#02: 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.

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.

---

0. Setup

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

---

1. 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)

1.1 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.

1.2 Examples

Below we define three Boolean functions demonstrating 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. 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. Activities and Sensitivities

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

3.1 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.

3.2 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.

3.3 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)

3.4 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, n
print("Normalized average sensitivity of g:", g.get_average_sensitivity(exact=exact, n
```

```
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.

Example: random 25-input function

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: 0.5009
Normalized average sensitivity: 0.4999
```

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).

---

4. 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.