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: ['positive' 'positive' 'conditional']
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
```

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 (list of 0s and 1s),
- 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 (e.g., truth table to/from polynomial 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.

2 Frequently Asked Questions (FAQ)

2.1 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` corresponds to the binary representation of `i` over `(A, B, C)`.

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

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

2.2 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`).

2.3 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.

2.4 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.is_constant()`, `f.is_degenerate()`
Get variable info	`f.variables`, `f.n`
Convert representations	`f.to_logical()`, `f.to_polynomial()`

3 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.

3.1 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.

3.2 Setup

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

3.3 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)

3.3.1 What is a symmetry?

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.

3.4 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, such functions are extremely rare unless intentionally created.

BoolForge therefore:

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

3.5 Activities and Sensitivities

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

3.5.1 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 $e_i = (0, \dots, 0, 1, 0, \dots, 0)$ 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.

3.5.2 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.

3.5.3 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)

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

```
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 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 set `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: 0.4997
Normalized average sensitivity: 0.5003
```

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).

3.6 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.