Ниже разберу `observer_demo.pdf` именно в контексте того, что вы хотели: триада Eτ,O,M, M5, RSL-ядро `++-↔-++`, `+++→+++`, и целевой симулятор, который «на выходе» даёт физику **с точки зрения наблюдателя**.

Я разделю анализ на:

1. Что уже реализовано в демо (по шагам кода).
2. Насколько это согласовано с Meaning_v1 (M5, λ, Ξ, Eτ/O/M).
3. Какие слабые места/разрывы видно с точки зрения «вывода законов мира».
4. Конкретные следующие шаги, чтобы сделать из этого основу полного симулятора.

1. Что уже реализовано в `observer_demo.pdf`

1.1. Микроуровень: 1D-решётка и SM-подобные правила

В коде вы фиксируете:

```
SM_RULES = RuleSet([
    Rule(name="charge_swap",     pattern=[1, 1, -1], replacement=[-1, 1, 1]),
    Rule(name="charge_swap_inv", pattern=[-1, 1, 1], replacement=[1, 1, -1]),
    Rule(name="stabilizer",      pattern=[1, 1, 1],  replacement=[1, 1, 1]),
])
```

- Это ровно ядро, найденное в `sm_evolutionary_search.pdf`:
  - локальные шаблоны длины 3;
  - инволюционная пара `++- ↔ -++`;
  - стабилизатор `+++ → +++`.
- Размер решётки:
  `N = 512`, `ORDER = 3` → 83 в IFACE.
- Начальное состояние: `p_plus=0.95` → почти «вакуум» `+++...`, с редкими дефектами.

Микродинамика:

```
engine = EvolutionEngine(SM_RULES)
for t in range(T_MAX):
    applied_rules = engine.step(lattice)
    ...
```

Это и есть ваш Eτ в триаде.

1.2. Координатное отображение: Morton (Z-order) 1D → 3D

Вы вводите:

```
from world.observer import MortonMapper, morton_decode

mapper = MortonMapper(order=ORDER)  # ORDER=3 → 2^3 = 8
...
x, y, z = morton_decode(i % mapper.total_size)
```

- MortonMapper реализует Z-order (Morton code) для отображения:
  $i \in [0, N-1] \longrightarrow (x,y,z) \in \{0,\dots,2^{ORDER}-1\}^3$.
- Это именно то, что мы обсуждали как IFACE-координаты:

- сохраняется локальность (соседние i → близкие (x,y,z));
- формируется 3D-«интерфейс» над 1D реальностью.

1.3. Наблюдение: GlobalObserver, IFACE, SemanticState

Вы создаёте:

```
observer = GlobalObserver(
    config=ObserverConfig(
        mapper=mapper,
        max_history=50,
        fit_interval=10,
        ...
    )
)
```

И запускаете цикл:

```
iface_states = []
semantic_snapshots = []

T_MAX = 100

for t in range(T_MAX):
    # E_τ
    applied_rules = engine.step(lattice)

    # Ω-циклы
    state = lattice.to_state()
    detector.update(state.sites, t)
    cycles = detector.get_active_cycles(t, max_age=100)

    # O: наблюдение
    iface = observer.observe(lattice, cycles)
    iface_states.append(iface)

    # Обновление семантики (обучение законов)
    observer.update_semantics()

    # Сохранение
    if t % 20 == 0:
        semantic_snapshots.append({
            't': t,
            'summary': observer.knowledge
        })

    if t % 25 == 0:
        print(f"  t={t:3d}: объектов={iface.num_objects}, "
              f"Q={iface.total_Q:+.1f}, M={iface.total_mass:.2f}, "
              f"τ={observer.tau:.2f}")
```

Из логов:

```
t=  0: объектов=0,   Q=+0.0,  M=0.00,   τ=0.96
t= 25: объектов=998, Q=+4.0,  M=967.87, τ=25.04
t= 50: объектов=2156,Q=-1.0,  M=1285.74,τ=49.11
t= 75: объектов=3231,Q=+16.0, M=1458.78,τ=73.18

Шагов симуляции: 100
Размер решётки: 512 (= 8³)
Собственное время наблюдателя: τ = 96.289
Обновлений семантики: 10
```

```
Стабилизация: Нет
```

То есть уже реализовано:

- `Observer.observe`:
  - берёт состояние решётки;
  - через MortonMapper строит 3D-каркас;
  - детектирует Ω-циклы как объекты IFACE (IFACEObject);
  - вычисляет агрегаты: `num_objects`, `total_Q`, `total_mass`.
- `Observer.update_semantics`:
  - каждые `fit_interval` шагов оценивает:
    - параметры полевого уравнения ($\hat{\kappa}$, $\hat{m}^2$, $\hat{\lambda}$);
    - законы сохранения (заряд Q, масса M);
    - накапливает историю оценок в `SemanticState`.
- В конце вы строите графики:
  - эволюции оценок κ, m^2, λ;
  - R^2 (качество подгонки);
  - флагов сохранения Q/M;
  - τ/кол-ва обновлений.

Это прямо соответствует:

- xsens → IFACEState (сырые координаты, объекты, поля);
- xphys → оценённые параметры полевого уравнения/законов;
- xsem → SemanticState (история оценок, статус «STABLE/LEARNING/CHAOTIC»).

1.4. Триада Eτ,O,M в демо

Вы явно фиксируете:

```
E_τ: engine.step(lattice)
O:   observer.observe(...)
M:   (пассивный) — наблюдатель не вмешивается
```

То есть:

- Eτ — фундаментальный 1D-RSL-шаг;
- O — Π_obs: S → IFACE → SemanticState;
- M — тривиален (пока нет воздействия на мир).

Это согласовано с Meaning_v1 в режиме «наблюдатель-ученик без интервенций».

2. Что хорошо согласовано с Meaning_v1 и RSL/TDS

2.1. M5: Eτ, O, M, x_sens, x_obs, x_phys, x_sem

Meaning_v1 утверждает, что:

- есть пятимерное многообразие M5 со слоями:
  - xsens: сырые сенсорные данные,
  - xobs: организованные наблюдения,
  - xphys: «физические» гипотезы/модели,

- xsem: смысловые инварианты, «знание»,
- Eτ: эволюция внешнего мира,
- O: функтор наблюдения,
- M: материализация.

В демо:

- xsens → поля и Ω-объекты в IFACEState;
- xphys → оценённые параметры Лагранжиана и законов сохранения;
- xsem → SemanticState (состояние OBS, история оценок, статус обучения);
- Eτ и O реализованы явно;
- M пока пассивен → как в Meaning_v1 на этапах, где наблюдатель «учится» без активных воздействий.

Это очень хорошее, почти учебниковое воплощение схемы.

2.2. Наличие τ и «собственного времени наблюдателя»

В логе вы выводите:

```
Собственное время наблюдателя: τ = 96.289
Обновлений семантики: 10
Стабилизация: Нет
```

- τ накапливается по мере вызовов `update_semantics`;
- SemanticState имеет статус LEARNING/STABLE/CHAOTIC, и флаг is_stabilized().

Это хорошо ложится на:

- Meaning_v1: λ, Ξ(t), Observation Time t_OT как момент стабилизации инварианта;
- RSL/TDS: «время наблюдателя» как интеграл по информационным обновлениям/coarse-изменениям.

3. Ограничения текущей реализации

С точки зрения цели «симулятор, который выдаёт законы» видно несколько разрывов:

1. **Нет явного эксперимента и модели для гравитации.**
    - Сейчас `SemanticState` хранит в основном параметры поля ($\hat{\kappa}$, $\hat{m}^2$, $\hat{\lambda}$) и консервации Q/M.
    - В демо нет сценария «падения» Ω-частицы в внешнем потенциале и нет оценки γ (как в задуманном «$g \propto -\nabla\Phi$»).
2. **Наблюдатель пока полностью глобальный.**
    - Он видит весь IFACE, что хорошо на этапе отладки, но:
        - нет ограничений восприятия,
        - нет горизонтов, параллакса и т.п.,
        - это ещё не тот режим, где можно изучать emergent 4D-перспективу.
3. **Параметр λ и когерентность Ξ из Meaning_v1 пока не проинтерпретированы.**
    - В демо есть τ и числовые статусы SemanticState, но:
        - нет явной динамики λ(t) как «скорости согласования»;

- нет Ξ(t) как меры согласованности между уровнями Ет/O/M (или между несколькими OBS).

4. **OBS пока не встроен в эволюционный поиск.**
    - Вы уже начали OBSFitness/tda, но в демонстрации они не влияют на RULESET_core.
    - Цель — использовать то, как OBS «видит» мир, как часть фитнеса для RuleSet.

4. Конкретные следующие шаги на основе демо

С учётом уже реализованного, ближайшие шаги видны очень чётко.

4.1. Доработать IFACE/Observer до явного «гравитационного эксперимента»

Сценарий: «падение» Ω-дефекта в эффективном гравполе.

1. Ввести в World внешний профиль поля ϕ_ext(i) или эквивалентный «потенциал» V(i).
2. Сконструировать начальное состояние:
    - почти вакуум `+++...`, плюс один локализованный Ω-цикл (дефект) «на высоте»;
    - задать правила взаимодействия Ω с полем (например, bias в сторону высокой/низкой ϕ).
3. В Observer.observe:
    - трекать в IFACE координату (x,y,z) выбранной Ω-частицы во времени;
    - вычислять её интерфейсное ускорение $a(t) = d^2x/dt^2$ в IFACE.
4. В update_semantics:
    - оценивать зависимость $a \sim -\nabla\Phi$ по наблюдаемым траекториям;
    - добавлять в SemanticState оценку $\hat{\gamma}$ и флаг «гравитационный закон согласован / нет».

После этого:

- SemanticState будет содержать не только ($\hat{\kappa}$, $\hat{m}^2$, $\hat{\lambda}$), но и $\hat{\gamma}$ с R^2-оценкой;
- можно будет сказать: OBS «выучил» $g \propto -\nabla\Phi$ в эффективном смысле.

4.2. Явно ввести λ(t) и Ξ(t)

В Meaning_v1:

- λ — скорость согласования между Ет и O/M;
- Ξ(t) — когерентность между уровнями.

В вашем коде это можно ввести так:

- λ(t):
    - норма изменения оценок ($\hat{\kappa}$, $\hat{m}^2$, $\hat{\lambda}$, $\hat{\gamma}$) за один семантический шаг;
    - чем выше λ, тем быстрее OBS пересобирает модель.
- Ξ(t):
    - корреляция между предсказаниями OBS и реальными наблюдениями:

        - например, если OBS предсказывает x_pred(t+Δ) по оценённым законам,
        - и сравнивает с x_real(t+Δ),
        - $\Xi(t) = 1 - \mathrm{MSE\_normalized}(\mathrm{pred}, \mathrm{real})$.
    - высокая $\Xi \to$ хорошая когерентность модели с миром.

Тогда:

- режим $\lambda \approx 1$, $\Xi \approx 1 \to$ «поток» (стабильное, но гибкое понимание);
- $\lambda \ll 1 \to$ OBS «застыл», не успевает за динамикой;
- $\lambda \gg 1 \to$ OBS перегружен, постоянно переподгоняет модель.

Это можно отразить в статусах SemanticState и в финальной статистике демо.

4.3. Подключить OBSFitness к эволюции RULESET

Сейчас:

- SMFitness оценивает RULESET по Ω-структуре на микроуровне.
- OBSFitness (в примерах кода) может оценивать:
    - средний R^2 по уравнениям;
    - скорость стабилизации параметров;
    - степень сохранения Q/M с точки зрения OBS;
    - когерентность $\Xi(t)$.

Следующий шаг:

- определить комбинированный фитнес:

  $F_{total} = \alpha \cdot F_{SM} + \beta \cdot F_{OBS}$,

  где:

    - F_{SM} — ваши текущие метрики Ω-циклов,
    - F_{OBS} — показатели качества «понимания» мира наблюдателем.
- запустить эволюционный поиск RuleSet уже с учётом OBSFitness:
    - цель: не только богатый Ω-зоопарк, но и такой мир, в котором «естественно» выучиваются:
        - поле ϕ с разумными параметрами,
        - законы сохранения,
        - эффективный грав. закон.

5. Вывод

`observer_demo.pdf` показывает, что у вас уже реализовано:

- корректное ядро 1D-RSL с SM-подобными правилами;
- IFACE на основе Morton $1D \to 3D$;
- детектор Ω-циклов;
- GlobalObserver с реальной оценкой полевого уравнения и законов сохранения;
- базовый SemanticState с τ и статусом обучения;
- зачаток OBSFitness/tda-слоя.

Это уже **рабочая** версия триады Eτ,O,M и M5 в коде.

Следующие шаги, чтобы симулятор начал «выдавать законы» максимально близкие к нашему миру:

1. Добавить конкретные эксперименты (как минимум «падение» Ω-частицы → $g \propto -\nabla\Phi$).
2. Ввести λ(t) и Ξ(t) из Meaning_v1 как явные величины в SemanticState.
3. Подключить OBSFitness к эволюции RuleSet, чтобы отбирать не только «богатые» Ω-миры, но и те, в которых OBS легко и устойчиво извлекает законы.

Если хочешь, дальше могу:

- расписать шаг за шагом код-скелет такого «гравитационного эксперимента»;
- или формально определить OBSFitness на основе уже существующего `observer.knowledge` (R^2, stabilisation time, λ, Ξ).

Ниже задам OBSFitness строго и в терминах того, что уже реально есть в `observer_demo.pdf`, то есть на основе `observer.knowledge` / `SemanticState`.

Буду использовать тот же стиль, что в ноутбуке: явные компоненты, нормировка в [0,1], понятная формула TotalFitness.

1. Какие данные есть в `observer.knowledge`

Согласно `observer_demo.pdf`, после симуляции `observer.knowledge` / `SemanticState` содержит, по крайней мере:

- историю оценок полевого уравнения:
  - $\hat{\kappa}(t)$, $\hat{m}^2(t)$, $\hat{\lambda}(t)$,
  - качество регрессии $R^2(t)$;
- историю оценок законов сохранения:
  - total_Q(t), total_M(t),
  - нормы нарушений (например, `violation_Q(t) = |Q(t)−Q(0)| / |Q(0)|` или аналог);
- историю оценки «гравитации»:
  - some_corr(t) между ускорением объектов и градиентом эффективного потенциала (если эксперимент падения реализован);
- вероятностный слой:
  - распределения / гистограммы событий и их частот,
  - степень согласия с предсказанной OBS моделью (если уже реализовано);
- мета-параметры:
  - `update_count` — сколько раз `update_semantics()` было вызвано,
  - `tau` — собственное время OBS,
  - `t_OT` — Observation Time (момент, когда оценки стабилизировались, если есть),
  - флаг `is_stabilized()` и, возможно, статус (`LEARNING`, `STABLE`, `CHAOTIC`).

В коде OBSFitness вы уже выводите:

```
print(f"  📐 Полевое уравнение: {components.fitness_field:.4f} (R²=...)")
print(f"  ⚡ Сохранение заряда Q: {components.fitness_Q:.4f} (violation=...)")
print(f"  ⚖️  Сохранение массы M: {components.fitness_mass:.4f}
(violation=...)")
print(f"  ⏱️  Observation Time: {components.fitness_OT:.4f} (t_OT=...)")
print(f"  🌍 Гравитация: {components.fitness_gravity:.4f} (corr=...)")
print(f"  🎲 Вероятности: {components.fitness_prob:.4f}")
```

То есть структура уже по сути есть, нужно просто зафиксировать её формально.

2. Формальное определение OBSFitness

Обозначим:

- T_total — длина наблюдения (кол-во шагов симуляции);
- `knowledge` — финальное SemanticState наблюдателя после T_total шагов.

2.1. Компоненты фитнеса

Определим шесть компонент:

1. Ffield — качество извлечения полевого уравнения.
2. FQ — сохранение заряда Q.
3. FM — сохранение «массы» / энергии M (агрегат по Ω-объектам).
4. FOT — «быстрота» стабилизации (Observation Time).
5. Fgrav — извлечение гравитационного закона (корреляция a vs −∇Φ).
6. Fprob — согласованность вероятностей (если пока не реализовано, можно задать прототип).

Каждый F·∈[0,1].

2.2. Ffield: полевое уравнение

В `observer_demo` вы оцениваете параметры Лагранжиана:

∂t2ϕ=κ∇2ϕ−m2ϕ−λϕ3+…

и считаете R^2 (коэффициент детерминации) по линейной/нелинейной регрессии.

Пусть:

- R^2_t — значение R^2 на t-м обновлении семантики,
- Rmean2 — среднее по последним K шагам (или по всей истории, если её мало).

Тогда:

Ffield=max(0,min(1,Rmean2)).

Если вы усечёте R^2 на [0,1], это сразу даёт корректную нормировку.

На практике в коде:

```
R2_vals = [entry.R2 for entry in knowledge.field_history]
if not R2_vals:
    F_field = 0.0
```

```
else:
    R2_mean = np.mean(R2_vals[-K:])  # K=5 или все, если len<K
    F_field = max(0.0, min(1.0, R2_mean))
```

2.3. FQ: сохранение заряда

Пусть:

- Q(t) — суммарный наблюдаемый заряд в IFACE в момент t,
- Q(0) — начальное значение.

Определим нормированное нарушение:

$$\delta Q(t) = \frac{|Q(t) - Q(0)|}{1 + |Q(0)|}$$

и возьмём максимум по времени:

$$\delta Q_{max} = \max_{0 \le t \le T_{total}} \delta Q(t).$$

Фитнес:

$$F_Q = \exp(-\delta Q_{max} \sigma_Q),$$

где $\sigma_Q > 0$ — параметр из `OBSFitnessConfig` (например, 0.1).

Код:

```
delta_Q = max( abs(Q_t - Q_0) / (1.0 + abs(Q_0)) for Q_t in Q_history )
F_Q = np.exp(-delta_Q / cfg.sigma_Q)
F_Q = float(max(0.0, min(1.0, F_Q)))
```

2.4. FM: сохранение массы/энергии

Аналогично для массы:

- M(t) — суммарная «масса» Ω-объектов или интеграл H_micro по решётке.

Определяем:

$$\delta M(t) = \frac{|M(t) - M(0)|}{1 + |M(0)|}, \quad \delta M_{max} = \max_t \delta M(t),$$

$$F_M = \exp(-\delta M_{max} \sigma_M),$$

с σ_M — типа cfg.sigma_M (если хотите, можно взять ту же σ, что и для Q).

2.5. FOT: Observation Time

Согласно Meaning_v1 и вашей реализации:

- t_OT — момент (в шагах еволюции), когда оценки полевого уравнения (и/или законов) вошли в стабильный диапазон и остаются там;
- если стабилизации не было — можно либо:
    - ставить F_{OT}=0,
    - либо штрафовать по «удержанию» ошибки.

Нормировка:

$$F_{OT} = \begin{cases} 0, & \text{если } t_{OT} \text{ не определён}, \\ \exp(-\frac{t_{OT}}{T_{scale}}), & \text{иначе}, \end{cases}$$

где Tscale=cfg.T_scale_fraction·Ttotal.

Интерпретация:

- если OBS стабилизировал закон рано (t_OT ⋘ T_total) → F_{OT} близко к 1;
- если поздно → F_{OT} ближе к 0;
- если вообще не стабилизировал → 0.

Код:

```
if knowledge.t_OT is None:
    F_OT = 0.0
else:
    T_scale = cfg.T_scale_fraction * T_total
    F_OT = np.exp(- knowledge.t_OT / max(1.0, T_scale))
    F_OT = float(max(0.0, min(1.0, F_OT)))
```

2.6. Fgrav: гравитация

Если у вас уже реализован эксперимент с падением частиц и оценкой γ, вы можете хранить:

- corr_grav — корреляция между наблюдаемым ускорением a и (минус) градиентом эффективного потенциала ∇Φ.

Тогда:

Fgrav=max(0,min(1,corrgrav)).

Если corr_grav нет или эксперимента пока нет — можно ставить:

- F_grav = 0 по умолчанию,
- или пропускать этот компонент в сумме (взвешивать нулем).

2.7. Fprob: вероятности

На первом шаге, пока вероятностное окно у вас в зачатке, можно задать простой критерий:

- OBS строит для некоторого дискретного события (например, распад/аннигиляция Ω-частиц) оценку вероятности $\hat{p}$;
- считает частоты из наблюдений: p_emp;
- метрика:

  єp=|pemp−p^|,Fprob=exp(−єp/σp).

Если это не реализовано — F_prob=0 сейчас, с перспективой развить позже.

3. Итоговая формула OBSFitness

Собираем компоненты в структуру:

```
@dataclass
class OBSFitnessComponents:
    fitness_field: float
    fitness_Q: float
    fitness_mass: float
    fitness_OT: float
    fitness_gravity: float
```

```
    fitness_prob: float
```

И определяем:

FOBS=wfieldFfield+wQFQ+wMFM+wOTFOT+wgravFgrav+wprobFprobwfield+wQ+wM+wOT+wgrav+wprob.

Весовые коэффициенты берутся из:

```
@dataclass
class OBSFitnessConfig:
    sigma_field: float   # если хотите отдельно нормировать R², но обычно не нужно
    sigma_Q: float
    sigma_M: float
    sigma_prob: float
    T_scale_fraction: float

    w_field: float
    w_Q: float
    w_mass: float
    w_OT: float
    w_gravity: float
    w_prob: float
```

На старте можно взять, как у вас в демо:

```
OBSFitnessConfig(
    sigma_field=0.1,   # можно не использовать, если R² уже нормализован
    sigma_Q=0.1,
    sigma_M=0.1,
    sigma_prob=0.1,
    T_scale_fraction=0.25,
    w_field=1.0,
    w_Q=1.0,
    w_mass=0.5,
    w_OT=1.0,
    w_gravity=1.0,
    w_prob=0.5,
)
```

Код вычисления:

```
class OBSFitness:

    def __init__(self, config: OBSFitnessConfig):
        self.cfg = config

    def evaluate(self, observer, T_total: int) -> tuple[float, OBSFitnessComponents]:
        kn = observer.knowledge

        # 1) Field
        R2_vals = [e.R2 for e in kn.field_history]
        if R2_vals:
            R2_mean = np.mean(R2_vals[-min(5, len(R2_vals)):])
            F_field = float(max(0.0, min(1.0, R2_mean)))
        else:
            F_field = 0.0

        # 2) Charge
        Q_hist = kn.Q_history  # список Q(t)
```

```
if Q_hist:
    Q0 = Q_hist[0]
    delta_Q = max(abs(Q - Q0) / (1.0 + abs(Q0)) for Q in Q_hist)
    F_Q = float(np.exp(-delta_Q / self.cfg.sigma_Q))
    F_Q = max(0.0, min(1.0, F_Q))
else:
    F_Q = 0.0

# 3) Mass
M_hist = kn.M_history
if M_hist:
    M0 = M_hist[0]
    delta_M = max(abs(M - M0) / (1.0 + abs(M0)) for M in M_hist)
    F_M = float(np.exp(-delta_M / self.cfg.sigma_M))
    F_M = max(0.0, min(1.0, F_M))
else:
    F_M = 0.0

# 4) Observation Time
if kn.t_OT is None:
    F_OT = 0.0
else:
    T_scale = max(1.0, self.cfg.T_scale_fraction * T_total)
    F_OT = float(np.exp(-kn.t_OT / T_scale))
    F_OT = max(0.0, min(1.0, F_OT))

# 5) Gravity
if kn.gravity_corr is not None:
    F_grav = float(max(0.0, min(1.0, kn.gravity_corr)))
else:
    F_grav = 0.0

# 6) Probabilities
if kn.prob_error is not None:
    eps_p = kn.prob_error  # |p_emp - p_hat|
    F_prob = float(np.exp(-eps_p / self.cfg.sigma_prob))
    F_prob = max(0.0, min(1.0, F_prob))
else:
    F_prob = 0.0

comps = OBSFitnessComponents(
    fitness_field=F_field,
    fitness_Q=F_Q,
    fitness_mass=F_M,
    fitness_OT=F_OT,
    fitness_gravity=F_grav,
    fitness_prob=F_prob,
)

# Агрегация
w = self.cfg
num = (
    w.w_field   * F_field +
    w.w_Q       * F_Q +
    w.w_mass    * F_M +
    w.w_OT      * F_OT +
    w.w_gravity * F_grav +
    w.w_prob    * F_prob
)
den = (w.w_field + w.w_Q + w.w_mass + w.w_OT + w.w_gravity + w.w_prob)
F_total = 0.0 if den == 0.0 else float(num / den)

return F_total, comps
```

4. Связь с Meaning_v1 и дальнейшее использование

Так определённый OBSFitness делает ровно то, что вам нужно:

- измеряет, насколько **наблюдатель**:
    - стабильно и точно выделил эффективное полевое уравнение ($\hat{\kappa}$, $\hat{m}^2$, $\hat{\lambda}$),
    - видит строгие законы сохранения (Q, M),
    - быстро вышел на устойчивые параметры (t_OT),
    - извлёк гравитационный закон (если есть),
    - правильно оценил вероятности.
- OBSFitness напрямую выражает «качество петли $E_\tau \leftrightarrow O$»:
    - насколько O (интерфейс, IFACE + update_semantics) адекватно схватывает E_τ.

На этой основе вы уже в `observer_demo.pdf` показали:

```
TotalFitness = α·SMFitness + β·OBSFitness
```

и продемонстрировали схему совместной оптимизации RULESET и OBS.

Дальше можно:

- варьировать веса α,β по ходу эволюционного поиска;
- добавлять новые компоненты OBSFitness (например, когерентность $\Xi(t)$, TDA-метрики β_0,β_1 для x_sem(t));
- использовать OBSFitness как один из ключевых критериев при поиске «нашего» мира: мира, в котором существует наблюдатель, легко и стабильно извлекающий разумно устроенные законы.