



Prophecy for CIB

Manual

Version 2.0.0

08 June 2026

Consistent with Prophecy version 2.0.0

To cite:

Kearney, Norman. M. 2026. *Prophecy for CIB: Manual*. Version 2.0.0.

<https://doi.org/10.5281/zenodo.18986525>



2026 Norman M. Kearney
norman.kearney@unibe.ch

This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International](https://creativecommons.org/licenses/by-sa/4.0/) license.

Table of Contents

INTRODUCTION	1
PART 1: CIB FUNDAMENTALS	4
1.1 DESCRIPTORS, VARIANTS, AND CROSS-IMPACTS	4
1.2 MATRICES, JUDGMENT SECTIONS, JUDGMENT GROUPS, AND IMPACT COLUMNS	4
1.3 RELATIVE PLAUSIBILITY OF CROSS-IMPACTS	5
1.4 SCENARIOS, BALANCES, AND CONSISTENCY	6
1.5 SUCCESSORS AND SUCCESSION ANALYSIS	7
1.6 RELATIVE STRENGTH OF CROSS-IMPACTS	10
1.7 RULES OF THUMB FOR ASSIGNING CROSS-IMPACTS	10
1.9 PERTURBATION ANALYSIS	10
PART 2: PROPHECY QUICK START GUIDE	12
2.1 INSTALL AND IMPORT	12
2.2 CREATE A PROJECT	13
2.3 LOAD A MODEL FROM A MATRIX CSV	14
2.3.1 <i>Alternative: build a model from a list CSV</i>	15
2.4 CONFIGURE DESCRIPTOR METADATA	17
2.5 CONFIGURE SUCCESSION	20
2.6 MODEL VERSIONS: GENERATE OR LOAD	20
2.7 SCENARIOS: GENERATE OR LOAD	26
2.8 LOAD A SCORING PROFILE	30
2.9 RUN SCENARIO ANALYSIS	30
2.10 RUN PERTURBATION ANALYSIS	32
2.11 RUN PATHWAY ANALYSIS	37
2.12 RESTART AN INTERRUPTED ANALYSIS	39
3. ACKNOWLEDGEMENTS	42
4. BIBLIOGRAPHY	43

Introduction

Cross-Impact Balances (CIB) analysis was designed for systematically generating scenarios and assessing them for internal consistency. Through succession analysis, CIB also “offer[s] an idea of the direction in which the system might move from its momentary state” (Weimer-Jehle 2006). In support of this goal, Prophecy introduces several improvements to succession analysis related to ties, succession rules, and auxiliary descriptors.

Two or more variants of a descriptor can be tied for the highest balance. The traditional approach to this problem has been to select the variant with the lowest index, which biases the successions. Prophecy gives the user control over how ties are broken. The legacy tiebreaker handles ties in the way described above. An advantage of it is that it ensures deterministic successions. The random tiebreaker selects one of the tied variants at random. The conservative tiebreaker handles two cases: (i) when the current variant is one of the tied variants, the current variant is selected; (ii) otherwise, the random tiebreaker is used.

The conservative and random tiebreakers introduce stochasticity into succession analysis. Since the outcome of succession analysis is no longer deterministic, Prophecy allows the user to run multiple succession trials per initial scenario. The modal outcome is then taken as the outcome for the tested scenario, be it a point attractor or a cyclical attractor. Stochasticity also complicates the identification of cyclical attractors, since they can be noisy. Prophecy includes strategies for identifying cyclical attractors in noisy successions.

Succession rules govern how descriptors are corrected at each step in a succession analysis. The four most widely used rules are global (simultaneously update each descriptor to its variant with the highest balance), local (update only the descriptor with the greatest inconsistency), incremental (simultaneously update all descriptors, but restrict their movement to an adjacent variant), and adiabatic (cycle through the descriptors in order of their indices, updating only one descriptor at a time) (Weimer-Jehle 2026). Stochastic rules have also been developed (Schweizer et al. 2023). While these rules are suitable for identifying attractors, they are less ideal for modelling system dynamics. Real systems can have multiple processes running in parallel, which neither the local nor the adiabatic rule can represent. Some system elements may undergo radical changes, while others may change incrementally; the incremental rule cannot represent both behaviours. Prophecy implements the global succession rule and devolves decision-making about descriptor behaviour to the descriptors themselves. It does this by introducing descriptor

types. Nominal descriptors can move directly from the current variant to the variant with the highest balance. Ordinal descriptors can move only one variant per succession step in the direction of the variant with the highest balance. This combination of global succession rule and descriptor types allows for finer-grained representations of the dynamics of real systems. In a subsequent release, rates of change will also be incorporated into Prophecy through an Intervals Matrix (currently under development), replacing the need for an adiabatic succession rule.

Auxiliary descriptors are used to represent impacts that depend on the variants of two or more other descriptors (Weimer-Jehle 2009). For example, in the Easterlin paradox, happiness increases with GDP growth when GDP levels are low, but this effect weakens or disappears at higher GDP levels (Richard A. Easterlin 1974). Neither GDP nor GDP growth alone can adequately represent this relationship. Happiness increases when GDP is both low and growing, meaning that the effect depends on the combination of the two descriptors. To represent such combined effects, an auxiliary descriptor is introduced whose variants correspond to the Cartesian product of the variants of the associated descriptors. Each variant of the auxiliary descriptor therefore represents a specific combination of the variants of the associated descriptors. Traditionally, the variant of an auxiliary descriptor is controlled by specifying cross-impacts on the auxiliary descriptor so that it is corrected at the next succession step (Weimer-Jehle 2009). However, this approach introduces delays and can distort successions, because the influence of the auxiliary descriptor should occur in the same step as the changes in its associated descriptors. Prophecy avoids this problem by updating the variant of the auxiliary descriptor programmatically within the same step, rather than relying on cross-impacts and succession analysis. A simple check is performed: in the current step, is the variant of the auxiliary descriptor consistent with the variants of its associated descriptors? If not, the auxiliary descriptor is updated accordingly. Proper handling of auxiliary variables extends CIB beyond pairwise causal relationships (e.g., $A \rightarrow B$) to more complex causal forms involving Boolean operators (e.g., $A \text{ and } B \rightarrow C$), which is necessary for contextualizing causal relationships (e.g., $A \rightarrow B$ holds if and only if C is present).

Through improved handling of ties, succession rules, and auxiliary descriptors, Prophecy makes CIB successions more realistic representations of the dynamics of real systems. Prophecy includes a variety of other features:

- **Perturbation analysis:** For a degree of perturbation (number of variants replaced simultaneously), Prophecy can generate all neighbours of a set of initial scenarios and subject them to succession analysis. The results are then presented in a Transformation Matrix with the initial scenarios in the first column and the attractors

in the first row. The remaining cells of the matrix contain combinations of variant replacements that cause the initial scenario to evolve into the attractor.

Perturbation analysis was developed by Kearney (2021) and is similar to the simulated annealing approach developed by Kemp-Benedict et al. (2019).

- **Pathway analysis:** For a given pair of initial scenario and attractor from the results of a perturbation analysis, Prophecy can report all pathways, which are subsequences of succession trials from the initial scenario to the attractor. Pathways are reported with summary statistics and can also be scored (e.g., desirability).
- **Sensitivity analysis:** Prophecy can generate multiple versions of a base model. Users can specify their confidence in the cross-impacts by judgment section, affecting the proportion of model versions in which those cross-impacts are included. Users can also specify the proportion of model versions in which the cross-impacts of judgment sections are deleted, have their signs flipped, or are multiplied by a specified multiplier.
- **Discrete Maximin designs:** Depending on the settings used, sensitivity analysis can involve sampling from a large space of possible model versions. To ensure diverse coverage of this space, Prophecy uses a discrete maximin design. This approach is also used for sampling from the scenario space of very large models.
- **Descriptive statistics and diagnostics:** Prophecy provides several new descriptive statistics and diagnostics for scenario analysis, perturbation analysis, and pathway analysis, such as minimum and average Hamming distance, trial convergence, model convergence, trial coverage, and pathway stability.

Prophecy is under active development (see Part 4 of this manual). Please report any bugs or feature requests to norman.kearney@unibe.ch.

Part 1: CIB Fundamentals

1.1 Descriptors, variants, and cross-impacts

CIB models consist of descriptors, variants, and cross-impacts. **Descriptors** are categorical variables representing system elements and their possible, mutually exclusive states, called **variants**. Variants interact with one another via numerical **cross-impacts**, where the impact of variant x_i of descriptor X on variant y_j of descriptor Y expresses how much the *plausibility* of y_j changes given x_i relative to the other variants of Y .

CIB models typically consist of multiple descriptors, with each descriptor affecting and being affected by one or more other descriptors. A descriptor is a **target descriptor** when it receives an impact, and a **source descriptor** when it gives an impact. Likewise, a variant is a **target variant** when it receives an impact, and a **source variant** when it gives an impact.

1.2 Matrices, judgment sections, judgment groups, and impact columns

A CIB model is represented as a symmetrical hypermatrix, with the descriptors and variants running down the lefthand columns and the top rows. CIB matrices consist of judgment sections, judgment groups, and impact columns. A **judgment section** is a sub-matrix consisting of all cells at the intersection of a source descriptor and a target descriptor. A **judgment group** is a row within a judgment section. An **impact column** is the set of all cells in the column of a target variant.

		X ₁			X ₂			Y		
		X ₁₁	X ₁₂	X ₁₃	X ₂₁	X ₂₂	X ₂₃	y ₁	y ₂	y ₃
X ₁	X ₁₁	0	0	0	0	0	0	2	1	-3
	X ₁₂	0	0	0	0	0	0	0	0	0
	X ₁₃	0	0	0	0	0	0	0	0	0
X ₂	X ₂₁	0	0	0	0	0	0	-2	0	2
	X ₂₂	0	0	0	0	0	0	0	0	0
	X ₂₃	0	0	0	0	0	0	0	0	0
Y	y ₁	1	2	-3	0	0	0	0	0	0
	y ₂	0	0	0	1	0	-1	0	0	0
	y ₃	0	0	0	0	0	0	0	0	0

Impact column (cross-impacts = relative strength of source variants)

Judgment section

Judgment group (cross-impacts = relative plausibility of target variants)

1.3 Relative plausibility of cross-impacts

Cross-impacts are zero-centred: positive impacts represent increases in plausibility and negative impacts represent decreases. Because plausibility is purely relative within a set of mutually exclusive variants, increases in plausibility must be offset by decreases elsewhere. A cross-impact of zero means “neither more nor less plausible”, so positive impacts are meaningless unless compensated by negative ones and vice versa. To ensure internal coherence and comparability, cross-impacts within a judgment group (i.e., the impacts of one source variant on all variants of a target descriptor) must sum to zero. Thus, the cross-impacts within a judgement group give the **relative plausibility** of the variants and comprise a **plausibility distribution** for the judgment group.

To see why, suppose we have two descriptors, X and Y, each with three mutually exclusive variants $[x_1, x_2, x_3]$ and $[y_1, y_2, y_3]$. Suppose that given x_1 , variants y_1 and y_2 are plausible, with y_1 more plausible than y_2 .

In Example 1, the cross-impact of x_1 on y_1 is 2, on y_2 is 1, and on y_3 is 0. This configuration is incoherent, as the increase in plausibility of y_1 and y_2 is not offset by a decrease in plausibility of y_3 .

In Example 2, this coherence is restored: if x_1 increases the plausibility of y_1 by 2 and y_2 by 1, it must decrease the plausibility of y_3 by -3.

In Example 3, the cross-impact of x_1 on y_3 is -2. This leaves a residual +1 increase in plausibility that is not accounted for. The increase in plausibility of y_1 and the decrease in y_3 cancel out, but the increased plausibility of y_2 remains undefined – increased relative to which alternative?

Only Example 2 is coherent: given x_1 , y_1 is most plausible, y_2 less so, and y_3 implausible.

		Y		
		y_1	y_2	y_3
X	x_1	2	1	0
	x_2	0	0	0
	x_3	0	0	0

Example 1: Incoherent cross-impacts

		Y		
		y_1	y_2	y_3
X	x_1	2	1	-3
	x_2	0	0	0
	x_3	0	0	0

Example 2: Coherent cross-impacts

		Y		
		y_1	y_2	y_3
X	x_1	2	1	-2
	x_2	0	0	0
	x_3	0	0	0

Example 3: Incoherent cross-impacts

1.4 Scenarios, balances, and consistency

The overall state of a CIB model is represented by a **scenario**, which consists of exactly one variant per descriptor. Depending on the variants that they contain, scenarios may be more or less internally consistent.

Internal consistency is a measure of the degree to which the variants of a scenario are mutually reinforcing and serves as an indicator of the plausibility of the scenario. Internal consistency is determined by calculating the balances of the variants and the consistencies of the descriptors.

The **balance** of a variant is calculated by summing the cross-impacts applied to the variant by the other variants in the scenario. In Example 4, there are two descriptors affecting Y: X₁ and X₂. Consider the scenario {x₁₁, x₂₁, y₁}. The cross-impact of x₁₁ on y₁ is 2, while the cross-impact of x₂₁ on y₁ is -2. Thus, the balance for y₁ is 2 + (-2) = 0.

The **consistency** of a descriptor is calculated by subtracting the maximum balance among its variants from the balance of the variant present in the scenario. This measures how far the current variant deviates from the most supported alternative. In Example 4, the variant of Y with the highest balance is y₂ with a balance of 1, while the balance of the current variant y₁ is 0. The consistency of Y in the scenario {x₁₁, x₂₁, y₁} is 0 - 1 = -1.

		X ₁			X ₂			Y		
		X ₁₁	X ₁₂	X ₁₃	X ₂₁	X ₂₂	X ₂₃	y ₁	y ₂	y ₃
X ₁	X ₁₁	0	0	0	0	0	0	2	1	-3
	X ₁₂	0	0	0	0	0	0	0	0	0
	X ₁₃	0	0	0	0	0	0	0	0	0
X ₂	X ₂₁	0	0	0	0	0	0	-2	0	2
	X ₂₂	0	0	0	0	0	0	0	0	0
	X ₂₃	0	0	0	0	0	0	0	0	0
Y	y ₁	1	2	-3	0	0	0	0	0	0
	y ₂	0	0	0	1	0	-1	0	0	0
	y ₃	0	0	0	0	0	0	0	0	0
Balances		1	2	-3	0	0	0	0	1	-1

Example 4: Balances and consistencies

By calculating the balance of each variant and the consistency of each descriptor, we can obtain a measure of the **internal consistency** of the scenario $\{x_{11}, x_{21}, y_1\}$. The consistency of X_1 is $1 - 2 = -1$, the consistency of X_2 is $0 - 0 = 0$, and the consistency of Y is $0 - (-1) = -1$.

The standard way to measure the internal consistency of a scenario is to use the consistency of the least consistent descriptor (Weimer-Jehle 2026). In this case, that would be -1 . An alternative way is to sum the consistencies of the descriptors, which gives $-1 + -1 = -2$. This more inclusive way of measuring scenario consistency is more informative and helps to compare scenarios.

Consider scenarios $\{x_{11}, x_{21}, y_1\}$ and $\{x_{11}, x_{22}, y_1\}$. In the standard way of measuring scenario consistency, both scenarios have the same consistency score of -1 . In the more inclusive way, $\{x_{11}, x_{22}, y_1\}$ has a consistency score of -1 and is more consistent than $\{x_{11}, x_{21}, y_1\}$, which has a consistency score of -2 . Prophecy implements the more inclusive measure of scenario consistency.

Scenario	Consistencies	Consistency (standard)	Consistency (inclusive)
$\{x_{11}, x_{21}, y_1\}$	-1,0,-1	$\min(-1,0,-1) = -1$	$-1 + 0 + (-1) = -2$
$\{x_{11}, x_{22}, y_1\}$	-1,0,0	$\min(-1,0,0) = -1$	$-1 + 0 + 0 = -1$

1.5 Successors and succession analysis

Inconsistencies are used to determine the **successor** (or successors) of a scenario, that is, the scenario(s) obtained by correcting inconsistencies in the current scenario. A descriptor is inconsistent when the balance of its current variant is lower than the maximum balance among its variants. Inconsistencies are corrected by replacing the variant of each inconsistent descriptor with a variant that has the highest balance.

Example 5 shows the successor of the scenario in Example 4: X_1 shifts from x_{11} to x_{12} and Y shifts from y_1 to y_2 . The resulting scenario, $\{x_{12}, x_{21}, y_2\}$, contains new inconsistencies. Example 6 shows the result of resolving these inconsistencies: a stable and internally consistent scenario in which no descriptor is inconsistent. The process of stepping through and correcting inconsistent scenarios is called **succession analysis**.

Succession analysis often leads to stable, internally consistent scenarios but may also lead to cycles. Example 7 adds cross-impacts for y_3 on X_1 and y_3 on Y , which generates a cycle starting from the scenario $\{x_{12}, x_{21}, y_3\}$ and proceeding through $\{x_{11}, x_{21}, y_3\}$, $\{x_{11}, x_{21}, y_1\}$, $\{x_{12}, x_{21}, y_2\}$, and back to $\{x_{12}, x_{21}, y_3\}$.

		X ₁			X ₂			Y		
		X ₁₁	X ₁₂	X ₁₃	X ₂₁	X ₂₂	X ₂₃	y ₁	y ₂	y ₃
X ₁	X ₁₁	0	0	0	0	0	0	2	1	-3
	X ₁₂	0	0	0	0	0	0	0	0	0
	X ₁₃	0	0	0	0	0	0	0	0	0
X ₂	X ₂₁	0	0	0	0	0	0	-2	0	2
	X ₂₂	0	0	0	0	0	0	0	0	0
	X ₂₃	0	0	0	0	0	0	0	0	0
Y	y ₁	1	2	-3	0	0	0	0	0	0
	y ₂	0	0	0	1	0	-1	0	0	0
	y ₃	0	0	0	0	0	0	0	0	0
Balances		0	0	0	1	0	-1	-2	0	2

Example 5: Succession step 1

		X ₁			X ₂			Y		
		X ₁₁	X ₁₂	X ₁₃	X ₂₁	X ₂₂	X ₂₃	y ₁	y ₂	y ₃
X ₁	X ₁₁	0	0	0	0	0	0	2	1	-3
	X ₁₂	0	0	0	0	0	0	0	0	0
	X ₁₃	0	0	0	0	0	0	0	0	0
X ₂	X ₂₁	0	0	0	0	0	0	-2	0	2
	X ₂₂	0	0	0	0	0	0	0	0	0
	X ₂₃	0	0	0	0	0	0	0	0	0
Y	y ₁	1	2	-3	0	0	0	0	0	0
	y ₂	0	0	0	1	0	-1	0	0	0
	y ₃	0	0	0	0	0	0	0	0	0
Balances		0	0	0	0	0	0	-2	0	2

Example 6: Succession step 2

		X ₁			X ₂			Y		
		X ₁₁	X ₁₂	X ₁₃	X ₂₁	X ₂₂	X ₂₃	y ₁	y ₂	y ₃
X ₁	X ₁₁	0	0	0	0	0	0	2	1	-3
	X ₁₂	0	0	0	0	0	0	0	0	0
	X ₁₃	0	0	0	0	0	0	0	0	0
X ₂	X ₂₁	0	0	0	0	0	0	-2	0	2
	X ₂₂	0	0	0	0	0	0	0	0	0
	X ₂₃	0	0	0	0	0	0	0	0	0
Y	y ₁	1	2	-3	0	0	0	0	0	0
	y ₂	0	0	0	1	0	-1	0	0	0
	y ₃	1	0	-1	0	0	0	1	-1	0
Balances		1	0	-1	0	0	0	-1	-1	2

Example 7: Start of cycle

Descriptors often have tied variants. How ties are broken affects how a succession unfolds. Example 8 adds a cross-impact for y_3 on x_{12} , which creates a tie between x_{11} and x_{12} in the scenario $\{x_{12}, x_{21}, y_3\}$. How this tie is broken determines whether the succession ends at $\{x_{12}, x_{21}, y_3\}$ or whether it enters the cycle from Example 7. If the tie is broken in favour of x_{11} , the succession enters the cycle. When it returns to $\{x_{12}, x_{21}, y_3\}$, the tie must be broken again.

		X ₁			X ₂			Y		
		X ₁₁	X ₁₂	X ₁₃	X ₂₁	X ₂₂	X ₂₃	y ₁	y ₂	y ₃
X ₁	X ₁₁	0	0	0	0	0	0	2	1	-3
	X ₁₂	0	0	0	0	0	0	0	0	0
	X ₁₃	0	0	0	0	0	0	0	0	0
X ₂	X ₂₁	0	0	0	0	0	0	-2	0	2
	X ₂₂	0	0	0	0	0	0	0	0	0
	X ₂₃	0	0	0	0	0	0	0	0	0
Y	y ₁	1	2	-3	0	0	0	0	0	0
	y ₂	0	0	0	1	0	-1	0	0	0
	y ₃	1	1	-1	0	0	0	1	-1	0
Balances		1	1	-1	0	0	0	-1	-1	2

Example 8: Tied balances

1.6 Relative strength of cross-impacts

Within judgment groups, cross-impacts indicate the relative plausibility of the variants of a target descriptor. Within impact columns, cross-impacts have a different meaning, indicating the **relative strength** of the impacts of source variants on a target variant. In example 8, the judgment group defined by x_{11} and Y contains three cross-impacts: 2, 1, and -3. Within the judgment group, these cross-impacts mean that y_1 is most plausible, y_2 less so, and y_3 implausible. The impact column of y_1 contains three cross-impacts: 2, -2, and 1. Within the impact column, these cross-impacts mean that x_{11} has the strongest positive impact on y_1 , x_{21} has an equally strong but negative impact on y_1 , and y_3 has a positive but half as strong impact on y_1 .

In summary, the cross-impacts within an impact column give the **relative strength** of the source variants and comprise a **strength distribution** for the impact column.

1.7 Rules of thumb for assigning cross-impacts

First, assign cross-impacts within judgment groups, focusing on the relative plausibility of the target variants given the source variant. Next, scan up and down the judgment groups within a target descriptor, iteratively adjusting the cross-impacts to achieve the desired relative strengths. Be careful to preserve the plausibility distributions within the judgment groups. An easy way to do this is to add, subtract, multiply, or divide all cross-impacts within a judgment group by the same value. For example, to weaken the relative strength of x_{21} on Y , the cross-impacts within the judgment group could be multiplied by 0.5. This transformation preserves the relative plausibility within the judgment group while adjusting the strength of x_{21} on Y relative to the other source variants.

1.9 Perturbation analysis

The set of all scenarios in a CIB model comprises a state space. Internally consistent scenarios and cycles are the attractors of the state space. Through succession analysis, basins of attraction can be identified, transforming the state space into a stability landscape. Perturbation analysis extends succession analysis, tracing how attractors evolve when a number their variants is replaced. The replaced variants are the **perturbations**, and the scenarios with the replacements are the **neighbours** of the attractor. The degree of perturbation to an attractor is measured by Hamming distance, which is the number of variants by which the attractor and the neighbour differ. For example, in Example 6, the scenario $\{x_{12}, x_{21}, y_3\}$ is internally consistent and therefore a point attractor. This attractor has six neighbours of degree 1:

- $\{x_{11}, x_{21}, y_3\}$, x_{12} is replaced by x_{11}
- $\{x_{13}, x_{21}, y_3\}$, x_{12} is replaced by x_{13}
- $\{x_{12}, x_{22}, y_3\}$, x_{21} is replaced by x_{22}
- $\{x_{12}, x_{23}, y_3\}$, x_{21} is replaced by x_{23}
- $\{x_{12}, x_{21}, y_1\}$, y_3 is replaced by y_1
- $\{x_{12}, x_{21}, y_2\}$, y_3 is replaced by y_2

Perturbations that result in a new attractor are **leverage points**, and the subsequence of scenarios from the neighbour to the attractor is a **pathway**. As mentioned above, the outcome of succession analysis on a neighbour depends on how ties are broken. Thus, multiple pathways and outcomes can be possible for a particular set of perturbations.

Part 2: Prophecy Quick Start Guide

This guide walks through a complete analysis sequence using the `Project` class as the primary interface. It will be replaced with a detailed user guide in a subsequent release.

Each section maps to one or more notebook cells.

2.1 Install and import

```
```python
Install (once, in a terminal or notebook cell)
pip install prophecy-cib

import dataclasses
import numpy as np

from prophecy_cib import (
 Project,
 build_matrix_from_list, # alternative model ingestion from a list CSV
 DescriptorType,
 ScenarioConfig, ScenarioSolver,
 SuccessionConfig,
 SensitivityConfig,
 PerturbationConfig,
 Tiebreaker,
```

```

 extract_attractor_scenarios, # optional; Project handles this automatically
 load_result, # reload a saved result in a new session
)
from prophesy_cib.io.input import load_scoring_profile
from prophesy_cib.analysis.perturbation import build_transformation_matrix
from prophesy_cib import write_leverage_points_summary
...

```

## 2.2 Create a project

```

```python
project = Project("MyProject", path=".")
# Creates ./MyProject/ with subdirectories for models and analyses.
# Pass fmt="named" to default all output to human-readable scenario labels.
...

```

To reload an existing project in a later session:

```

```python
project = Project.load("./MyProject")
...

```

## 2.3 Load a model from a matrix CSV

The primary input format is a matrix CSV with a two-row header. The first two columns identify source variants; the remaining columns identify target variants.

```
...

,,Economy,Economy,Environment,Environment
,,Contraction,Growth,Stressed,Stable
Economy,Contraction,0,0,2,-2
Economy,Growth,0,0,-1,1
Environment,Stressed,1,-1,0,0
Environment,Stable,-2,2,0,0
...

```python  
model = project.load_base_model("impacts.csv")  
# Saves to MyProject/models/base/ and sets project.model.  
...
```

Optional companion files can be added:

```
```python  
model = project.load_base_model(
 "impacts.csv",
 confidences="confidences.csv", # same layout; cells = confidence weights (0–1)
 intervals="intervals.csv", # rows = source variants; cols = target descriptors
```

```
)
...
```

Pass `overwrite=True` when reloading after correcting a file.

---

### 2.3.1 Alternative: build a model from a list CSV

If your impacts are easier to author as a row-per-impact list (rather than a full matrix), use `build_matrix_from_list`. It takes two files:

**`descriptors.csv`** — one row per variant, defines structure and ordering:

```
...
Descriptor,Type,Variant
Economy,nominal,Contraction
Economy,nominal,Growth
Environment,ordinal,Stressed
Environment,ordinal,Stable
...
```

Optional columns `Descriptor Index` and `Variant Index` let you fix the 1-based index of any descriptor or variant; unspecified entries are assigned sequentially (filling any gaps left by explicit values).

**\*\*` impacts\_list.csv `\*\*** — one row per non-zero cross-impact:

```
...
```

```
Source Descriptor,Source Variant,Impact,Target Descriptor,Target Variant,Confidence
```

```
Economy,Contraction,2,Environment,Stressed,1.0
```

```
Economy,Contraction,-2,Environment,Stable,1.0
```

```
Economy,Growth,-1,Environment,Stressed,
```

```
Economy,Growth,1,Environment,Stable,0.8
```

```
...
```

`Confidence` defaults to 1.0 when the column is absent or a cell is blank.

Zero-value rows are skipped.

```
```python
```

```
model = build_matrix_from_list(
```

```
    "impacts_list.csv",
```

```
    "descriptors.csv",
```

```
    normalize=True,      # distribute offset to unlisted variants so each
```

```
                        # judgment group sums to zero (default False)
```

```
    output_path="matrix.csv", # optional: write a human-readable matrix CSV
```

```
)
```

```
# Inspect matrix.csv to verify the layout, then load into the project.
```

```
project.load_base_model("matrix.csv")
```

```
...
```

****` normalize=True `**** fills in impacts for target variants not mentioned in a

judgment group, distributing the offset so the group sums to zero:

- **Nominal target**: equal share to each unlisted variant.
- **Ordinal target, one explicit impact at position k of L**:
 - k = 1 or k = L: skew by distance (farther variant receives larger share).
 - k = middle: offset split equally between variants on each side.
- Groups where every target variant is already listed are left unchanged (no error); the existing zero-sum check in `load_model_from_matrix` will warn.

If you prefer not to write an intermediate matrix file, assign the returned

`Model` directly:

```
```python
model = build_matrix_from_list("impacts_list.csv", "descriptors.csv")
project.model = model
project.save_base_model()
```

---
```

2.4 Configure descriptor metadata

Descriptors are immutable (`frozen=True` dataclass). Use `dataclasses.replace()` to produce an updated instance, then reassign the slot in `project.model.descriptors` and persist the change.

```

` `` python
# Example: set a descriptor to ordinal and add stochastic movement
policy = next(d for d in project.model.descriptors if d.name == "Policy")
policy_updated = dataclasses.replace(
    policy,
    type=DescriptorType.ORDINAL,
    chance_random=5.0, # 5 % chance of random variant assignment each step
)
project.model.descriptors = [
    policy_updated if d.index == policy.index else d
    for d in project.model.descriptors
]

project.save_base_model() # persists changes to MyProject/models/base/
` ``

```

****To change all descriptors to a particular type (e.g., ordinal)****

```

project.model.descriptors = [
    dataclasses.replace(d, type=DescriptorType.ORDINAL)
    for d in project.model.descriptors
]

project.save_base_model() # persists changes to MyProject/models/base/

```

****To convert a descriptor to auxiliary**** — use ``promote_to_auxiliary()`` when your matrix includes a Cartesian-product variable alongside its component descriptors.

The method rebuilds variants from the operand descriptors and removes any

cross-impacts that target the auxiliary (since auxiliary descriptors are updated from their operands, not via cross-impacts), warning you about each correction.

```
```python
Step 1 — update the in-memory model.
project.model.promote_to_auxiliary(3, operands=(1, 2))
Variant names are rebuilt as "&"-joined operand variant names,
e.g. "Contraction&Stressed". Cross-impacts targeting this descriptor
are removed. Cross-impacts sourced from it are kept.

Step 2 — persist to models/base/ (required; promote_to_auxiliary only updates memory).
project.save_base_model()
```
```

The impacts matrix may include the auxiliary descriptor's rows (source impacts on other variables). Its target columns will be dropped automatically if present.

Verify the result:

```
```python
for d in project.model.descriptors:
 print(d.index, d.name, d.type.value, d.chance_random)
print(project.model.calculate_scenario_count(), "possible scenarios")
```

---
```

2.5 Configure succession

``SuccessionConfig`` controls how each scenario evolves toward an attractor.

```
```python
succession_cfg = SuccessionConfig(
 tiebreaker=Tiebreaker.CONSERVATIVE, # retain current variant on tie
 steps=20, # max succession steps per trial
 trials=1, # trials per initial scenario
 cycle_min=2, # minimum cycle length (transitions)
 cycle_max=4, # maximum cycle length
 noise_tolerance=0, # exact cycle matching
)
```
```

Use ``Tiebreaker.RANDOM`` with ``trials >= 10`` whenever any descriptor has ``chance_random > 0`` or non-deterministic tiebreaking is intended.

2.6 Model versions: generate or load

``SensitivityConfig`` determines how many model versions are used.

****Base model only**** (no sensitivity analysis):

```
```python
sensitivity_cfg = SensitivityConfig(n_models=1)
```
```

****Generate versions**** using a discrete maximin design over judgment sections:

```
```python
sensitivity_cfg = SensitivityConfig(
 n_models=20,
 proportion_flip=0.2, # 20 % of judgment sections sign-flipped
 proportion_delete=0.1, # 10 % deleted
 design_iterations=500,
)
n_models is the total count including the base model.
With the default include_base=True, this yields 1 base version (v001) +
19 generated versions = 20 total.
Versions are generated on first use and cached in MyProject/models/variations/.
Re-running with the same config reuses the cached versions automatically.
```
```

To exclude the base model from the sensitivity ensemble, set `include_base=False`:

```
```python
sensitivity_cfg = SensitivityConfig(
 n_models=20,
```

```
 proportion_flip=0.2,
 proportion_delete=0.1,
 include_base=False, # 20 generated versions, base not included
)
` `` `
```

**\*\*Load custom versions\*\*** from a directory of matrix CSV files — one file per model version, loaded in alphabetical filename order:

```
` `` `

my_versions/
 version_01.csv ← loaded first
 version_02.csv ← loaded second
 version_03.csv ← loaded third
` `` `
```

Each file uses the same format as the base model's impacts matrix (see Section 3).

Key points:

- Descriptor and variant names must match the base model exactly.
- Zero cells mean the impact is absent in that version (equivalent to deletion).
- Use zero-padded filenames ( `version\_01.csv` , not `version\_1.csv` ) when you have 10 or more versions, to ensure correct alphabetical ordering.
- `n\_models` is not used for custom source; the version count is determined automatically from the number of CSV files in the directory.
- The old plain-text single-file format is no longer supported.

```

` `` `python
sensitivity_cfg = SensitivityConfig(
 model_source="custom",
 custom_path="my_versions/",
)
With the default include_base=True, the base model is prepended as v001.
If any custom version is identical to the base, a UserWarning is emitted.
Pass include_base=False to use the CSV files only, without prepending the base.
` `` `

```

When running through `Project`, custom versions are copied into `models/variations/` on first use and served from there on subsequent runs — your source directory is only read once.

> **Cache gotcha**: the cache key is the `SensitivityConfig` struct (including > the `custom\_path` string), not the contents of the directory. If you edit the > CSV files without changing `custom\_path`, the project will keep serving the > stale cached copies. To force a reload, either point `custom\_path` at a new > directory or delete the relevant entry from `models/variations/index.json`.

**Batched mode** — run multiple generation passes in one config by supplying lists instead of scalars. Every list field must have the same length; scalar fields are broadcast to all batches. The base model is prepended once to the combined list of all batches (when `include\_base=True`).

```

` `` python
sensitivity_cfg = SensitivityConfig(
 n_models=[10, 10, 10], # 10 varied versions per batch (30 total)
 proportion_flip=[0.1, 0.2, 0.3], # different flip rates per batch
 proportion_multiply=0.2, # scalar → broadcast to all batches
 multiplier=2.0, # scalar → broadcast to all batches
 design_iterations=200,
)
Result: 1 base + 10 + 10 + 10 = 31 model versions total.
` ``

```

**\*\*Per-batch RNG\*\*** — the default `batch_rng_mode="sequential"` draws from the shared RNG in order, so each batch gets a distinct random sample. Use `batch_rng_mode="reset"` to reset the RNG state to what it was before batch 1 at the start of every subsequent batch; batches with identical parameters then produce the same model sets (useful for isolating the effect of one parameter):

```

` `` python
sensitivity_cfg = SensitivityConfig(
 n_models=[10, 10],
 proportion_flip=[0.1, 0.2],
 batch_rng_mode="reset", # batch 2 reuses the same RNG state as batch 1
)
` ``

```

**\*\*Per-batch multiplier\*\*** — when `proportion_multiply > 0`, the `multiplier` field

controls the scale factor applied to judgment sections assigned the multiply treatment. In batched mode, pass a list of scalars to use a different multiplier per batch:

```
```python
sensitivity_cfg = SensitivityConfig(
    n_models=[10, 10, 10],
    proportion_multiply=0.3,
    multiplier=[2.0, 3.0, 4.0], # batch 1 ×2, batch 2 ×3, batch 3 ×4
)
```
```

**\*\*Sub-multipliers within a batch\*\*** — each element of the `multiplier` list may itself be a list. In that case the sub-multipliers are distributed as evenly as possible across the sections assigned the multiply treatment *\*within that batch\** (the same largest-remainder balance used for treatment codes), then shuffled at random. For example:

```
```python
sensitivity_cfg = SensitivityConfig(
    n_models=[10, 10, 10],
    proportion_multiply=0.4,
    multiplier=[[2, 3], [2, 4], [2, 3, 4]],
    # Batch 1: ≈50 % of multiply-treated sections scaled ×2, ≈50 % scaled ×3
    # Batch 2: ≈50 % ×2, ≈50 % ×4
    # Batch 3: ≈33 % ×2, ≈33 % ×3, ≈34 % ×4
)
```
```

```
)
...
```

You can mix scalars and inner lists freely in the outer list:

```
```python  
multiplier=[2.0, [3.0, 5.0], 4.0]  
# Batch 1: scalar ×2 | Batch 2: ×3 or ×5 | Batch 3: scalar ×4  
...  
  
---
```

2.7 Scenarios: generate or load

Pass one of these `ScenarioConfig` objects to `run_scenario_analysis`.

****Full enumeration**** — raises `ValueError` in `run_scenario_analysis()` if the count exceeds `scenario_count_threshold` (default 10,000) unless you pass `confirm_large=True`:

```
```python  
scenario_cfg = ScenarioConfig(solver=ScenarioSolver.FULL)

Suppress the threshold check entirely (no ValueError, no warning):
scenario_cfg = ScenarioConfig(solver=ScenarioSolver.FULL,
scenario_count_threshold=None)
```

```
...
```

When the count is above threshold, pass `confirm\_large=True` to proceed:

```
```python
scenario_result = project.run_scenario_analysis(
    scenario_cfg, succession_cfg, sensitivity_cfg, scoring_profile, rng,
    confirm_large=True, # required when count > scenario_count_threshold
)
```
```

**\*\*Space-filling sample\*\*** — recommended for large models:

```
```python
scenario_cfg = ScenarioConfig(
    solver=ScenarioSolver.DISCRETE_MAXIMIN,
    count=1000,
    design_iterations=100,
    seed=42,
)
```
```

**\*\*Choosing `count` and `design\_iterations`\*\***

`count` sets how many scenarios are sampled; `design\_iterations` sets how many candidate balanced designs are generated before the best one is kept.

\*`count`\*: the goal is to cover the attractor landscape, not to enumerate the scenario space. Attractors have large basins of attraction, so well-distributed samples find them reliably without exhaustive sampling. As a rule of thumb:

| Scenario space  | Suggested `count` |
|-----------------|-------------------|
| up to ~10,000   | use `FULL`        |
| ~10,000–100,000 | 200–500           |
| 100,000–10 M    | 500–1,000         |
| 10 M+           | 1,000             |

\*`design\_iterations`\*: each iteration shuffles the balanced column assignments and keeps the best result by minimum then average pairwise Hamming distance. Because the column construction is already perfectly balanced, the improvement curve is steep for the first 20–50 iterations and nearly flat after 100–200. The default of 100 (applied when `design\_iterations=0`) is well-chosen; 200 is a reasonable upper bound. Values above 200 yield negligible quality gains.

The design evaluation step is  $O(n^2)$  in memory and time (it computes all pairwise Hamming distances). This makes high `count` combined with high `design\_iterations` very expensive:

| `count` | memory per iteration | 100 iterations |
|---------|----------------------|----------------|
| 500     | ~4 MB                | ~0.2 s         |

|        |         |          |  |
|--------|---------|----------|--|
| 1,000  | ~15 MB  | ~1 s     |  |
| 2,000  | ~60 MB  | ~4 s     |  |
| 5,000  | ~375 MB | ~25 s    |  |
| 10,000 | ~1.5 GB | ~1.5 min |  |

At `count=10,000`, increasing `design\_iterations` from 100 to 1,000 adds ~13 minutes of design time with almost no quality improvement — the succession phase is almost always the dominant cost and the right place to invest run time.

**\*\*Custom CSV\*\*** — first row is descriptor names; subsequent rows are variant indices (1-based):

...

Economy,Environment,Policy

1,2,3

2,1,2

...

```
```python
```

```
scenario_cfg = ScenarioConfig(
```

```
    solver=ScenarioSolver.CUSTOM,
```

```
    custom_path="my_scenarios.csv",
```

```
)
```

```
```
```

```

```

## 2.8 Load a scoring profile

Scoring profiles assign numerical scores to (descriptor, variant) pairs. The CSV requires columns `name`, `descriptor\_index`, `variant\_index`, and `score`.

```
```python
scoring_profile = load_scoring_profile("profiles.csv", "welfare")
# Pass None to any analysis function to run without scoring.
```

```

## 2.9 Run scenario analysis

```
```python
rng = np.random.default_rng(seed=42)

scenario_result = project.run_scenario_analysis(
    scenario_cfg,
    succession_cfg,
    sensitivity_cfg,
    scoring_profile, # or None
    rng,
    name="baseline", # optional; appended to the timestamped output folder name
    fmt="named",    # optional per-run override; "index" or "named"
```

```
)
# All output tables written to MyProject/analyses/scenario/YYYY-MM-DD_HH-MM-SS_baseline/
# result.json is also written automatically — reload it in a later session with load_result().
...

```

Before the progress bars begin, a pre-flight summary is printed:

```
...
Scenario analysis: 243 scenarios × 5 models × 10 trials = 12,150 succession runs
...

```

For `FULL`` solver when the count exceeds `scenario_count_threshold`` (default 10,000), a `ValueError`` is raised unless you add `confirm_large=True`` to the call (see Section 7).

Inspect results:

```
```python
Global attractor summary
trial_convergence values sum to 1.0 across all attractors.
model_convergence is the fraction of model versions that produced this
attractor at least once (does not sum to 1).
for entry in scenario_result.attractor_summary:
 print(
 entry.label, entry.count,
 f"trial_conv={entry.trial_convergence:.2f}",
)

```

```

 f"model_conv={entry.model_convergence:.2f}",
)

Per-scenario detail
for entry in scenario_result.entries:
 print(
 entry.initial_scenario.to_index_string(),
 "→", entry.global_attractor,
 f"({entry.trial_convergence:.2f})",
)
...

```

Key columns in `raw\_trials.csv` :

- `steps` — total succession steps configured per trial
- `steps\_to\_attractor` — transitions taken to reach the attractor (used in stability formula)
- `tiebreaks` — random tiebreak count for steps 1...steps\_to\_attractor
- `tiebreaks\_initial` —  $V_0$ : tied-balance count in the initial scenario (why stability < 1.0 even with zero step tiebreaks)
- `stability` — per-trial pathway stability computed toward the trial's own attractor

---

## 2.10 Run perturbation analysis

Perturbation analysis generates all degree- $k$  neighbours of each initial scenario and identifies leverage points — neighbours whose attractor differs from the home

attractor. Pass the scenario analysis result directly; the function extracts the unique attractor scenarios automatically. For point attractors one entry is created per attractor; for cyclical attractors one entry is created per distinct step in the cycle.

```
```python
perturbation_cfg = PerturbationConfig(
    degree=1,      # replace one variant at a time
    trial_cutoff=0.0, # minimum trial convergence for an attractor to appear as a TM column
    model_cutoff=0.0, # minimum model convergence; 0.0 = include all attractors
)

perturbation_result = project.run_perturbation_analysis(
    scenario_result,    # attractor scenarios extracted automatically
    perturbation_cfg,
    succession_cfg,
    sensitivity_cfg,
    scoring_profile,   # or None
    rng,
    name="baseline",
)

# All output tables written to MyProject/analyses/perturbation/YYYY-MM-DD_HH-MM-SS_baseline/
```
```

To run perturbation on a custom set of scenarios instead, pass a `list[Scenario]`

directly. The helper `extract_attractor_scenarios(result, model)` is also available as a standalone function if you need the list for inspection before passing it.

**\*\*Starting from a saved result (new session)\*\*** — if the scenario analysis ran in a previous Python session, reload `result.json` from the output folder and pass it in exactly the same way. `model_versions` is not required; perturbation regenerates them from `sensitivity_config`.

```
```python
# New session — reload the project.
project = Project.load("./MyProject")

# Find the scenario analysis folder; list_analyses() returns entries sorted
# by folder name (timestamp order), so the last entry is the most recent.
analyses = project.list_analyses(type="scenario")
scenario_folder = analyses[-1]["path"] # or pick by name/timestamp

scenario_result = load_result(
    f"{scenario_folder}/result.json",
    project.model,
)

# Then run perturbation exactly as in the same-session workflow.
perturbation_result = project.run_perturbation_analysis(
    scenario_result,
    perturbation_cfg,
```

```

succession_cfg,
sensitivity_cfg,
scoring_profile, # or None
rng,
name="baseline",
)
...

```

Inspect leverage points and the transformation matrix:

```

```python
for entry in perturbation_result.entries:
 lps = [n for n in entry.neighbours if n.is_leverage_point]
 print(entry.initial_scenario.to_index_string(),
 f"home={entry.home_attractor} leverage_points={len(lps)}")

The attractor_summary for a perturbation result reflects neighbour succession
outcomes: count sums to N_neighbours × N_model_versions (not N_scenarios × M).
for entry in perturbation_result.attractor_summary:
 print(
 entry.label, entry.count,
 f"trial_conv={entry.trial_convergence:.3f}",
 f"model_conv={entry.model_convergence:.3f}",
)

tm = build_transformation_matrix(perturbation_result)

```

```

Resilience: fraction of (neighbour, model) pairs that stayed in the home attractor.
for sc in tm.initial_scenarios:
 key = sc.to_index_string()
 print(key, f"resilience={tm.resilience[key]:.2f}")

Accessibility: fraction of all (neighbour, model) pairs across all home scenarios
that led to this attractor (excluding self-perturbations).
for attr in tm.attractors:
 print(attr, f"accessibility={tm.accessibility[attr]:.4f}")

Transformation matrix cells use P:M format:
P = distinct perturbation combinations that reached this attractor
M = distinct model versions with at least one such leverage point
An empty cell means no neighbour succeeded to that attractor.
for sc in tm.initial_scenarios:
 key = sc.to_index_string()
 for attr in tm.attractors:
 lps = tm.cells.get((key, attr), [])
 if lps:
 n_p = len({lp.perturbation for lp in lps})
 n_m = len({lp.model_index for lp in lps})
 print(f" {key} → {attr}: {n_p}:{n_m}")
 ...

```

To see which specific descriptor changes are the levers for each transition —

without running a full pathway analysis — inspect the Leverage Points Summary.

``Project.run_perturbation_analysis()`` writes it automatically to

``leverage_points.csv`` alongside ``transformation_matrix.csv`` in the output folder.

To write it manually (e.g. when not using the ``Project`` class):

```
```python
write_leverage_points_summary(tm, "leverage_points.csv", fmt="named")
# One row per unique (source scenario, target attractor, perturbation).
# Columns: source_scenario, target_attractor, perturbation, n_models, model_ids.
# fmt="named" uses human-readable scenario labels; fmt="index" uses index strings.
```

```

## 2.11 Run pathway analysis

Pathway analysis reports every unique sequence of states from a leverage point to a chosen target attractor. The perturbation result must be in memory (not reloaded from disk) so that leverage-point trial states are available.

First choose a source scenario and a reachable target attractor:

```
```python
entry = perturbation_result.entries[0]
source = entry.initial_scenario.to_index_string()
home = entry.home_attractor
```

```
target = next(a for a in tm.attractors if a != home)
print(f"Source: {source} → Target: {target}")
...

```

Then run the analysis:

```
```python
pathway_result = project.run_pathway_analysis(
 perturbation_result,
 source_scenario=source,
 target_attractor=target,
 scoring_profile=scoring_profile, # or None
 name="baseline",
)
Output written to MyProject/analyses/pathway/YYYY-MM-DD_HH-MM-SS_baseline/
...

```

Inspect pathways:

```
```python
for pw in pathway_result.pathways:
    print(f" count={pw.count} stability={pw.avg_stability:.2f}")
    print(f" {pw.key}")
...

```

2.12 Restart an interrupted analysis

Use `Project.resume_scenario_analysis()` or `Project.resume_perturbation_analysis()` to recover from a crash without leaving the `Project` API. Pass the path to the interrupted output folder; all configs and model versions are reconstructed from the `analysis.json` written at the start of that run. Completed scenarios are skipped; a new timestamped output folder is created for the resumed run.

```
```python
interrupted_dir = "./MyProject/analyses/scenario/2026-05-14_10-30-00_baseline"

scenario_result = project.resume_scenario_analysis(
 interrupted_dir,
 rng,
 scoring_profile=scoring_profile, # optional; re-apply scores if desired
)
```
```

The same method exists for perturbation analysis:

```
```python
interrupted_dir = "./MyProject/analyses/perturbation/2026-05-14_11-00-00_baseline"

perturbation_result = project.resume_perturbation_analysis(
 interrupted_dir,
```

```
 rng,
 scoring_profile=scoring_profile,
)
````
```

Both methods require that `project.model`` is already loaded (call `Project.load()`` if starting a new session). The scoring profile is not stored in `analysis.json``; pass `None`` to run without scores or re-supply the same profile to restore them.

> **Note**: `resume_perturbation_analysis()`` requires that the interrupted > folder was produced by the current version of Prophecy, which stores the > initial scenario list in `analysis.json``. Folders produced by older versions > will raise `ValueError``; use the low-level workaround below instead.

Low-level fallback (for legacy folders or full control over configs):

```
```python  
from prophecy_cib.analysis.scenario import run_scenario_analysis

scenario_result = run_scenario_analysis(
 project.model,
 scenario_cfg,
 succession_cfg,
 sensitivity_cfg,
 scoring_profile, # or None
```

```
rng,
output_dir=interrupted_dir,
)
...
```

Use the same configs and seed as the interrupted run. Resuming with different configs against an existing checkpoint will produce inconsistent results.

### 3. Acknowledgements

Prophecy has been a labour of love since 2021. During my PhD defence, Prof. Dr. Garry Peterson, my external examiner, suggested I develop my own CIB software for perturbation and pathway analysis. I took his suggestion and began working on Prophecy during my first postdoc position at the University of Bern, where my work was funded by Prof. Dr. Thomas Breu. My aim with Prophecy has been to build on the pioneering work of Prof. Dr. Wolfgang Weimer-Jehle, the founder of this brilliant method. My PhD supervisor, Prof. Dr. Vanessa Schwiezer, introduced me to CIB and deftly guided me through the development of my ideas about perturbation analysis and pathway analysis. Prof. Dr. Homer-Dixon, who co-supervised my PhD work, introduced me to complexity science, which enriched my engagement with CIB. Over the years, I have profited from conversations and collaborations with several wonderful colleagues, including Dr. Jude Kurniawan, Dr. Natalya Siddhantakar, Dr. Ajar Sharma, Dr. Anita Luzurko, Dr. Hannah Kosow, Dr. Wolfgang Hauser, and Dr. Mert Duygan. Special thanks to the students and colleagues of my *Systems Modelling for Sustainable Development* course at the University of Bern for their thought-provoking questions and inspiring engagements with CIB.

## 4. Bibliography

- Kearney, Norman. 2024. "Human Dignity and Ecological Identity." In *Policy Sciences and the Human Dignity Gap*, edited by Susan G. Clark, Evan J. Andrews, and Ana E. Lambert, vol. 58. Natural Resource Management and Policy. Springer Nature Switzerland. [https://doi.org/10.1007/978-3-031-52501-8\\_16](https://doi.org/10.1007/978-3-031-52501-8_16).
- Kearney, Norman M. 2021. "Guided Cultural Evolution and Sustainable Development: Proof of Concept and Exploratory Results." University of Waterloo. <https://doi.org/10.13140/RG.2.2.31015.88488>.
- Kemp-Benedict, Eric, Henrik Carlsen, and Sivan Kartha. 2019. "Large-Scale Scenarios as 'Boundary Conditions': A Cross-Impact Balance Simulated Annealing (CIBSA) Approach." *Technological Forecasting and Social Change* 143 (June): 55–63. <https://doi.org/10.1016/j.techfore.2019.03.006>.
- Richard A. Easterlin. 1974. "Does Economic Growth Improve the Human Lot? Some Empirical Evidence." In *Nations and Households in Economic Growth*, edited by PAUL A. David and MELVIN W. Reder. Academic Press. <https://doi.org/10.1016/B978-0-12-205050-3.50008-7>.
- Schweizer, Vanessa Jine, Alastair David Jamieson-Lane, Hua Cai, Stephan Lehner, and Matteo Smerlak. 2023. "Pathways for Socio-Economic System Transitions Expressed as a Markov Chain." *PLOS ONE* 18 (7): e0288928. <https://doi.org/10.1371/journal.pone.0288928>.
- Weimer-Jehle, Wolfgang. 2006. "Cross-Impact Balances: A System-Theoretical Approach to Cross-Impact Analysis." *Technological Forecasting and Social Change* 73 (4): 334–61. <https://doi.org/10.1016/j.techfore.2005.06.005>.
- Weimer-Jehle, Wolfgang. 2009. *Properties of Cross-Impact Balance Analysis*. Version 1. <https://doi.org/10.48550/ARXIV.0912.5352>.
- Weimer-Jehle, Wolfgang. 2026. "ScenarioWizard 5.3: Manual." CIB-Lab, ZIRIUS. [https://cross-impact.org/ressourcen/ScenarioWizardManual\\_en.pdf](https://cross-impact.org/ressourcen/ScenarioWizardManual_en.pdf).