Hamming weight	Absolute bias	Layer structure	Average sensitivity	Canalizing strength	Effective degree
0	1	0.0625	[5]	0.3125	1.0000	1.1250
1	3	0.1875	[3, 2]	0.6875	0.7705	1.3984
2	5	0.3125	[2, 1, 2]	0.9375	0.6369	1.5938
3	7	0.4375	[2, 3]	1.0625	0.5993	1.5833
4	9	0.5625	[1, 1, 3]	1.1875	0.5033	1.7266
5	11	0.6875	[1, 1, 1, 2]	1.3125	0.4657	1.8021
6	13	0.8125	[1, 2, 2]	1.3125	0.4657	1.7708
7	15	0.9375	[1, 4]	1.1875	0.5033	1.6094

We notice that nested canalizing functions with higher absolute bias tend to be more sensitive to input changes and also less canalizing. However, the relationship between absolute bias and these other metrics is far from monotonic. Further, we notice that there is a perfect correlation between the average sensitivity of a nested canalizing function and its canalizing strength, and a near perfect correlation between average sensitivity and effective degree.

To investigate the non-monotonic behavior further, we can vary the degree and create line plots that reveal a clear pattern, as shown in Kadelka et al., Physica D, 2017.

```
ns = np.arange(5, 9)
fig, ax = plt.subplots()

for n in ns:
    all_hamming_weights = np.arange(1, 2 ** (n - 1), 2)
    all_abs_bias = 2 * np.abs(all_hamming_weights/2**n - 0.5)
    avg_sens = np.zeros(2 ** (n - 2))

    for i, w in enumerate(all_hamming_weights):
        layer = boolforge.utils.hamming_weight_to_ncf_layer_structure(n, w)
        f = boolforge.random_function(n, layer_structure=layer)
        avg_sens[i] = f.get_average_sensitivity(exact=True, normalized=False)
```

```
    ax.plot(all_abs_bias, avg_sens, "x--", label=f"n={n}")

ax.legend(frameon=False)
ax.set_xlabel("Absolute bias")
ax.set_ylabel("Average sensitivity")
plt.show()
```



5.7 Summary and outlook

This tutorial illustrated how ensembles of Boolean functions generated under explicit constraints reveal systematic relationships between canalization, bias, redundancy, and sensitivity.

The key findings include:

1. Canalization is rare in random functions but common in biology.
2. Canalizing strength and input redundancy both decrease with degree.
3. Functions with high absolute bias are more likely to be highly canalizing.
4. For NCFs, layer structure is uniquely determined by bias.
5. Average sensitivity varies systematically with layer structure.

These relationships constrain the space of biologically plausible functions and suggest evolutionary optimization for robustness.

We now move on to Boolean networks, where Boolean functions serve as node update rules and give rise to collective dynamical behavior.

6 Working with Boolean Networks

While previous tutorials focused on individual Boolean functions, this tutorial introduces Boolean networks, which combine multiple Boolean functions into a dynamical system.

6.1 What you will learn

In this tutorial you will learn how to:

- create Boolean networks,
- compute basic properties of the wiring diagram,
- compute basic properties of Boolean networks.

6.2 0. Setup

```
import boolforge
import numpy as np
import matplotlib.pyplot as plt
```

6.3 Boolean network theory

A Boolean network $F = (f_1, \dots, f_N)$ is a dynamical system consisting of N Boolean update functions. Each node can be in one of two states, 0 or 1, often interpreted as OFF/ON in biological contexts.

Under synchronous updating, all nodes update simultaneously, yielding a deterministic state transition graph on $\{0,1\}^N$.

Under asynchronous updating, only one node is updated at a time, yielding a stochastic transition graph. BoolForge implements both schemes.

Real biological networks are typically sparsely connected. The **in-degree** of a node is the number of essential inputs of its update function. The **wiring diagram** encodes which nodes regulate which others.

Despite their simplicity, Boolean networks can:

- reproduce complex dynamics (oscillations, multistability),
- predict gene knockout effects,
- identify control strategies,
- scale to genome-wide networks (1000s of nodes).

6.4 Wiring diagrams

We first construct wiring diagrams, which encode network structure independently of specific Boolean functions.

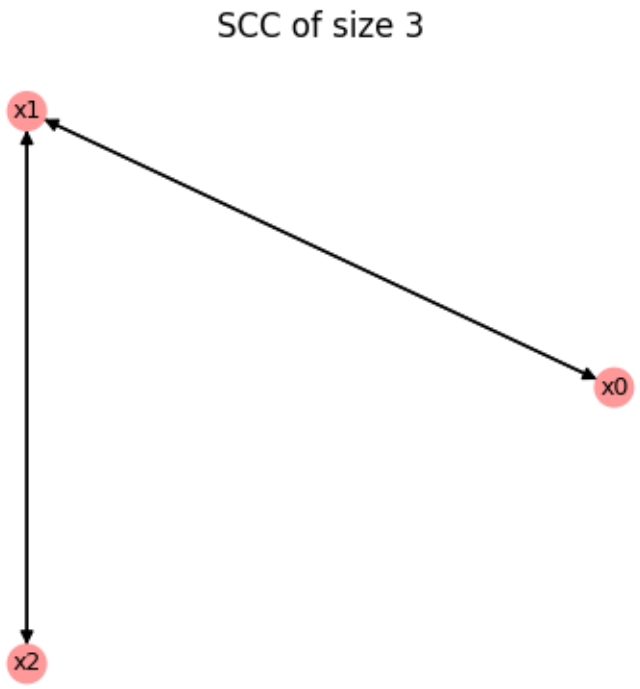
Separating topology (I) from dynamics (F) allows:

- studying structural properties independent of specific Boolean rules,
- swapping different rule sets on the same topology,
- efficient storage (sparse I, local F vs dense full truth table).

```
# Wiring diagram of a 3-node network
I = [
    [1],
    [0, 2],
    [1],
]

W = boolforge.WiringDiagram(I=I)
```

```
print("W.N:", W.N)
print("W.variables:", W.variables)
print("W.indegrees:", W.indegrees)
print("W.outdegrees:", W.outdegrees)

fig = W.plot(show=False);
```

```
W.N: 3
W.variables: ['x0' 'x1' 'x2']
W.indegrees: [1 2 1]
W.outdegrees: [1 2 1]
```



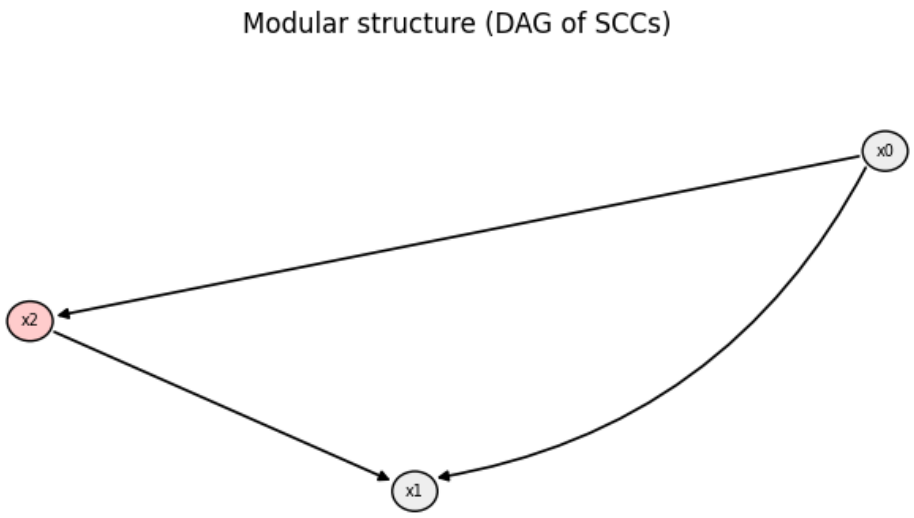
The wiring diagram above uses default variable names $x_0, \ldots, x_{N-1}$. The vectors `indegrees` and `outdegrees` describe incoming and outgoing edges for each node.

6.4.1 Example with constants and unequal degrees

```
I = [
    [],
    [0, 2],
    [0],
]

W = boolforge.WiringDiagram(I=I)

print("W.N:", W.N)
print("W.variables:", W.variables)
print("W.indegrees:", W.indegrees)
print("W.outdegrees:", W.outdegrees)
```

```
fig = W.plot(show=False)
```

```
W.N: 3
W.variables: ['x0' 'x1' 'x2']
W.indegrees: [0 2 1]
W.outdegrees: [2 0 1]
```



This wiring diagram encodes a **feed-forward loop**, one of the most common *network motifs* in transcriptional networks. It can:

- filter transient signals (coherent FFL with AND gate),
- accelerate response (incoherent FFL),

See Mangan & Alon, PNAS, 2003 for a detailed analysis. `BoolForge` enables the identification of all feed-forward loops:

```
print("W.get_ffls()", W.get_ffls())
```

```
W.get_ffls() {'FFLs': [[0, 2, 1]]}
```

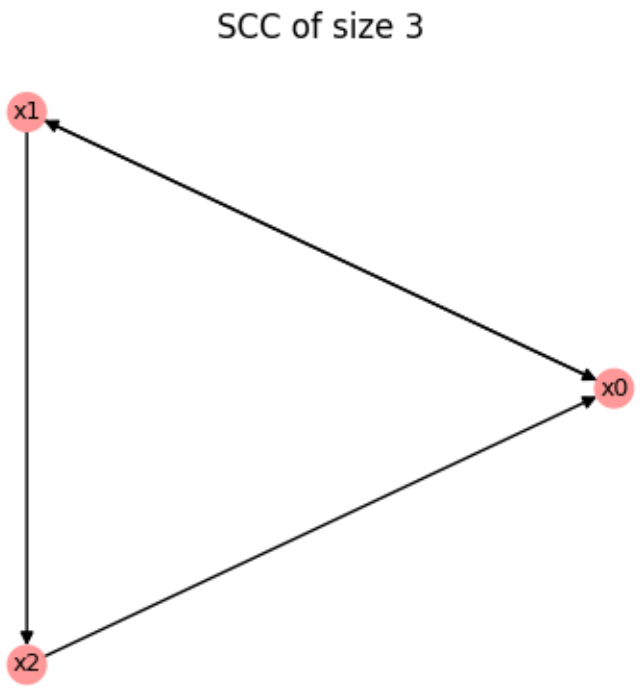
This tells us that `W` contains one FFL, in which x_0 regulates both x_1 and x_2, while x_1 is also regulated by x_2.

`BoolForge` can also identify all feedback loops. For this, we consider another wiring diagram:

```
I2 = [
    [2,1],
    [0],
    [1],
]

W2 = boolforge.WiringDiagram(I=I2)
fig = W2.plot(show=False)
fig
```

```
print("W2.get_fbls()", W2.get_fbls())
```

```
W2.get_fbls() {'FBLs': [[0, 1, 2], [0, 1]]}
```



The function `.get_fbls()` identifies all simple cycles in the wiring diagram. In this case, there exists a 2-cycle $x_0 \leftrightarrow x_1$ and a 3-cycle $x_0 \to x_1 \to x_2 \to x_0$.

6.5 Creating Boolean networks

To create a Boolean network, we must specify:

1. A wiring diagram `I`, describing who regulates whom.
2. A list `F` of Boolean update functions (or truth tables), one per node.

```
I = [
    [1],
    [0, 2],
    [1],
]

F = [
    [0, 1],
    [0, 1, 1, 1],
    [0, 1],
]

bn = boolforge.BooleanNetwork(F=F, I=I)

print(bn.to_truth_table().to_string())
```

```
   x0(t)  x1(t)  x2(t)  x0(t+1)  x1(t+1)  x2(t+1)
0      0      0      0        0        0        0
1      0      0      1        0        1        0
2      0      1      0        1        0        1
```

```
3      0      1      1        1        1        1
4      1      0      0        0        1        0
5      1      0      1        0        1        0
6      1      1      0        1        1        1
7      1      1      1        1        1        1
```

The full truth table of a Boolean network has size $N \times 2^N$ and therefore grows exponentially with the number of nodes.
In practice, however, `BoolForge` never stores this object explicitly. Instead, a Boolean network is represented internally by its wiring diagram `I` and the list of update functions `F`, which is far more memory-efficient – especially for sparse networks with few regulators per node.

When a Boolean network is constructed from `F` and `I`, `BoolForge` automatically performs a series of consistency checks to guard against common modeling errors. For example, it verifies that each update function has the correct length, namely 2^n, where n is the number of regulators of the corresponding node as specified in `I`. If any of these checks fail, an informative error is raised immediately, helping ensure that the resulting network is well-defined.

6.5.1 Creating networks from strings

Alternatively, Boolean networks can be specified using a human-readable string representation, where each line defines the update rule of one node. This format closely mirrors the way Boolean models are written in the literature and is often more convenient than manually specifying wiring diagrams and truth tables.

In the example below, each line has the form $x_i = f_i(\text{regulators of } x_i)$, where Boolean operators such as `AND`, `OR`, and `NOT` can be used to define the update functions.

```
string = """
x = y
y = x OR z
z = y
"""

bn_str = boolforge.BooleanNetwork.from_string(string, separator="=")
print(bn_str.to_truth_table().to_string())
```

```
   x(t)  y(t)  z(t)  x(t+1)  y(t+1)  z(t+1)
0     0     0     0       0       0       0
1     0     0     1       0       1       0
2     0     1     0       1       0       1
3     0     1     1       1       1       1
4     1     0     0       0       1       0
5     1     0     1       0       1       0
6     1     1     0       1       1       1
7     1     1     1       1       1       1
```

Here, the update rule `x = y` specifies that node `x` copies the state of `y`, while `y = x OR z` indicates that node `y` is regulated by both `x` and `z`. From this symbolic description, `BoolForge` automatically:

- extracts the wiring diagram,
- determines the regulators of each node,
- constructs the corresponding Boolean update functions.

Internally, the string representation is converted into the same `(F, I)` representation used throughout the package. As a result, Boolean networks created from strings behave identically to those created explicitly from wiring diagrams and truth tables.

This interface is particularly useful for loading Boolean network models from external sources, such as `.bnet` files, or for quickly prototyping models in an interactive setting.

6.5.2 Interoperability with CANA

`BoolForge` provides native interoperability with the `CANA` package for the analysis of Boolean functions and Boolean networks. Existing `BoolForge` networks can be converted into CANA objects and back without loss of information.

In the example below, we convert a `BoolForge` Boolean network into its CANA representation using `to_cana()`, and then reconstruct a new `BoolForge` Boolean network from that CANA object.

The final assertion verifies that this round-trip conversion preserves:

- the Boolean update functions,
- the wiring diagram,
- and the variable names.

This guarantees that `BoolForge` and CANA can be used interchangeably within a workflow, allowing users to leverage CANA's analytical tools while continuing to build and manipulate models using `BoolForge`.

```
cana_bn = bn.to_cana()
bn_from_cana = boolforge.BooleanNetwork.from_cana(cana_bn)

assert (
    np.all([np.all(bn.F[i].f == bn_from_cana.F[i].f) for i in range(bn.N)])
    and np.all([np.all(bn.I[i] == bn_from_cana.I[i]) for i in range(bn.N)])
    and np.all(bn.variables == bn_from_cana.variables)
), "BooleanNetwork CANA conversion failed"
```

6.6 Types of nodes in Boolean networks

Nodes in a Boolean network can be classified as follows:

- **Constant nodes**
  Nodes with constant update functions (always 0 or always 1). These nodes act as parameters and are removed internally, with their values propagated through the network.
- **Identity nodes**
  Nodes whose update function is the identity, i.e., $f(x_i) = x_i$. Their value is determined by the initial condition and remains constant over time. Identity nodes are retained as part of the Boolean network state. They may be viewed as nodes with a self-loop and no other incoming edges.
- **Regulated nodes**
  Nodes whose update functions depend on one or more other nodes.

```
F = [
    [0, 0, 0, 1],  # regulated
    [0, 1, 1, 1],  # regulated
    [0, 1],        # identity
    [0],           # constant
]

I = [
    [1, 2],        # regulated
    [0, 3],        # regulated
    [2],           # identity
    [],            # constant
]

bn = boolforge.BooleanNetwork(F, I)

print("bn.variables:", bn.variables)
```

```
print("bn.constants:", bn.constants)
print("bn.F:", bn.F)
print("bn.I:", bn.I)
```

```
bn.variables: ['x0' 'x1' 'x2']
bn.constants: {'x3': {'value': 0, 'regulatedNodes': ['x1']}}
bn.F: [BooleanFunction(f=[0, 0, 0, 1]), BooleanFunction(f=[0, 1]), BooleanFunction(f=[0, 1])]
bn.I: [array([1, 2]), array([0]), array([2])]
```

The constant node is removed, and its value is propagated into downstream update functions.

If we now change the value of the constant node from 0 to 1, the network is constructed in the same way, and the constant value 1 is substituted directly into all downstream update functions, before removal of the constant node.

As a result, the Boolean update functions of downstream nodes may simplify, potentially reducing the number of regulators or changing the logical form of the function. This illustrates how constant nodes act as parameters whose values influence the effective dynamics of the network.

Importantly, this simplification is performed symbolically at construction time and does not depend on the dynamical evolution of the network.

```
F = [
    [0, 0, 0, 1],
    [0, 1, 1, 1],
    [0, 1],
    [1],
]

I = [
    [1, 2],
    [0, 3],
    [2],
    [],
]

bn = boolforge.BooleanNetwork(F, I)

print("bn.F:", bn.F)
print("bn.I:", bn.I)
print("bn.variables:", bn.variables)
```

```
bn.F: [BooleanFunction(f=[0, 0, 0, 1]), BooleanFunction(f=[1, 1]), BooleanFunction(f=[0, 1])]
bn.I: [array([1, 2]), array([0]), array([2])]
bn.variables: ['x0' 'x1' 'x2']
```

Although x_1 becomes fixed at 1 after one update, it is not treated as a constant node. In `BoolForge`, constant nodes are identified by their update functions (always 0 or always 1), not by their long-term dynamical behavior. Since $x_1 = 0$ remains a valid initial condition, the node is retained as part of the network state.

6.7 Boolean network properties

The class `BooleanNetwork` inherits basic structural properties and methods from `WiringDiagram`. In particular, all graph-theoretic attributes of the wiring diagram – such as the number of nodes, in-degrees, and out-degrees – are directly accessible on a Boolean network object.

Moreover, `BooleanNetwork` inherits visualization utilities from `WiringDiagram`, including methods for plotting the wiring diagram and its modular structure, using `.plot()`. This allows users to inspect the topology of a Boolean network independently of the specific update functions.

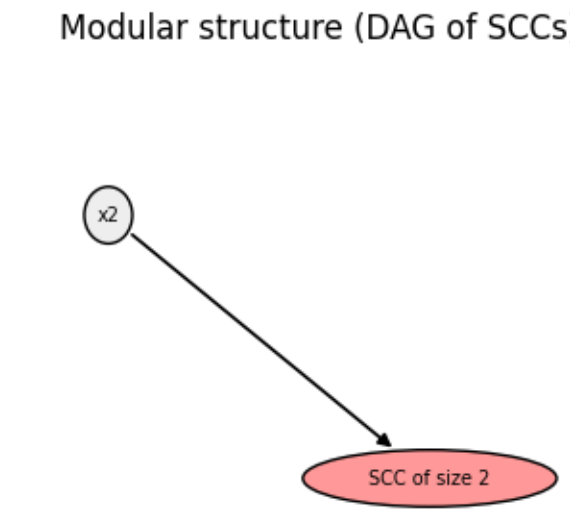
Beyond these inherited features, `BooleanNetwork` provides a rich collection of additional methods for analyzing the dynamics, structure, and control properties of Boolean networks. These include functionality for:

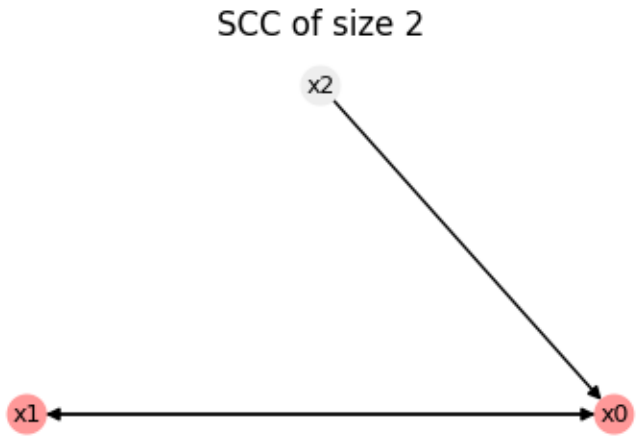
- computing fixed points and attractors,
- analyzing transient dynamics and state transition graphs,
- studying robustness and sensitivity to perturbations,
- performing node and edge interventions.

Many of these methods will be introduced and discussed in detail in the following tutorials. Here, we focus only on a few basic and commonly used properties.

```
print("bn.N:", bn.N)
print("bn.indegrees:", bn.indegrees)
print("bn.outdegrees:", bn.outdegrees)
print("bn.variables:", bn.variables)

bn.plot();
```

```
bn.N: 3
bn.indegrees: [2 1 1]
bn.outdegrees: [1 1 2]
bn.variables: ['x0' 'x1' 'x2']
```





6.8 Outlook

In the remaining tutorials, we build on this foundation to study the dynamical behavior of Boolean networks, including attractors, basins of attraction, and stability under perturbations.

7 Dynamics of Boolean Networks

In this tutorial, we study the *dynamics* of Boolean networks. Building on the construction and structural analysis from previous tutorials, we now focus on how Boolean networks evolve over time and how their long-term behavior can be characterized.

You will learn how to:

- simulate Boolean network dynamics under different updating schemes,
- compute and classify attractors,
- analyze basins of attraction,
- relate network structure to dynamical behavior.

7.1 Setup

```
import boolforge
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

7.2 State space of a Boolean network

A Boolean network with N nodes defines a dynamical system on the discrete state space $\{0,1\}^N$.

Each state is a binary vector

$$\mathbf{x} = (x_0, \dots, x_{N-1}) \in \{0,1\}^N,$$

where x_i denotes the state of node i.

We use a small Boolean network as a running example.

```
string = """
x = y
y = x OR z
z = y
"""

bn = boolforge.BooleanNetwork.from_string(string, separator="=")

print("Variables:", bn.variables)
print("N:", bn.N)
print("bn.I:", bn.I)
print("bn.F:", bn.F)
```

```
Variables: ['x' 'y' 'z']
N: 3
bn.I: [array([1]), array([0, 2]), array([1])]
bn.F: [BooleanFunction(f=[0, 1]), BooleanFunction(f=[0, 1, 1, 1]), BooleanFunction(f=[0, 1])]
```

All state vectors follow the variable order given by `bn.variables`. For small networks, we can enumerate all 2^N states explicitly.

```
all_states = boolforge.get_left_side_of_truth_table(bn.N)
pd.DataFrame(all_states, columns=bn.variables)
```

x

y

z
0
0
0
0
1
0
0
1
2
0
1
0
3
0
1
1
4
1
0
0
5
1
0
1
6
1
1
0
7
1
1
1

7.3 Dynamics of synchronous Boolean networks

Under *synchronous updating*, all nodes are updated simultaneously, defining a deterministic update map

$$\mathbf{x}(t+1) = F(\mathbf{x}(t)).$$

7.3.1 Exact computation

```
for state in all_states:
    print(state, "-->", bn.update_network_synchronously(state))
```

```
[0 0 0] --> [0 0 0]
[0 0 1] --> [0 1 0]
[0 1 0] --> [1 0 1]
[0 1 1] --> [1 1 1]
[1 0 0] --> [0 1 0]
[1 0 1] --> [0 1 0]
[1 1 0] --> [1 1 1]
[1 1 1] --> [1 1 1]
```

This output matches the synchronous truth table representation:

```
print(bn.to_truth_table().to_string())
```

```
   x(t)  y(t)  z(t)  x(t+1)  y(t+1)  z(t+1)
0     0     0     0       0       0       0
1     0     0     1       0       1       0
2     0     1     0       1       0       1
3     0     1     1       1       1       1
4     1     0     0       0       1       0
5     1     0     1       0       1       0
6     1     1     0       1       1       1
7     1     1     1       1       1       1
```

Each state has exactly one successor, so the dynamics consist of transient trajectories leading into *attractors* (steady states or cycles).

In this example, the network has:

- two steady states: $(0,0,0)$ and $(1,1,1)$,
- one cyclic attractor of length 2: $(0,1,0) \leftrightarrow (1,0,1)$.

7.3.2 Exhaustive attractor computation

```
dict_dynamics = bn.get_attractors_synchronous_exact()
dict_dynamics
```

```
{'Attractors': [[0], [2, 5], [7]],
 'NumberOfAttractors': 3,
 'BasinSizes': array([0.125, 0.5  , 0.375]),
 'AttractorID': array([0, 1, 1, 2, 1, 1, 2, 2], dtype=int32),
 'STG': array([0, 2, 5, 7, 2, 2, 7, 7])}
```

The returned dictionary contains:

- `STG`: the synchronous state transition graph,
- `NumberOfAttractors`,
- `Attractors`,
- `AttractorDict`,

- `BasinSizes`.

The state transition graph can be decoded as follows:

```
for state in range(2 ** bn.N):
    next_state = dict_dynamics["STG"][state]
    print(
        state,
        "=",
        boolforge.dec2bin(state, bn.N),
        "-->",
        next_state,
        "=",
        boolforge.dec2bin(next_state, bn.N),
    )
```

```
0 = [0, 0, 0] --> 0 = [0, 0, 0]
1 = [0, 0, 1] --> 2 = [0, 1, 0]
2 = [0, 1, 0] --> 5 = [1, 0, 1]
3 = [0, 1, 1] --> 7 = [1, 1, 1]
4 = [1, 0, 0] --> 2 = [0, 1, 0]
5 = [1, 0, 1] --> 2 = [0, 1, 0]
6 = [1, 1, 0] --> 7 = [1, 1, 1]
7 = [1, 1, 1] --> 7 = [1, 1, 1]
```

Attractors can be printed in binary representation:

```
for attractor in dict_dynamics["Attractors"]:
    print(f"Attractor of length {len(attractor)}:")
    for state in attractor:
        print(state, boolforge.dec2bin(state, bn.N))
    print()
```

```
Attractor of length 1:
0 [0, 0, 0]

Attractor of length 2:
2 [0, 1, 0]
5 [1, 0, 1]

Attractor of length 1:
7 [1, 1, 1]
```

Basin sizes count how many states flow into each attractor. They always sum to 2^N.

7.3.3 Monte Carlo simulation

For larger networks, exhaustive enumeration is infeasible. Monte Carlo simulation approximates the attractor landscape.

```
dict_dynamics = bn.get_attractors_synchronous(n_simulations=100)
dict_dynamics
```

```
{'Attractors': [[2, 5], [7], [0]],
 'NumberOfAttractors': 3,
 'BasinSizes': [50, 38, 12],
 'AttractorDict': {2: 0, 5: 0, 7: 1, 1: 0, 4: 0, 3: 1, 0: 2, 6: 1},
 'InitialSamplePoints': [2,
  7,
```

```
5,
1,
7,
5,
1,
1,
4,
3,
0,
2,
7,
4,
5,
4,
5,
1,
0,
4,
2,
4,
4,
6,
0,
6,
3,
4,
6,
4,
7,
3,
3,
6,
6,
6,
6,
7,
1,
6,
3,
0,
2,
0,
4,
7,
6,
3,
3,
5,
5,
5,
6,
4,
1,
2,
```

```
  1,
  4,
  0,
  0,
  5,
  6,
  0,
  6,
  2,
  1,
  5,
  7,
  7,
  0,
  7,
  7,
  1,
  6,
  7,
  7,
  0,
  1,
  2,
  0,
  2,
  0,
  2,
  1,
  1,
  7,
  1,
  2,
  2,
  1,
  7,
  7,
  5,
  7,
  2,
  2,
  3,
  1,
  7,
  1],
 'STG': {2: 5, 7: 7, 5: 2, 1: 2, 4: 2, 3: 7, 0: 0, 6: 7},
 'NumberOfTimeouts': 0}
```

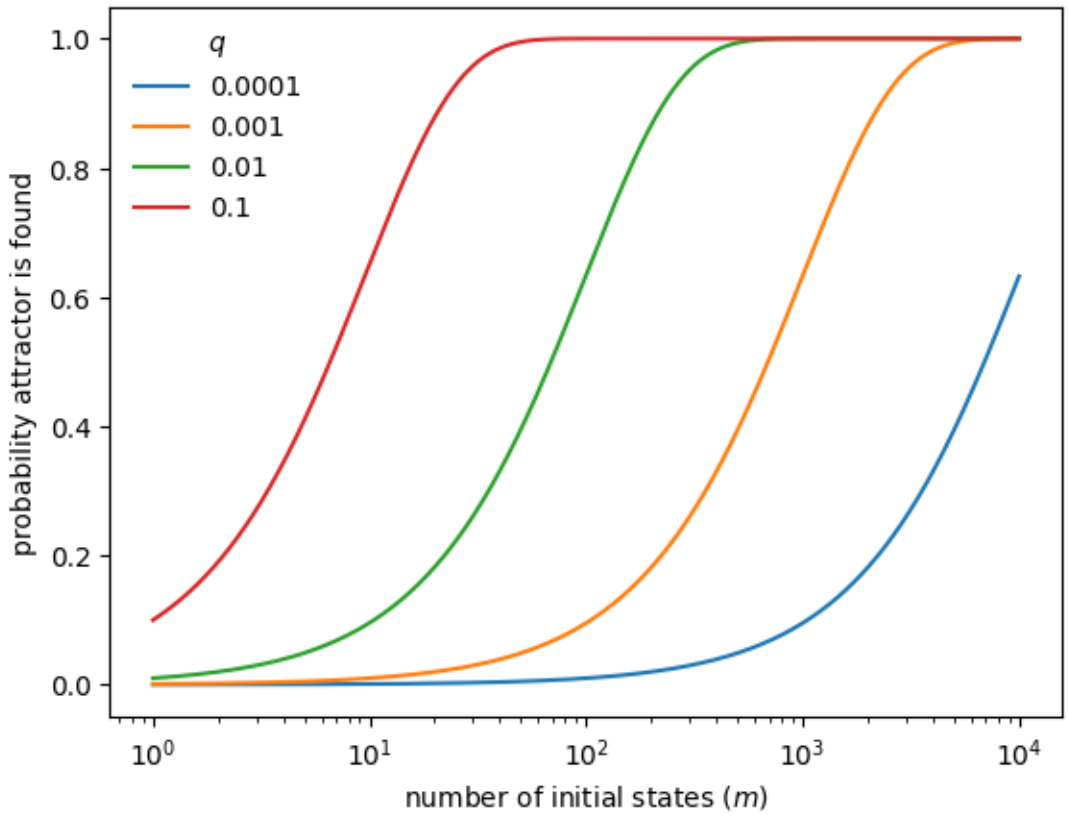
The simulation returns additional information:

- sampled initial states,
- the number of timeouts (trajectories not reaching an attractor in time).

If an attractor has relative basin size q, the probability that it is found after m random initializations is $1-(1-q)^m$.

```
qs = [0.0001, 0.001, 0.01, 0.1]
```

```
ms = np.logspace(0, 4, 1000)

fig, ax = plt.subplots()
for q in qs:
    ax.semilogx(ms, 1 - (1 - q) ** ms, label=str(q))

ax.legend(title=r"$q$", frameon=False)
ax.set_xlabel("number of initial states ($m$)")
ax.set_ylabel("probability attractor is found")
plt.show()
```



7.4 Dynamics of asynchronous Boolean networks

Synchronous updating is computationally convenient but biologically unrealistic. Asynchronous updating assumes that only one node is updated at a time.

7.4.1 Steady states under general asynchronous update

BoolForge can compute steady states under general asynchronous updating.

```
dict_dynamics = bn.get_steady_states_asynchronous_exact()
dict_dynamics
```

```
{'SteadyStates': [0, 7],
 'NumberOfSteadyStates': 2,
 'BasinSizes': array([0.33333333, 0.66666667]),
 'STGAsynchronous': {0: {0: np.float64(1.0)},
  1: {1: np.float64(0.3333333333333333),
```

```
   3: np.float64(0.3333333333333333),
   0: np.float64(0.3333333333333333)},
  2: {6: np.float64(0.3333333333333333),
   0: np.float64(0.3333333333333333),
   3: np.float64(0.3333333333333333)},
  3: {7: np.float64(0.3333333333333333), 3: np.float64(0.6666666666666666)},
  4: {0: np.float64(0.3333333333333333),
   6: np.float64(0.3333333333333333),
   4: np.float64(0.3333333333333333)},
  5: {1: np.float64(0.3333333333333333),
   7: np.float64(0.3333333333333333),
   4: np.float64(0.3333333333333333)},
  6: {6: np.float64(0.6666666666666666), 7: np.float64(0.3333333333333333)},
  7: {7: np.float64(1.0)}},
 'FinalTransitionProbabilities': array([[1.        , 0.        ],
        [0.5       , 0.5       ],
        [0.33333333, 0.66666667],
        [0.        , 1.        ],
        [0.5       , 0.5       ],
        [0.33333333, 0.66666667],
        [0.        , 1.        ],
        [0.        , 1.        ]])}
```

This reveals the same two steady states as in the synchronous case. In addition, the full asynchronous transition graph and absorption probabilities are returned.

BoolForge currently does not detect complex cyclic attractors under asynchronous updating; for those, specialized tools such as `pystablemotifs` are recommended.

7.4.2 Monte Carlo approximation

```
dict_dynamics = bn.get_steady_states_asynchronous(n_simulations=500)
dict_dynamics
```

```
{'SteadyStates': [7, 0],
 'NumberOfSteadyStates': 2,
 'BasinSizes': [344, 156],
 'STGAsynchronous': {(6, 1): 6,
  (6, 0): 6,
  (6, 2): 7,
  (7, 1): 7,
  (7, 0): 7,
  (7, 2): 7,
  (4, 0): 0,
  (0, 1): 0,
  (0, 2): 0,
  (0, 0): 0,
  (4, 1): 6,
  (5, 2): 4,
  (2, 2): 3,
  (3, 0): 7,
  (3, 1): 3,
  (3, 2): 3,
  (4, 2): 4,
  (1, 2): 0,
  (1, 0): 1,
```

```
 (2, 0): 6,
 (1, 1): 3,
 (5, 0): 1,
 (5, 1): 7,
 (2, 1): 0},
'InitialSamplePoints': [6,
 4,
 4,
 0,
 5,
 2,
 3,
 2,
 7,
 4,
 3,
 0,
 7,
 1,
 4,
 1,
 0,
 7,
 7,
 1,
 1,
 6,
 2,
 4,
 1,
 1,
 2,
 1,
 4,
 3,
 2,
 7,
 7,
 5,
 5,
 3,
 0,
 0,
 7,
 1,
 3,
 3,
 6,
 3,
 1,
 4,
 2,
 4,
 6,
```

```
2,
2,
0,
5,
0,
2,
5,
2,
3,
1,
0,
4,
1,
6,
4,
3,
3,
7,
4,
1,
6,
5,
7,
2,
5,
6,
4,
7,
0,
3,
2,
6,
4,
6,
4,
5,
5,
6,
2,
1,
0,
2,
2,
6,
4,
2,
2,
2,
0,
7,
7,
6,
7,
3,
```

```
2,
4,
0,
4,
3,
1,
1,
4,
5,
6,
4,
5,
5,
2,
3,
3,
5,
4,
5,
2,
3,
0,
0,
1,
1,
6,
4,
7,
3,
7,
6,
0,
0,
5,
1,
1,
4,
5,
1,
4,
6,
5,
3,
6,
0,
4,
0,
7,
4,
0,
5,
7,
0,
6,
```

```
0,
3,
1,
6,
3,
2,
1,
3,
3,
4,
7,
0,
0,
6,
7,
5,
7,
5,
2,
3,
7,
7,
5,
3,
0,
5,
5,
7,
6,
0,
2,
4,
2,
2,
0,
7,
0,
5,
3,
6,
2,
5,
2,
2,
7,
2,
6,
3,
5,
6,
7,
6,
3,
7,
```

```
4,
6,
7,
6,
3,
4,
1,
3,
3,
2,
7,
7,
3,
7,
2,
1,
1,
3,
6,
3,
1,
6,
0,
4,
0,
4,
6,
5,
2,
1,
6,
2,
2,
0,
7,
5,
7,
1,
2,
3,
6,
2,
1,
7,
3,
2,
2,
2,
6,
7,
4,
4,
3,
2,
```

```
7,
1,
3,
0,
5,
3,
5,
7,
6,
0,
7,
1,
6,
7,
3,
0,
0,
0,
7,
1,
0,
5,
1,
2,
5,
2,
3,
4,
3,
6,
4,
5,
3,
3,
0,
2,
4,
7,
2,
1,
4,
0,
0,
3,
4,
5,
3,
1,
6,
2,
5,
0,
7,
6,
```

```
6,
2,
1,
2,
1,
2,
3,
0,
4,
4,
3,
7,
0,
1,
6,
2,
7,
7,
3,
6,
0,
2,
1,
1,
5,
6,
0,
5,
4,
7,
2,
7,
5,
6,
0,
6,
6,
4,
5,
7,
6,
5,
5,
2,
2,
5,
3,
2,
1,
3,
7,
2,
3,
2,
```

```
0,
7,
6,
3,
0,
1,
4,
0,
5,
7,
0,
2,
2,
2,
6,
4,
0,
1,
4,
1,
3,
1,
7,
0,
6,
6,
1,
4,
6,
5,
5,
7,
1,
1,
5,
5,
3,
3,
6,
5,
5,
3,
5,
5,
2,
6,
0,
5,
4,
7,
3,
3,
2,
7,
```

```
6,
5,
5,
4,
6,
7,
5,
5,
5,
3,
3,
7,
7,
4,
6,
7,
7,
0,
5,
7,
1,
0,
7,
2,
0,
0,
7,
3,
6,
3,
2,
6,
4,
7,
4,
0,
5,
0,
1,
2,
0,
0,
1,
6,
5,
6,
5,
4,
2,
5,
3,
7,
3,
1,
```

```
  6,
  5,
  7,
  5,
  7,
  5,
  2,
  6,
  3,
  0,
  1,
  4,
  7,
  0,
  2,
  4,
  4,
  5,
  1]}
```

The simulation provides:

- a lower bound on the number of steady states,
- approximate basin size distributions,
- samples of the asynchronous state transition graph.

7.4.3 Sampling from a fixed initial condition

```
dict_dynamics = bn.get_steady_states_asynchronous_given_one_initial_condition(
    initial_condition=[0, 0, 1], n_simulations=500
)
dict_dynamics
```

```
{'SteadyStates': [0, 7, 9, 2],
 'NumberOfSteadyStates': 4,
 'BasinSizes': [246, 132, 36, 86],
 'TransientTimes': [[1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
   1,
```

```
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
```

```
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
```

```
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
```

```
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
1,
```

```
 1,
 1,
 1,
 1,
 1,
 1,
 1,
 1,
 1,
 1],
[2,
 3,
 2,
 2,
 2,
 3,
 3,
 2,
 2,
 2,
 2,
 2,
 2,
 3,
 2,
 2,
 2,
 3,
 2,
 3,
 2,
 2,
 3,
 3,
 2,
 3,
 2,
 2,
 3,
 2,
 2,
 2,
 3,
 3,
 2,
 3,
 2,
 3,
 3,
 2,
 3,
 2,
 3,
 2,
 2,
```

3,
2,
3,
3,
2,
2,
3,
3,
2,
2,
3,
3,
3,
2,
3,
3,
2,
3,
3,
3,
2,
2,
3,
2,
2,
2,
2,
2,
2,
3,
3,
2,
3,
3,
3,
3,
3,
2,
2,
3,
2,
3,
2,
2,
2,
3,
3,
2,
2,
3,
2,
2,
2,
2,
2,
2,

```
 2,
 2,
 3,
 2,
 3,
 2,
 2,
 2,
 2,
 3,
 2,
 2,
 2,
 2,
 3,
 3,
 2,
 3,
 3,
 2,
 2,
 2,
 2,
 2,
 2,
 3,
 3,
 2,
 2,
 2,
 3,
 3,
 2,
 3],
[3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
```

```
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3],
[2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
 2,
```

```
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2,
  2]],
'STGAsynchronous': {(1, 2): 0,
 (0, 2): 0,
 (0, 1): 0,
 (0, 0): 0,
 (1, 1): 3,
 (3, 0): 7,
```

```
 (7, 2): 7,
 (7, 1): 7,
 (7, 0): 7,
 (1, 0): 1,
 (3, 1): 5,
 (5, 2): 5,
 (5, 0): 9,
 (9, 0): 9,
 (9, 2): 9,
 (9, 1): 9,
 (3, 2): 2,
 (2, 0): 2,
 (2, 1): 2,
 (2, 2): 2,
 (5, 1): 7},
'UpdateQueues': [[1, 0],
 [1, 0],
 [1, 3, 7],
 [1, 0],
 [1, 3, 5, 9],
 [1, 0],
 [1, 0],
 [1, 3, 2],
 [1, 0],
 [1, 0],
 [1, 0],
 [1, 0],
 [1, 3, 5, 7],
 [1, 3, 2],
 [1, 3, 5, 9],
 [1, 3, 2],
 [1, 3, 7],
 [1, 3, 7],
 [1, 3, 2],
 [1, 0],
 [1, 3, 7],
 [1, 0],
 [1, 3, 5, 7],
 [1, 3, 5, 7],
 [1, 3, 7],
 [1, 0],
 [1, 0],
 [1, 3, 7],
 [1, 3, 7],
 [1, 0],
 [1, 0],
 [1, 3, 7],
 [1, 3, 5, 9],
 [1, 0],
 [1, 0],
 [1, 3, 7],
 [1, 0],
 [1, 0],
 [1, 0],
```

```
[1, 3, 2],
[1, 3, 5, 7],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 3, 5, 7],
[1, 3, 5, 9],
[1, 3, 2],
[1, 3, 5, 9],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 3, 5, 9],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 3, 5, 9],
[1, 0],
[1, 3, 2],
[1, 3, 5, 7],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 5, 7],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 3, 5, 7],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 3, 5, 9],
[1, 0],
```

```
[1, 0],
[1, 0],
[1, 3, 7],
[1, 3, 5, 7],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 5, 9],
[1, 3, 7],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 3, 5, 9],
[1, 3, 2],
[1, 3, 5, 9],
[1, 3, 5, 7],
[1, 3, 2],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 3, 7],
[1, 3, 5, 7],
[1, 0],
[1, 3, 7],
[1, 3, 5, 7],
[1, 3, 2],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 3, 2],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 3, 2],
[1, 3, 7],
[1, 3, 5, 9],
[1, 3, 2],
[1, 3, 5, 7],
[1, 0],
[1, 0],
```

```
[1, 0],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 5, 9],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 3, 5, 9],
[1, 0],
[1, 3, 5, 9],
[1, 0],
[1, 3, 2],
[1, 3, 5, 7],
[1, 0],
[1, 3, 2],
[1, 3, 5, 9],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 3, 5, 7],
[1, 3, 5, 7],
[1, 3, 7],
[1, 3, 2],
[1, 3, 5, 9],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 3, 7],
[1, 3, 5, 7],
[1, 0],
[1, 0],
[1, 3, 5, 9],
[1, 3, 5, 7],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 3, 5, 7],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
```

```
[1, 3, 5, 7],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 5, 9],
[1, 3, 5, 9],
[1, 0],
[1, 3, 5, 9],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 7],
[1, 0],
[1, 3, 5, 7],
[1, 3, 5, 7],
[1, 3, 5, 9],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 5, 7],
[1, 3, 7],
[1, 3, 2],
[1, 0],
[1, 3, 7],
[1, 3, 5, 7],
[1, 3, 5, 9],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 5, 9],
[1, 3, 7],
[1, 3, 7],
[1, 3, 7],
[1, 3, 7],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 3, 2],
[1, 0],
[1, 3, 5, 7],
[1, 0],
```

```
[1, 0],
[1, 0],
[1, 0],
[1, 3, 5, 9],
[1, 3, 2],
[1, 3, 2],
[1, 0],
[1, 3, 2],
[1, 3, 2],
[1, 3, 5, 7],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 5, 7],
[1, 3, 5, 9],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 3, 5, 7],
[1, 3, 2],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 3, 2],
[1, 3, 7],
[1, 3, 2],
[1, 3, 5, 7],
[1, 3, 7],
[1, 3, 5, 7],
[1, 3, 7],
[1, 3, 7],
[1, 3, 2],
[1, 0],
[1, 3, 2],
[1, 3, 2],
[1, 0],
[1, 3, 5, 7],
[1, 3, 2],
[1, 0],
[1, 3, 5, 9],
[1, 0],
[1, 3, 5, 7],
```

```
[1, 0],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 7],
[1, 0],
[1, 3, 5, 7],
[1, 3, 5, 9],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 7],
[1, 0],
[1, 3, 2],
[1, 3, 5, 9],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 3, 7],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 2],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 2],
[1, 3, 7],
```

```
[1, 0],
[1, 3, 7],
[1, 3, 2],
[1, 3, 2],
[1, 3, 5, 7],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 5, 9],
[1, 3, 7],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 3, 2],
[1, 3, 7],
[1, 3, 2],
[1, 3, 7],
[1, 3, 2],
[1, 3, 5, 7],
[1, 3, 2],
[1, 3, 7],
[1, 3, 7],
[1, 3, 5, 9],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 3, 7],
[1, 3, 5, 7],
[1, 0],
[1, 0],
[1, 3, 5, 9],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 2],
[1, 0],
[1, 3, 2],
```

```
[1, 0],
[1, 3, 2],
[1, 3, 2],
[1, 0],
[1, 3, 5, 9],
[1, 0],
[1, 3, 5, 9],
[1, 0],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 3, 2],
[1, 3, 7],
[1, 0],
[1, 3, 5, 7],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 3, 2],
[1, 3, 2],
[1, 3, 5, 7],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 3, 7],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 7],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 2],
[1, 0],
[1, 0],
[1, 0],
[1, 0],
[1, 3, 5, 9],
[1, 0],
[1, 3, 5, 7],
[1, 0],
[1, 0],
```

```
 [1, 3, 2],
 [1, 0],
 [1, 3, 5, 7],
 [1, 3, 7],
 [1, 0],
 [1, 0],
 [1, 3, 2],
 [1, 3, 7],
 [1, 3, 7],
 [1, 0],
 [1, 3, 5, 9],
 [1, 0],
 [1, 3, 2],
 [1, 3, 5, 7],
 [1, 0],
 [1, 3, 2],
 [1, 0],
 [1, 0],
 [1, 0],
 [1, 0],
 [1, 3, 5, 7],
 [1, 0],
 [1, 3, 2],
 [1, 3, 2],
 [1, 3, 7],
 [1, 3, 5, 7],
 [1, 3, 2],
 [1, 0],
 [1, 0]]}
```

7.5 Summary and outlook

In this tutorial you learned how to:

- simulate Boolean network dynamics,
- compute synchronous attractors exactly and approximately,
- analyze basin sizes,
- compute steady states under asynchronous updating.

This concludes the function- and network-level analysis. Subsequent work focuses on large-scale dynamical analysis, perturbations, and stability in Boolean network models.

8 Perturbation and sensitivity analysis of Boolean networks

In this tutorial, we study how Boolean networks respond to perturbations. Rather than implementing perturbations manually, we leverage BoolForge's built-in robustness and sensitivity measures.

You will learn how to: - quantify robustness and fragility of Boolean networks under synchronous update, - interpret basin-level and attractor-level robustness measures, - compare exact and approximate robustness computations, and - compute Derrida values as a measure of dynamical sensitivity.

These tools allow us to assess dynamical stability and resilience of Boolean network models in a principled and computationally efficient way.

8.1 Setup

```
import boolforge
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

8.2 A running example Boolean network

We reuse the small Boolean network from the previous tutorial.

```
string = """
x = y
y = x OR z
z = y
"""

bn = boolforge.BooleanNetwork.from_string(string, separator="=")

print("Variables:", bn.variables)
print("Number of nodes:", bn.N)
```

```
Variables: ['x' 'y' 'z']
Number of nodes: 3
```

8.3 Exact attractors and robustness measures

BoolForge provides a single method that computes: - all attractors, - basin sizes, - overall network coherence and fragility, - basin-level coherence and fragility, and - attractor-level coherence and fragility.

These quantities are defined via systematic single-bit perturbations in the Boolean hypercube and can be computed *exactly* for small networks.

```
results_exact = bn.get_attractors_and_robustness_synchronous_exact()
results_exact.keys()
```

```
dict_keys(['Attractors', 'NumberOfAttractors', 'BasinSizes', 'AttractorID', 'Coherence', 'Fragility', 'BasinCoherenc
```

```
print("Number of attractors:", results_exact["NumberOfAttractors"])
print("Attractors (decimal states):", results_exact["Attractors"])
print("Eventual attractor:", results_exact["AttractorID"])

print("Basin sizes:", results_exact["BasinSizes"])
print("Overall coherence:", results_exact["Coherence"])
print("Overall fragility:", results_exact["Fragility"])
```

```
Number of attractors: 3
Attractors (decimal states): [[0], [2, 5], [7]]
Eventual attractor: [0 1 1 2 1 1 2 2]
Basin sizes: [0.125 0.5   0.375]
Overall coherence: 0.3333333333333333
Overall fragility: 0.3333333333333333
```

8.4 Basin-level and attractor-level robustness

Robustness can be resolved at different structural levels. We now inspect basin-specific and attractor-specific measures.

```
df_basins = pd.DataFrame({
    "BasinSize": results_exact["BasinSizes"],
    "BasinCoherence": results_exact["BasinCoherence"],
    "BasinFragility": results_exact["BasinFragility"],
})

df_attractors = pd.DataFrame({
    "AttractorCoherence": results_exact["AttractorCoherence"],
    "AttractorFragility": results_exact["AttractorFragility"],
})

print("Basin-level robustness:")
print(df_basins)

print("Attractor-level robustness:")
print(df_attractors)
```

```
Basin-level robustness:
   BasinSize  BasinCoherence  BasinFragility
0      0.125        0.000000        0.500000
1      0.500        0.333333        0.333333
2      0.375        0.444444        0.277778
Attractor-level robustness:
   AttractorCoherence  AttractorFragility
0            0.000000            0.500000
1            0.333333            0.333333
2            0.666667            0.166667
```

Interpretation:

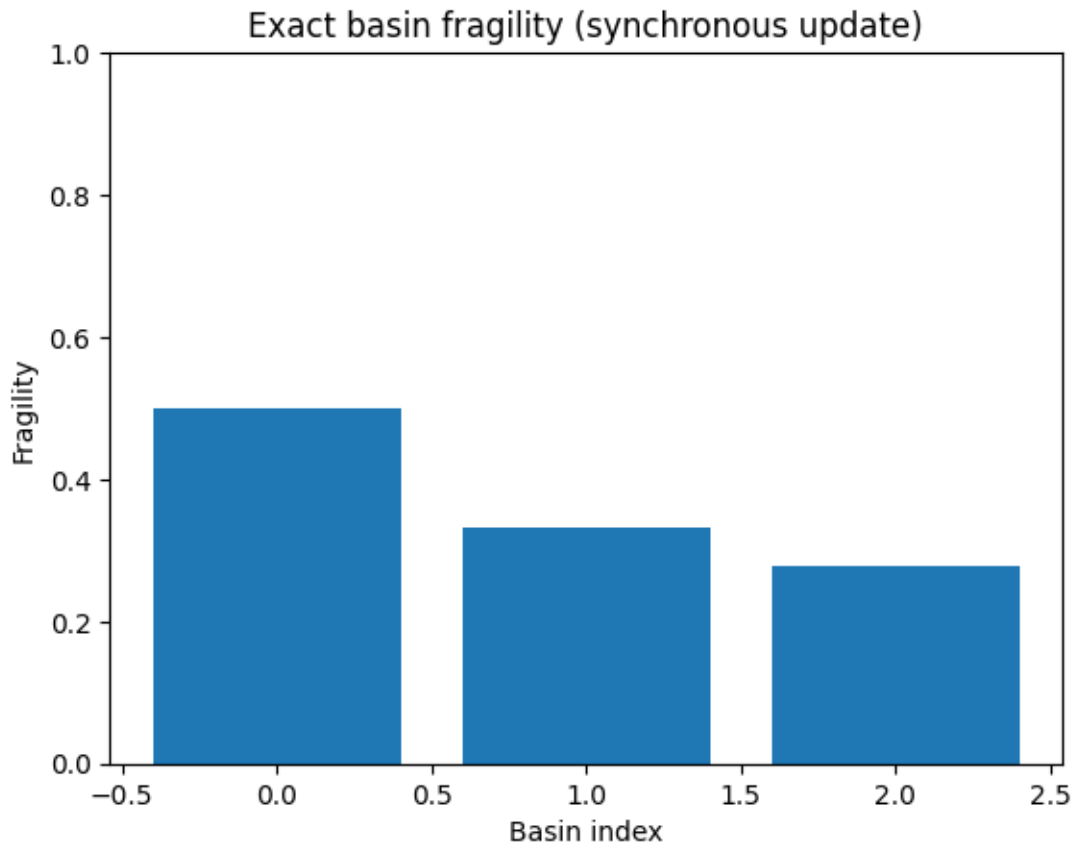
- **Coherence** measures the fraction of single-bit perturbations that do *not* change the final attractor.
- **Fragility** measures how much the attractor state changes *when* a perturbation does lead to a different attractor.

Importantly, attractors are often less stable than their basins, a phenomenon explored in detail in Tutorial #10.

8.5 Visualization of basin robustness

```
fig, ax = plt.subplots()

ax.bar(
    np.arange(len(results_exact["BasinSizes"])),
    results_exact["BasinFragility"],
    label="Basin fragility",
```

```
)
ax.set_xlabel("Basin index")
ax.set_ylabel("Fragility")
ax.set_title("Exact basin fragility (synchronous update)")
ax.set_ylim(0, 1)

plt.show()
```



8.6 Approximate robustness for larger networks

For larger networks, exact enumeration of all 2^N states is infeasible. BoolForge therefore provides a Monte Carlo approximation that samples random initial conditions and perturbations.

```
results_approx = bn.get_attractors_and_robustness_synchronous(
    n_simulations=500
)

results_approx.keys()
```

```
dict_keys(['Attractors', 'LowerBoundOfNumberOfAttractors', 'BasinSizesApproximation', 'CoherenceApproximation', 'Fr
```

```
print("Lower bound on number of attractors:", results_approx["LowerBoundOfNumberOfAttractors"])
print("Approximate coherence:", results_approx["CoherenceApproximation"])
print("Approximate fragility:", results_approx["FragilityApproximation"])
print("Final Hamming distance approximation:",
      results_approx["FinalHammingDistanceApproximation"])
```

```
Lower bound on number of attractors: 3
```

```
Approximate coherence: 0.322
Approximate fragility: 0.33899999999999997
Final Hamming distance approximation: 0.339
```

Even for this small network, the approximate values closely match the exact ones. For larger networks, these approximations are often the only feasible option.

8.7 Derrida value: dynamical sensitivity

The Derrida value measures how perturbations *propagate* after one synchronous update. It is defined as the expected Hamming distance between updated states that initially differed in exactly one bit.

```
derrida_exact = bn.get_derrida_value(exact=True)
derrida_approx = bn.get_derrida_value(n_simulations=2000)

print("Exact Derrida value:", derrida_exact)
print("Approximate Derrida value:", derrida_approx)
```

```
Exact Derrida value: 1.0
Approximate Derrida value: 1.0
```

Interpretation:

- Small Derrida values indicate ordered, stable dynamics.
- Large Derrida values indicate sensitive or chaotic dynamics.

Derrida values are closely related to average sensitivity of the update functions, and provide a complementary notion of robustness to basin-based measures.

8.8 Summary and outlook

In this tutorial you learned how to: - compute exact robustness measures for small Boolean networks, - interpret coherence and fragility at network, basin, and attractor levels, - approximate robustness measures for larger networks, and - assess dynamical sensitivity using the Derrida value.

Next steps: In Tutorial 9, we will move from global robustness measures to *trajectory-based* sensitivity analysis, including damage spreading, Hamming distance dynamics, and time-resolved perturbation experiments.