Кратко: из observer_demo + graph_gravity_model у тебя уже есть все куски, чтобы встроить «графовую» гравитацию в 1D-RSL-мир, сделать её детерминированной и оставить простор для дальнейшего поиска правил. Ниже — формальные выводы и минимальный план правок, без случайности.

1. Что уже показал graph_gravity_model

Сводка ключевых результатов из `graph_gravity_model.pdf`:

1. На графе с лапласианом $L\phi=-\rho$ и power-law связями $P(i \leftrightarrow j) \propto |i-j|^{-\alpha}$ виден степенной спад поля:

   $\phi(d_{graph}) \sim d_{graph}^{-\beta}, F(d_{graph}) \sim d_{graph}^{-(1+\beta)}$.

2. Для **геометрического расстояния** $d = |i-source|$ на «голой» 1D-решётке:
   - поле ≈ константа,
   - $F \approx 0$ (нет истинной дальнодействующей гравитации).
3. Для power-law графов:
   - при некоторых α можно добиться $\beta \approx 1$, $F \sim 1/d_{graph}^2$,
   - эффективная спектральная размерность $d_s \approx 3$ → классическая 3D-гравитация.
4. Однако по d_geom (линейный индекс) в последнем измерении:
   - $F \sim d_{geom}^{-0.57}$,
   - это далеко от $1/r^2$, разрыв ~1.4 в показателе.

Вывод самого ноутбука:

> Для получения $F \sim 1/r^2$ нужен граф со спектральной размерностью $d_s \approx 3$.
> Такой граф можно построить из 1D решётки, добавляя дальние связи по закону $P(d) \sim d^{(-3)}$ (или близким).
> Это эквивалентно «встраиванию» 1D-системы в эффективное 3D-пространство через топологию связей.

2. Как это увязать с 1D-симулятором и OBS

Из `observer_demo.pdf`:

- фундаментальный 1D-мир:
  - RULESET_core: `++- ↔ -++`, `+++ → +++`,
  - 1D-решётка длины N (512 или $16^3 = 4096$),
- Observer:
  - проецирует 1D в 3D IFACE через MortonMapper (Z-order),
  - детектирует Ω-циклы как частицы в IFACE (IFACEObject),
  - оценивает уравнение поля, законы сохранения, время τ.

Сейчас в мире:

- эффективная «геометрия» — линейная по индексу i (или «куб» по Morton’у без дальних рёбер),
- capacity и tension завязаны только на спин s_i и его локальную окружность,

- φ-поле реализуется через coarse-ϕ_R и лаплас по 1D/решётке.

Если мы хотим привести это в согласие с `graph_gravity_model`:

- нам нужна **графовая структура связей** над 1D-индексами;
- эта структура должна быть **детерминированной (без случайностей)** и **с правильно заданным законом количества рёбер на расстоянии d**;
- laplacian по этому графу должен задавать φ-поле;
- OBS должен измерять расстояния в пространстве IFACE так, чтобы видеть $F \sim 1/r^2$.

3. Как «вшить» power-law граф без случайности

Ты отметил:

> Никаких случайных связей в графе не должно быть — весь процесс строго детерминирован.

Это не проблема. Power-law граф можно сделать **конструктивным**, не вероятностным:

3.1. Детерминированное правило связей

Вместо «с вероятностью P(d)» задаём **число рёбер на расстоянии d** по закону:

- $n_{edges}(d) \propto d^{-k}$ ($k \approx 1$ или $k \approx$(something), в твоём анализе фигурирует $\alpha = 2{,}3$; точное значение подберём по теории d_s),
- но выбираем константами:

  Например:

  - для каждого узла i:
    - для каждого расстояния d в [1, d_max]:
      - если $\lfloor c/d^k \rfloor \geq 1$, добавляем связь с узлом (i+d) mod N.

Код-скетч:

```
def build_powerlaw_graph(N, k=2, c=1.0, d_max=None):
    if d_max is None:
        d_max = N//2
    edges = set()
    for i in range(N):
        for d in range(1, d_max+1):
            # детерминированное правило: число рёбер ~ c/d^k
            num_edges = int(c / (d**k))
            if num_edges >= 1:
                j = (i + d) % N
                edges.add(tuple(sorted((i,j))))
    return edges
```

- Можно добавить shift/phase, чтобы избежать избыточной симметрии.
- Если требуется ровно один edge на расстояние d на весь граф, задаёшь итерацию по d и равномерно распределяешь по i.

Главное: никакого randomness — всё детерминировано.

3.2. Связь с α=2/3 из тетради

Ты численно и теоретически показал, что:

- для некоторого α (2 или 3, в разных формулах) power-law связи дают d_s≈3.

Сейчас важно только:

- зафиксировать **один конкретный закон**, например:

  ```
  P_эффективных связей(d) ~ 1/d^2
  ```

- и реализовать его детерминированно, как выше.

4. Встраиваем граф в φ-слой (поле)

Сейчас φ-поле у тебя определяется через 1D лаплас и coarse-ϕ_R. Нужно:

1. Расширить World:

   ```
   class World:
       def __init__(self, N, ruleset, graph_edges):
           self.N = N
           self.s = np.ones(N, dtype=int)
           self.ruleset = ruleset
           self.graph_edges = graph_edges  # список рёбер (i,j)
           self.phi = np.zeros(N, dtype=float)  # полевой слой
           ...
   ```

2. Ввести лаплас на графе G:

   Для каждого узла i:

   ```
   neighbors = [j for (u,j) in edges if u==i] + [u for (u,j) in edges if
   j==i]
   lap_phi[i] = sum(phi[j] - phi[i] for j in neighbors)
   ```

3. Обновление φ по дискретному уравнению:

   ```
   def update_phi(world):
       # φ(t+1) = φ(t) + α * Lφ + β * source(s)
       L_phi = np.zeros_like(world.phi)
       for (i,j) in world.graph_edges:
           L_phi[i] += world.phi[j] - world.phi[i]
           L_phi[j] += world.phi[i] - world.phi[j]
       rho = compute_source_from_spins(world.s)  # например, суммарный defect
   density
       world.phi += alpha * L_phi + beta * rho
   ```

Так ты получишь **laplacian-динамику φ на power-law графе**.

5. IFACE-координаты: через embedding, а не через линейный index

Самый важный момент: чтобы OBS **видел F~1/r²**, его notion of «distance» должна соответствовать **графовой геометрии**, а не просто |i−j|.

Решение:

- вместо прямого MortonMapping(i → (x,y,z)) по индексу i, **строить embedding узлов графа в R^3**, чтобы:
    - расстояние в IFACE r_obs(i,j) ≈ d_graph(i,j), или
    - по крайней мере так, чтобы лаплас на графе выглядел как обычный лаплас в 3D.

5.1. Спектральный embedding

Стандартная техника:

1. Собираем лапласиан матрицу L графа G (N×N):
    - L_{ii} = degree(i),
    - L_{ij} = −1, если (i,j) — ребро,
    - L_{ij} = 0 иначе.
2. Находим 3 наименьших *ненулевых* собственных вектора L:

```
vals, vecs = np.linalg.eigh(L)
# пропускаем нулевой eigenvector (const), берём следующие 3
idxs = [1,2,3]
X = vecs[:, idxs]  # shape (N,3)
```

3. Для узла i:

```
x_i, y_i, z_i = X[i,:]
```

Это embedding G в R^3, сохраняющий (в среднем) структуру d_graph.

Дальше:

- IFACEObject.pos = (x_i, y_i, z_i),
- OBS считает расстояния и ускорения именно в этом пространстве.

Результат:

- если φ на графе имеет φ~1/d_graph, то в таком embedding'е OBS будет видеть φ(r) ≈ 1/r, F(r) ≈ $1/r^2$.

Это ровно то, что тебе нужно: **OBS видит 3D-гравитацию, хотя субстрат 1D**.

6. Совместимость с «поиском миров»

Ты хочешь сохранить общность по отношению к поиску правил миров:

- не зашить «один конкретный граф» навсегда,
- а использовать аналитическое условие d_s=3 как конструктор.

Решение:

1. На «слое геометрии» у тебя есть параметр α (или k в n_edges(d)~d^{-k}).
2. По теории спектральной размерности (и твоим численным данным) ты можешь **вычислить α**, при котором d_s≈3.
3. Это **аналитический конструктор**:

- для любой длины N и любого требуемого d_s ты можешь построить power-law граф G(α,N), детерминированно,
- и know, что laplacian на G даёт φ~1/r и F~1/r² в графовой метрике.

4. Тогда твой «поиск миров» в пространстве RULESET остаётся максимально общим:
    - на фундаментальной стороне ты варьируешь только спиновые правила (и возможно coupling к φ),
    - геометрию графа φ-поля задаёшь аналитически, а не через GA,
    - OBSFitness оценивает, насколько хорошо OBS извлекает F~1/r² и прочие законы.

7. Выводы и минимальные правки

На основе `observer_demo.pdf` и `graph_gravity_model.pdf` можно сделать следующие практические выводы:

1. **SM-правила ++- ↔ -++ действительно оптимальны** (через ILP) среди локальных обратимых правил длины 3 с сохранением заряда. Оставляем их как фундаментальный «материальный» слой.
2. Чтобы получить 1/r²-гравитацию:
    - нужно изменить **полевую/геометрическую структуру**, а не спиновые правила:
        - добавить power-law граф G поверх 1D (детерминированно),
        - обновлять φ по лапласу на G,
        - интерпретировать φ как гравпотенциал.
3. Чтобы OBS **видел** 1/r² в IFACE:
    - IFACE-координаты должны быть не просто Morton(i), а графовый embedding (спектральный, MDS и т.п.),
    - так, чтобы r_obs ≈ d_graph, тогда φ(r_obs)~1/r_obs, F(r_obs)~1/r_obs².
4. Это полностью детерминированный конструктор:
    - никаких случайных связей;
    - весь граф строится по фиксированной формуле (α, N, n_edges(d));
    - лаплас и embedding вычисляются из этого графа.
5. При этом:
    - пространство возможных миров остаётся широким на уровне RULESET по s (спиновые правила, Ω-зоопарк),
    - а геометрия φ-поля задаётся семейством G(α), которое ты можешь либо:
        - подбирать аналитически (через d_s),
        - либо включить в поисковое пространство (но уже сильно структурированное).

Если дальше хочешь, можем:

- расписать точный интерфейс `World`/`Observer` с графовой структурой (список рёбер, laplacian update, спектральный embedding),
- или перейти к тому, как OBSFitness должен учитывать «гравитационный блок» (корреляция a vs −∇φ) в новой, графовой модели.

Ниже разобью ответ на две части, как вы просили:

1. Точный интерфейс `World` / `Observer` с:
   - явной графовой структурой рёбер,
   - лапласианом для φ-поля,
   - спектральным embedding для IFACE-координат.
2. Как OBSFitness должен учитывать «гравитационный блок» (корреляция a vs −∇φ) в этой графовой модели.

Ориентируюсь на уже реализованные структуры в observer_demo.pdf: `World.step()`, `GlobalObserver`, `IFACEState`, `SemanticState`, `OBSFitness`.

1. Интерфейс `World` / `Observer` с графовой структурой

Идея:

- фундаментальный мир остаётся 1D-решёткой спинов `s[i] ∈ {+1,−1}` с SM-правилами `++- ↔ -++`, `+++→+++` (как уже у вас);
- поверх 1D-индексов вводим **граф G** с рёбрами `edges = {(i,j)}`, построенный детерминированно по power-law (без случайностей);
- по этому графу эволюционирует скалярное поле `φ[i]` (гравпотенциал), через лапласиан Lφ;
- OBS строит IFACE-координаты не по линейному индексу и не по чистому Morton, а по **спектральному embedding'у** графа в $\mathbb{R}^3$, чтобы r_obs≈d_graph;
- в этом embedding'е OBS измеряет ускорения Ω-частиц и градиенты φ, и использует их в OBSFitness.

1.1. World: состояние и шаг эволюции

Расширим World:

```
import numpy as np
from dataclasses import dataclass
from typing import List, Tuple

@dataclass
class WorldConfig:
    N: int                  # размер 1D решётки
    graph_edges: List[Tuple[int,int]]  # список рёбер графа G
    alpha_phi: float = 0.1 # шаг для лаплассиана
    beta_source: float = 0.01  # вес источника от спинов


class World:
    def __init__(self, config: WorldConfig, ruleset):
        self.cfg = config
        self.N = config.N
        self.ruleset = ruleset

        # Спиновый слой (материя)
        self.s = np.ones(self.N, dtype=int)  # +1 вакуум
        # Можно инициализировать случайно, как в observer_demo:
        # self.s = np.random.choice([-1,1], size=N, p=[0.05,0.95])

        # Полевой слой (гравпотенциал φ)
```

```
        self.phi = np.zeros(self.N, dtype=float)

        # Предварительно построить структуру соседей для графа G
        self.neighbors = [[] for _ in range(self.N)]
        for i,j in self.cfg.graph_edges:
            self.neighbors[i].append(j)
            self.neighbors[j].append(i)

        # Кеш для Ω-циклов и других метрик, если нужно
        self.omega_catalog = []
        self.t = 0  # шаг времени

    def step_spins(self):
        """
        Обновление спинов s[i] по RULESET_core (SM-правила) —
        ваш уже реализованный EvolutionEngine.step().
        """
        # Здесь можно использовать существующий EvolutionEngine,
        # который работает поверх self.s как state.sites.
        # Для скелета просто оставим вызов внешнего объекта:
        self.s = apply_spin_rules(self.s, self.ruleset)

    def step_phi(self):
        """
        Обновление поля φ по лапласу на графе: φ_{t+1} = φ_t + α Lφ + β ρ(s).
        """
        N = self.N
        alpha = self.cfg.alpha_phi
        beta  = self.cfg.beta_source

        # Лапассиан Lφ
        L_phi = np.zeros_like(self.phi)
        for i in range(N):
            # deg(i) = len(neighbors[i])
            for j in self.neighbors[i]:
                L_phi[i] += (self.phi[j] - self.phi[i])

        # Источник от спинов (дефекты как источники)
        # Простейший вариант: ρ_i(s) = s_i (или s_i-1 и т.п.)
        rho = self.s.astype(float)  # можно взять np.where(s==-1, 1.0, 0.0) и
т.п.

        # Обновление поля
        self.phi += alpha * L_phi + beta * rho

    def step(self):
        """
        Полный E_τ: один шаг эволюции мира.
        """
        self.step_spins()
        self.step_phi()
        self.t += 1

    def get_state(self):
        """
        Удобный сборник данных для наблюдателя.
        """
        return {
            't': self.t,
            's': self.s.copy(),
            'phi': self.phi.copy(),
            'edges': self.cfg.graph_edges,
        }
```

Вместо `apply_spin_rules` у вас уже есть `EvolutionEngine`; можно адаптировать его к массиву `self.s`.

1.2. Строим power-law граф G детерминированно

Из graph_gravity_model.pdf видно, что хорошее поведение по d_graph даёт power-law P(edge) ~ d^{−α} с α≈2.

Нужно преобразовать это в **детерминированное** правило:

```
def build_powerlaw_edges(N: int, k: float = 2.0, c: float = 1.0, d_max: int =
None):
    """
    Детерминированно строим список рёбер (i,j), таких что число рёбер
    на расстоянии d примерно пропорционально c / d^k.
    """
    if d_max is None:
        d_max = N // 2

    edges = set()

    for d in range(1, d_max+1):
        # целочисленное количество "шагов" на расстоянии d
        # например, num = floor(c / d**k), и равномерно распределяем по узлам
        num = int(c / (d**k))
        if num <= 0:
            continue
        step = max(1, N // (num * 2))  # распределим по решётке
        for i in range(0, N, step):
            j = (i + d) % N
            if i != j:
                e = (min(i,j), max(i,j))
                edges.add(e)

    # Обязательно добавляем локальные 1D-соседства (цепь):
    for i in range(N-1):
        edges.add((i, i+1))

    return sorted(edges)
```

- Это детерминированно, без случайных выборов;
- степень «размазанности» задаётся `k` и `c`;
- можно подобрать их по спектральной размерности (как в тетради).

1.3. Спектральный embedding графа в $\mathbb{R}^3$

Чтобы OBS видел φ и F в координатах, где действует $1/r^2$, нужны IFACE-координаты, построенные из лапласиана графа.

Стандартная схема:

```
import scipy.sparse as sp
import scipy.sparse.linalg as spla

def spectral_embedding_3d(N: int, edges: List[Tuple[int,int]]) -> np.ndarray:
    """
    Строим 3D-спектральный embedding на основе графового лаплассиана.
    Возвращает массив coords shape (N,3).
    """
    row = []
```

```
    col = []
    data = []

    # Лапласиан L = D - A
    deg = np.zeros(N, dtype=float)
    for i,j in edges:
        deg[i] += 1
        deg[j] += 1
        row.extend([i,j])
        col.extend([j,i])
        data.extend([-1.0, -1.0])

    # диагональ
    for i in range(N):
        row.append(i)
        col.append(i)
        data.append(deg[i])

    L = sp.coo_matrix((data, (row, col)), shape=(N,N)).tocsr()

    # Ищем несколько наименьших собственных значений/векторов
    # skip eigenvector с λ=0 (константа)
    vals, vecs = spla.eigsh(L, k=4, which='SM')  # 4, чтобы отбросить 0-й
    # сортируем по значению
    idxs = np.argsort(vals)
    vals = vals[idxs]
    vecs = vecs[:, idxs]

    # coords = eigenvectors 1,2,3 (игнорируя первый с λ≈0)
    coords = vecs[:, 1:4]
    return coords  # shape (N,3)
```

- `coords[i] = (x_i, y_i, z_i)` — IFACE-позиция узла i;
- OBS будет использовать именно эти координаты для измерения расстояний, скоростей и ускорений объектов.

1.4. Observer: теперь с графовым φ и спектральными координатами

Обновим схему `GlobalObserver`:

```
@dataclass
class ObserverConfig:
    coords_3d: np.ndarray       # embedding: shape (N,3)
    max_history: int = 100
    fit_interval: int = 10
    ...


class GlobalObserver:
    def __init__(self, cfg: ObserverConfig):
        self.cfg = cfg
        self.coords_3d = cfg.coords_3d
        self.semantic_state = SemanticState(max_history=cfg.max_history)
        self.tau = 0.0  # собственное время
        self.last_objects = {}  # id -> IFACEObject for velocity/accel

    def observe(self, world: World, omega_catalog) -> 'IFACEState':
        """
        Формирует IFACEState: список IFACEObject и поле φ в 3D.
```

```
        """
        s = world.s
        phi = world.phi
        edges = world.cfg.graph_edges

        # 1. Объекты (Ω-циклы как частицы)
        objects = []
        for omega in omega_catalog:
            i_center = int(np.mean(omega.support))
            x,y,z = self.coords_3d[i_center]
            # скорость и ускорение по разности позиций; можно хранить историю
last_objects
            vel = self._estimate_velocity(omega.id, (x,y,z))
            obj = IFACEObject(
                id=omega.id,
                type=omega.type_name,   # из кластеризации Ω
                mass=omega.mass,
                Q=omega.Q,
                pos=(x,y,z),
                vel=vel,
            )
            objects.append(obj)

        # 2. Полевой срез в 3D (для OBSFitness достаточно phi и градиента)
        field = IFACEField(
            phi=phi.copy(),
            C=None  # capacity можно строить поверх s,phi, если нужно
        )

        iface_state = IFACEState(
            t=world.t,
            objects=objects,
            field=field,
        )

        # обновляем IFACE-историю в SemanticState
        self.semantic_state.append_iface(iface_state)

        return iface_state

    def update_semantics(self):
        """
        Как и в observer_demo: обновляем оценки κ,m²,λ,
        законы сохранения и теперь еще гравитационный блок.
        """
        self.semantic_state.update_field_equation()
        self.semantic_state.update_conservation()
        self.semantic_state.update_gravity_law(coords_3d=self.coords_3d)
        self.tau += 1.0
```

`SemanticState.update_gravity_law` будет как раз тем местом, где считается корреляция `a vs −∇φ`.

2. OBSFitness и «гравитационный блок» в новой модели

У вас уже есть OBSFitness (в observer_demo.pdf):

- компоненты:
  - `fitness_field` (качество полевого уравнения),
  - `fitness_Q` (сохранение заряда),

- `fitness_mass`,
- `fitness_OT`,
- `fitness_gravity`,
- `fitness_prob`.

Нужно уточнить, как именно считать `fitness_gravity` в графовой модели.

2.1. Что такое «гравитационный блок» в терминах данных

В IFACE/Observer после N шагов у вас есть:

- история IFACEState[t]:
  - для каждого объекта k:
    - позиции pos_k(t) = (x_k(t), y_k(t), z_k(t)) в спектральном embedding’e,
- история φ(t,i) по узлам графа.

Для конкретного объекта (Ω-частицы) мы можем:

1. Оценить **ускорение a(t)** по IFACE-траектории:

   ```
   v_k(t) = (pos_k(t+1) - pos_k(t-1)) / (2*Δt)
   a_k(t) = (pos_k(t+1) - 2*pos_k(t) + pos_k(t-1)) / (Δt**2)
   ```

2. Оценить **градиент поля −∇φ** в точке, где объект находится:

   - найдём ближайший узел i к позиции pos_k(t) (по евклиду в coords_3d);
   - по графу G вычислим локальный градиент:

     ```
     grad_phi(i) = average_over_neighbors( (phi[j] - phi[i]) *
     (coords_3d[j] - coords_3d[i]) / |coords_3d[j]-coords_3d[i]|^2 )
     ```

   - это приближённый вектор ∇φ(i) в IFACE-координатах.

3. Собираем выборку пар:

   ```
   { (a_k(t), -grad_phi(i_k(t))) }
   ```

по всем доступным t и всем объектам.

2.2. Мера «гравитационного фитнеса»

В идеале, хотим:

$$\mathbf{a} \approx -\gamma \, a \nabla \varphi,$$

т.е. линейную регрессию:

```
a ≈ γ * (-∇φ)
```

(массу можно учесть отдельно, γ ~ 1/m.)

Практически:

1. Строим матрицу:

   ```
   X = []  # features: -grad_phi components
   Y = []  # targets:   a components
   ```

```
for each (a, grad_phi) in samples:
    X.append([-grad_phi_x, -grad_phi_y, -grad_phi_z])  # или компоненты по
отдельности
    Y.append([a_x, a_y, a_z])
```

2. Для каждого компонента решаем линейную регрессию (или единую матрицу):

```
# например, по каждой оси
γx, *_ = np.linalg.lstsq(X[:,0:1], Y[:,0], rcond=None)
γy, *_ = ...
γz, *_ = ...
```

3. Считаем корреляции:

```
corr_x = np.corrcoef(Y[:,0], X[:,0])[0,1]
corr_y = ...
corr_z = ...
corr_mean = np.mean([abs(corr_x), abs(corr_y), abs(corr_z)])
```

4. `SemanticState` запоминает:

```
self.gravity_corr = corr_mean
self.gamma_est = (γx, γy, γz)
```

2.3. Обновлённый `fitness_gravity`

В текущем OBSFitness (в демо) вы уже используете:

```
if kn.gravity_corr is not None:
    F_grav = max(0.0, min(1.0, kn.gravity_corr))
else:
    F_grav = 0.0
```

Теперь это приобретает чёткий смысл:

- `gravity_corr` — средняя корреляция между компонентами ускорения и компонентами $-\nabla\varphi$ в графовом embedding'е;
- при идеальном законе $a=-\gamma\nabla\varphi$ она $\to 1$;
- при отсутствии связи она близка к 0.

Можно сделать нормировку и штраф за мало данных:

```
n_samples = len(samples)
if n_samples < N_min:
    F_grav = 0.0
else:
    F_grav = max(0.0, min(1.0, corr_mean))
```

2.4. OBSFitness в новом мире

OBSFitness остаётся в той же схеме (см. observer_demo.pdf):

```
TotalFitness_OBS = (w_field * F_field + w_Q * F_Q + w_mass * F_mass +
                    w_OT * F_OT + w_gravity * F_grav + w_prob * F_prob) / Σw
```

Теперь:

- `F_field` отвечает за $\partial t2\phi=\kappa\nabla2\phi-m2\phi-\lambda\phi3$ на Ω-внешнем уровне;

- `F_gravity` — за $a \approx -\gamma \nabla \varphi$ в IFACE-координатах (embedding графа);
- и оба они завязаны на **графовом лаплассиане** и embedding, а не только на 1D линейной геометрии.

3. Как это всё совместить с поиском правил

После введения:

- графовой структуры G(α,N) детерминированным способом,
- φ-поля, обновляемого по лапласу на G,
- спектрального embedding'а coords_3d,
- GlobalObserver, который видит мир в этих координатах,
- и OBSFitness с гравитационным блоком,

вы можете:

1. Зафиксировать G(α,N) на основании анализа graph_gravity_model.pdf (выбрав α, дающий $d_s \approx 3$).
2. Оставить RULESET_core по спинам (SM-правила) как стартовую точку, или позволить эволюции варьировать расширенный RULESET (например, coupling $s \to \varphi$).
3. Запустить эволюционный поиск:

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

   с:

   - SMFitness — за Ω-структуру, сохранения и стабильность;
   - OBSFitness — за способность OBS вывести:
     - корректное полевое уравнение по φ,
     - законы сохранения,
     - гравитационный закон $a \approx -\gamma \nabla \varphi$.

Так вы сможете искать **миры**, в которых:

- на фундаментальном уровне — 1D-RSL + power-law граф;
- на уровне наблюдателя — эффективная 3D-геометрия с $1/r^2$-гравитацией.

Если нужно, дальше можно:

- пройтись по одному эксперименту «падения частицы» в этом мире — от инициализации Ω-цикла до оценки F_grav;
- или детализировать ILP-часть для φ-дискретной динамики на графе (чтобы минимизировать ручной подбор α,β).