Ниже — структурированный план построения «новой физики» поверх вашего симулятора, но уже с иными первыми принципами: допускающими **нелокальные связи**, **управляемую вероятность**, **символьную арифметику вещественных** (включая «конечные строки для π»), и **расширенную логику** (тетраллема / многозначность), при этом сохраняя **эмпирическую совместимость** с текущей RSL-физикой на всех проверяемых масштабах.

Я разбью план на фазы, и в каждой дам:

- формализм (что считается состоянием, чем является закон),
- алгоритм отбора миров (что фильтруем),
- какие «невозможные» эффекты целимся получить (FTL, антигравитация) и как обеспечить, чтобы они не противоречили экспериментам.

# Фаза 0. Зафиксировать цель и «контракт совместимости»

## 0.1. Цель новой физики (NP — New Physics)

1. На уровне экспериментальных данных (низкоэнергетическая физика, SM+GR, квант. вероятности):
    - воспроизводить то, что уже воспроизводит ваш RSL-мир (N=512, α=2, L=3) — как **предельный режим**.
2. Сверх этого:
    - допускать режимы/фазы, где возможны:
        - FTL-перемещения,
        - антигравитация,
        - управляемое перераспределение вероятностей (как в ваших экспериментах),
        - нелокальная корреляция/связность.

## 0.2. “Contract of compatibility”

Это ключевой инженерный принцип, чтобы не получить «всё возможно, значит ничего не проверяется»:

- Вводим параметр(ы) деформации $\epsilon$ (или несколько), такие что:
    - при $\epsilon \to 0$ новая физика строго редуцируется к RSL-физике (или её «наблюдательскому» классу эквивалентности),
    - при $\epsilon > 0$ появляются новые явления.

То есть NP должна содержать RSL как «низкоэнергетическую фазу».

# Фаза 1. Смена основы: состояние мира, числа, логика

Это архитектурная модернизация симулятора.

## 1.1. Состояние мира как “World = (Structure, Rules, Semantics)”

Вместо чисто дискретного состояния:

- **RSL**: $S(t) \in \{-1,+1\}^N$

вводим расширенное состояние:

$W(t)=(X(t),R(t),L(t))$,
где:

- $X(t)$ — конфигурация символических структур (строки/графы/категории),
- $R(t)$ — правила эволюции (тоже как строки/объекты),
- $L(t)$ — логика/семантика интерпретации (в т.ч. тетраллема).

Это прямо соответствует SymStructures-идее: правила — тоже данные, мир самописательный.

## 1.2. Структурные вещественные числа (SR — Structural Reals)

Нужно перейти от float/рациональных «приближений» к **символьным вещественным**.

Минимальный рабочий набор:

- представление числа как конечной строки/терма:

  $r := Expr(\Sigma R)$

где Expr — выражения над символами:

- рациональные: `p/q`,
- алгебраические: `root(n, poly(p,q))`,
- трансцендентные константы: `PI`, `E`,
- функциональные: `sin(x)`, `exp(x)`, `log(x)`, `Γ(x)`,
- пределы/ряды: `sum(k=0..∞, term(k))`, `limit(n→∞, expr(n))` — но в **конечной** форме (ленивая семантика).

Ключ: арифметика должна быть **символьной** и **нормализующей**, а не численной.

Практический слой:

- ядро: канонизация (simplify),
- сравнение: не полное (в общем неразрешимо), а:
  - строгое, если удаётся доказать,
  - иначе интервальное / вероятностное / трёх- или четырёхзначное.

## 1.3. Логика: тетраллема / многозначность

Чтобы учесть Probability_v1 и тетраллему, вводим тип значения утверждения:

$V=\{T,F,B,N\}$

где, например:

- T = истинно,
- F = ложно,
- B = и истинно и ложно (противоречиво, “both”),
- N = ни истинно ни ложно (неопределено, “neither”).

Это позволит:

- не ломаться на неразрешимых сравнениях вещественных (PI vs сумма ряда),
- удерживать логические «дыры» как часть онтологии.

На уровне симулятора:

- predicates возвращают V,
- фильтры миров работают не бинарно “pass/fail”, а по логике V + шкалам доверия.

# Фаза 2. Нелокальность как управляемый ресурс (не “всё со всем”)

Чтобы получить FTL/антигравитацию и одновременно не убить causal-структуру, нужен механизм **контролируемой нелокальности**.

## 2.1. Два канала динамики

Вводим два параллельных канала эволюции:

1. **локальный канал** (RSL-совместимый):

- сохраняет каузальность на малых масштабах,
- даёт SM+GR в пределе.

2. **нелокальный канал**:

- действует редко / в специальных условиях,
- имеет ресурсную цену,
- подчиняется ограничениям «не-сигналинга» в обычных условиях (чтобы совпадать с экспериментами),
- но может “пробиваться” в режимах, где вы хотите FTL.

Формально:

$X(t+1)=E_{loc}(X(t))\circ E_{nl}(X(t);\theta)$

## 2.2. Нелокальность как граф “wormhole edges” + правила их активации

У вас уже есть power-law граф для геометрии. Добавляем **слой динамических нелокальных рёбер**:

- базовый граф G (локальный / power-law для гравитации),
- дополнительный граф H(t) (wormhole-рёбра), который:
    - **не случайный**, а определяется детерминированно из структуры/смыслов/контекста,
    - активируется только при выполнении условий.

Важное: H(t) должен быть детерминированным функционалом:

$$H(t)=F_H(X(t),S_{sem}(t),context)$$

То есть “нелокальность” может зависеть от:

- смыслового состояния наблюдателя,
- символьной структуры мира (например, совпадения сигнатур, резонансные Ω-циклы),
- условий в стиле «управления вероятностями».

# Фаза 3. Вероятность как управляемая структура (в духе ваших экспериментов)

Согласно Probability_v1:

- можно держать позицию «всё детерминировано, случайность эпистемична»,
- но вы хотите допустить альтернативу: «мир подбрасывает монетки» или «вероятность управляется».

Новая физика должна уметь обе картины как фазы или как разные режимы.

## 3.1. Два уровня вероятности

1. **эпистемическая вероятность**:

- возникает у OBS из coarse-graining и ограничений (как сейчас в RSL).

2. **онтологическая/активная вероятность**:

- мир действительно содержит ветвление или выбор (не обязательно случайный — может быть “selective” в ответ на контекст),
- либо «распределение» можно смещать процедурой M (материализацией).

В симуляторе это можно формализовать как:

$$P_{t+1}=U(P_t,X(t),S_{sem}(t))$$

где U — оператор обновления вероятностной меры.

## 3.2. Управление вероятностями как обратная связь Ет/O/M

Вот где триада Ет/O/M становится ключевой:

- OBS наблюдает (O),
- формирует смысл/цель,
- совершает действие M,
- а мир отвечает изменением вероятностной структуры.

Это похоже на ваши файлы про “probability control”: детерминированная система, но с каналом, где вероятностная мера меняется из-за контекстной интервенции.

Задача: встроить это в симулятор так, чтобы:

- в “обычной” фазе (совместимость с экспериментами) обратная связь очень мала,
- в “аномальной” фазе (FTL, антигравитация) обратная связь становится существенной.

# Фаза 4. Как получить FTL в новой физике и не сломать всё

## 4.1. Что считать FTL в вашем симуляторе

FTL должно быть определено **на уровне IFACE**:

- скорость в IFACE-метрике (embedding) превышает максимум, который получается из локального канала.

Нужно обеспечить:

- в обычной фазе FTL невозможен (совместимость),
- в новой фазе FTL реализуется как:
    - изменение активного графа связности (включение wormhole рёбер),
    - или сокращение IFACE-расстояния при сохранении локальности в подложке.

**Самый аккуратный механизм:**

- не «частица бежит быстрее», а **метрика IFACE меняется** (или активируется иной путь).

Это аналог «варп-эффекта»/wormholes в OTO: локально с не превышается, но глобально маршрут короче.

В вашей модели:

- это естественно, потому что метрика IFACE строится из спектрального embedding’а графа. Если вы детерминированно изменяете граф (активируете H(t)), вы меняете IFACE-геометрию.

## 4.2. Детерминированный триггер включения нелокальных рёбер

Триггер должен быть:

- редким,
- ресурсно дорогим,
- связанным со смысловым/контекстным состоянием (чтобы это было «устройство»).

Пример условия:

$$\text{activate}(i,j,t)=1[\text{Resonance}(\Omega i,\Omega j)>\theta \wedge Q\text{meaning}(t)>Q* \wedge \text{Capacity}(t)>C*]$$

Это делает FTL:

- не произвольным,
- а «инженерным» — нужен резонанс, смысловая структура, ресурс.

# Фаза 5. Как получить антигравитацию (в модели, где гравитация — геометрия)

Антигравитация означает:

- изменение локального ускорения относительно $\nabla\varphi$,
- или создание области с противоположным знаком эффективного g.

В вашей текущей модели:

- $\varphi$ — решение $L\varphi=\rho$, и $\rho\geq 0$ (массы “только положительные”), поэтому «настоящего отталкивания» нет.

Чтобы сделать антигравитацию, не ломая совместимость:

## 5.1. Ввести второй гравитационный канал / поле

Минимальная правка:

- добавить второе поле $\chi$ с источником другого типа:

$$L\phi=\rho m, L\chi=\rho\chi$$

и определить эффективный потенциал:

$$\Phi\text{eff}=\phi-\eta\chi$$

где $\eta$ — параметр устройства/фазы.

- В обычной фазе $\chi\approx 0 \rightarrow \Phi\text{eff}\approx\phi$, всё как у нас.
- В “антиграв” фазе $\chi$ активируется локально, создавая область, где $\nabla\Phi\_\text{eff}$ меняет знак.

Важно:

- $\chi$ должен быть **структурным**, не произвольным, и вытекающим из тех же символьных правил/ресурсов.

## 5.2. Антигравитация как метрика + управление graph geometry

Альтернатива без второго поля:

- менять метрику, т.е. структуру графа (как в FTL):
    - если g_tt и g_xx зависят от capacity/edge structure, можно детерминированно создать область, где траектории Ω-частиц ведут себя как «отталкивание».

Это по сути «инженерная геометрия»: устройство меняет локальную связность графа, а OBS видит антиграв.

# Фаза 6. Мультиверс-фильтрация миров: новый критерий отбора

Ты уже умеешь «искать миры» (ILP/сканы) и валидировать их по гравитации и SM.

Теперь нужно построить **новую OBSFitness / WorldFitness**, которая включает две зоны:

- “Compatibility mode”: мир должен воспроизводить текущие тесты SM+GR.
- “Beyond mode”: мир должен допускать FTL/антиграв в специальных условиях, без противоречий.

## 6.1. Двухрежимная валидация (двойной протокол)

Для каждого кандидата мира (X, Rules, Logic):

1. **Режим A (обычный)**:

- всё должно совпадать с текущим rsl_physics_complete:
    - Born rule corr≈1,
    - $F \sim 1/r^2$ в нужном окне,
    - SM-частицы/заряды/вершины.

2. **Режим B (аномальный / устройство)**:

- включается специальный контекст (M-действие), активирующий нелокальный канал:
    - проверяем, что возникает FTL/антиграв,
    - и при этом не рушится глобальная непротиворечивость (не появляются «вечные двигатели», не ломаются основные консервации без объяснения).

Итоговый фитнес:

$$F = w_A F_A + w_B F_B - w_P \cdot \text{paradox\_penalty}$$

где paradox_penalty — штраф за «логическую катастрофу» (неустранимые $\beta_1$_sem, невозможность стабилизации, некорректные циклы).

## 6.2. Использование Probability_v1 и тетраллемы

Чтобы избежать «противоречий» на уровне стандартной логики:

- многие проверки должны быть четырёхзначными:
    - True / False / Both / Neither,
- и штрафовать нужно не «Both» как факт (в паранепротиворечивой логике это допустимо),
  а:
    - неконтролируемые «Both» там, где модель должна быть однозначной (например, в режимах совместимости).

# Фаза 7. План работ: конкретные шаги

Ниже — “engineering roadmap”.

### Шаг 7.1. Базовая модернизация чисел и логики

- Ввести модуль `StructuralReal`:
    - канонизация, нормализация выражений;
    - сравнение: доказуемо / интервально / 4-значно.
- Ввести логику V={T,F,B,N} для проверок фильтров.

### Шаг 7.2. Нелокальный слой графа без случайности

- Добавить H(t) (wormhole-edges) как детерминированный функционал:
    - H(t)=F_H(X(t), S_sem(t), context).
- Обязательное условие: при context=0 → H(t)=∅ и всё редуцируется к RSL-миру.

### Шаг 7.3. Вероятностный слой как объект состояния

- Добавить `P_t` (мера вероятностей) в WorldState и обновление:
    - P_{t+1} = U(P_t, X_t, S_sem_t).
- В обычном режиме U ≈ identity.
- В режиме устройства — U меняет меру.

### Шаг 7.4. Два демонстратора

1. FTL-демо:
    - локально скорость не превышается,
    - глобально путь сокращается через H(t).
2. Антиграв-демо:
    - либо второе поле χ,
    - либо локальное изменение графа/метрики.

### Шаг 7.5. Новый “мультиверс-поиск”

- Использовать уже существующий симулятор как ядро, но расширить:
    - пространство поиска включает не только RULESET, но и:
        - правила формирования H(t),
        - правила U(P),
        - параметры структурных чисел.
- Фильтры:

- совместимость с экспериментами (режим A),
- наличие «новых эффектов» в режиме B,
- топология понимания $\beta_1$_sem(t) должна быть:
  - сначала ↑ (открытие),
  - потом ↓ (стабилизация),
- иначе мир «непонимаем» или «парадоксален».

# Фаза 8. Что считать «идеальным» успехом новой физики

Мир-кандидат считается успешным, если:

1. При context=0 (обычная физика):
   - воспроизводит SM+QM+GR результаты как в `rsl_physics_complete.pdf`[/files/OHW5xueA14w94V2Qku6iR].
2. При context=1 (активация устройства):
   - демонстрирует:
     - эффективный FTL (сокращение IFACE-пути),
     - антигравитацию (локальное изменение $\nabla\Phi$_eff),
   - но не создаёт логических/термодинамических катастроф:
     - сохраняет базовые инварианты (или объяснимо нарушает),
     - не рушит способность наблюдателя стабилизировать смысл ($\lambda$_s $\to 1$, $\beta_1$_sem $\to$ низко).

## Заключение

Ты предлагаешь не «добавить пару фич к RSL», а действительно перейти к следующей метатеории, где:

- дискретное вычисление + графовая геометрия сохраняются как устойчивый слой;
- поверх них вводится:
  - структурная арифметика вещественных,
  - паранепротиворечивая логика,
  - детерминированная нелокальность,
  - управляемая вероятностная мера.

И это логично: твой симулятор уже доказал, что «физика как следствие информационной структуры» работает на уровне MVP. Следующий шаг — расширить класс информационных структур так, чтобы они включали **новые режимы**, не противоречащие «старому миру».

Если ты пришлёшь/прикрепишь `Phil_Borsch_v1.md`, `Algorithm_Final.md`, `Results_15-12-25.md`, я могу:

- привязать предложенный план к их конкретным механизмам «управления вероятностями» и «нелокальных связей»,

- и предложить точные формулы/интерфейсы в коде симулятора (какие новые классы, какие метрики, какие тесты).

Ниже — конкретный, «инженерный» дизайн механизмов **управления вероятностями** в рамках новой модели (поверх уже работающего RSL-симулятора), с точными формулами и интерфейсами. Я буду прямо опираться на:

- идею триады Eτ,O,M и «вычисления через архивы» из Meaning_v1.pdf (гл. 15: наблюдение архива, реконструкция, управляемый обратный прогноз);
- практику “Stone / Philosophical Stone” из Algorithm_Final.md и Results_15-12-25.md (у тебя там показано, что можно поднять вероятность попадания в целевую область на порядки, вплоть до 100%);
- прикладные рассуждения о “кухне” (управляемые параметры + целевой семантический класс) из Phil_Borsch_v1.md.

Ключевое требование: **всё детерминировано** (никаких случайных рёбер/рандома как онтологии), а «вероятность» — это либо:

- эпистемический объект наблюдателя (распределение по микросостояниям/историям),
- либо управляема через интервенции M (контроль начальных условий, контекста измерений, ресурсов, выбора окон архива), но не как “бог с рандомом”.

# 0) Концепция: где в симуляторе живёт вероятность

## 0.1. Два уровня вероятности (как это совместить с вашим текущим миром)

**Уровень мира (онтология):**

- Мир эволюционирует детерминированно:
  $X_{t+1} = E_\tau(X_t;\theta)$
  где $\theta$ — параметры мира (RULESET, $\alpha$ графа, k=512 и т.п.).

**Уровень наблюдателя (эпистемика):**

- Наблюдатель не знает микро-деталей (или не различает их из-за Π), поэтому держит распределение по «микро-вариантам» $\omega \in \Omega$:
  $\mu_t(\omega) = P(\omega \mid \text{архив/наблюдения до } t)$
- «Вероятность» цели — это мера множества микро-вариантов, при которых цель достигается:
  $P_t(\text{hit}) = \mu_t(\{\omega : \text{Target}(E_\tau(X_0(\omega);\theta)) = 1\})$

**Управление вероятностями** в этой картине означает:

- не “сломать детерминизм”, а
- найти такие интервенции u (в рамках M), чтобы множество {ω:hit} имело большую меру или чтобы из архива можно было вычислить/выбрать ω из нужного класса.

# 1) Формальная постановка «Stone-механизма» в новой физике

## 1.1. Сопоставление с Algorithm_Final (Stone v2)

В Algorithm_Final.md:

- есть генератор “фона” (BZ patterns),
- есть encoder (β-VAE),
- есть policy π(θ|z),
- есть генератор объектов (SBM graphs),
- есть TargetSpec,
- есть метрика успеха: hit rate.

В твоём мире это переводится так:

- **Мир-генератор**: детерминированный симулятор RSL-мира с параметрами θ:
  - θ включает:
    - RULESET (переписывания),
    - параметры графа (α, k, построение power-law),
    - параметры наблюдателя Π_obs,
    - параметры «интервенций» M (что мы разрешаем менять).
- **TargetSpec**: целевой класс «IFACE-свойств», напр.:
  - “FTL произошло”,
  - “антиграв-эффект измерен”,
  - “Born-корреляция>0.99”,
  - “$F \sim 1/r^2$ в 3D embedding”,
  - “SM-заряды восстановлены автоматически”.
- **Policy π**: детерминированная функция, выдающая интервенции/настройки:
  $u=\pi(z,\text{target})$
  где z — латентное состояние “контекста/архива”.

**Ключ**: в твоих результатах Stone v2 важный инсайт — BZ не был критичен: политика в итоге возвращает фиксированное θ* и даёт 100% hit rate. Это означает, что “управление вероятностями” может быть реализовано как:

- **поиск** существования θ*, после чего “вероятность” становится ≈1 в выбранном классе условий.

В новой физике это будет выглядеть как:

- нахождение «конструктора режимов» (FTL/антиграв), который в обычном режиме подавлен, а в специальном — почти гарантирован.

# 2) Как именно сделать управление вероятностями в RSL-симуляторе

## 2.1. Определяем “архив” (Meaning_v1, гл. 15)

Согласно Meaning_v1.pdf, архив — это «замороженный срез динамики», из которого NOBS может вычислять.

В коде:

```
@dataclass(frozen=True)
class Archive:
    # детерминированный лог наблюдений/состояний
    # можно хранить как IFACE или как сырой мир + Π_obs
    iface_trace: tuple  # последовательность IFACEState
    sem_trace: tuple    # последовательность SemanticState/векторов
    world_hash: str     # хэш параметров мира θ
    obs_hash: str       # хэш Π_obs
```

Архив может быть:

- полным (мир+IFACE),
- или частичным (только IFACE и семантика), как в ваших экспериментах.

## 2.2. Определяем «контекст» ω как детерминированный вход

Чтобы избежать “рандома”, вводим параметр “контекст” ω как детерминированный индекс:

- ω может кодировать:
    - начальное состояние мира X0(ω),
    - начальное состояние наблюдателя,
    - начальное состояние графа, если он параметризован,
    - набор внешних «условий эксперимента».

Важно: ω — не случайность, а **вход**.

Например:

```
@dataclass(frozen=True)
class Context:
    seed: int         # но используется как детерминированный генератор X0
    init_mode: str    # "vacuum+defects", "prepared_omega_cycle", ...
    prepared_payload: bytes | None  # если нужно заранее заданное состояние
```

Тогда:

$X0=Init(\omega)$
— строго детерминированно.

## 2.3. Управление вероятностью как задача “максимизировать меру удачных ω”

Если ты задаёшь пространство контекстов Ω (например, seed ∈ {0..9999}) и фиксированную меру μ (равномерную), то:

- базовая вероятность:
  $P0=\mu(\{\omega:hit(\theta,\omega)=1\})$
- управление означает найти интервенцию u (или параметр θ) так, что:
  $Pu=\mu(\{\omega:hit(\theta,u,\omega)=1\})$
  существенно больше.

В Stone-подходе это делается через поиск θ*. В нашем случае мы различаем:

- θ: параметры мира (можно оставить фиксированными в “нашем мире”),
- u: параметр интервенции (то, что умеет делать OBS/устройство).

Таким образом, “антиграв” и “FTL” нужно делать не как переписывание законов, а как **режимы**, активируемые интервенциями u.

# 3) Какие интервенции M разрешить, чтобы получить “невозможное”, не ломая совместимость

Чтобы физика совпадала с экспериментами “по умолчанию”, но допускала «режимы», M должен быть:

- ресурсно ограничен,
- детерминированен,
- и не сводиться к прямому изменению RULESET.

## 3.1. Минимальный набор M-интервенций

Я бы ввёл три класса интервенций:

### M1) Подготовка начальных условий (state preparation)

- Создать специфические конфигурации Ω-циклов/полей/графа, которые редко возникают “естественно”.
- Это эквивалент «очень точной лабораторной подготовки».

Пример:

```
@dataclass(frozen=True)
class ActionPrepare:
    patches: list[tuple[int, np.ndarray]]  # (index, local_state_patch)
```

### M2) Изменение наблюдательного окна / проекции Π_obs (measurement context)

- Вероятности на уровне OBS зависят от Π (coarse-graining). Контекст измерения — часть физики.

Пример:

```
@dataclass(frozen=True)
class ActionMeasureContext:
    projector_id: str      # какой Π_meas применяется
    region: tuple[int,int] # где измеряем
    resolution: int        # масштаб coarse-graining
```

Это прямо следует из Meaning_v1 и RSL-логики: измерение — это новый projector Π_meas, меняющий классы эквивалентности.

### M3) Управляемое изменение графовой геометрии в допустимом классе (wormhole edges)

Если ты хочешь FTL/антиграв как “геометрические устройства”, самый «чистый» путь — менять не RULESET, а **активный слой геометрии** в строго заданном семействе.

Пример:

```
@dataclass(frozen=True)
class ActionGeometry:
    mode: str  # "baseline", "wormhole"
    params: dict  # параметры добавочных рёбер H(t)
```

Важно: геометрическое изменение должно быть детерминированным и ресурсным:

- “wormhole edges” появляются, если выполнены условия резонанса/смысловой плотности/ресурса.

# 4) Конкретный алгоритм “Stone” для вашего симулятора

Пусть цель — увеличить вероятность попадания в целевую область Ω_target мира наблюдателя (например, событие FTL или антиграв).

## 4.1. Интерфейсы кода (минимально)

### 4.1.1. Генератор мира

```
class WorldSimulator:
    def run(self, theta_world, context: Context, action: Action | None) ->
Archive:
        # 1) X0 = Init(context)
        # 2) применить action (M): подготовка/контекст измерения/геометрия
        # 3) прокрутить Et+O (с OBS)
        # 4) вернуть Archive с IFACE и semantic trace
        ...
```

### 4.1.2. Целевая спецификация

```
class TargetSpec:
    def evaluate(self, archive: Archive) -> dict:
        """
        Возвращает:
          hit: bool
          score: float (0..1)
          diagnostics: ...
        """
        ...
```

Пример целей:

- антиграв: средняя корреляция a·(+∇φ) > 0.8 (т.е. отталкивание) в регионе;
- FTL: событие, где объект за Δt проходит IFACE-дистанцию > c_eff·Δt при этом Eτ локален (т.е. сокращение пути через геометрию).

### 4.1.3. Обучение политики управления

В духе Stone v2, но без BZ:

- пространство действий U = семейство Action (prepare/measure/geometry),
- пространство контекстов Ω = {seed=0..S-1}.

Определим:

- базовую вероятность:

```
def estimate_P0(theta_world, target, contexts):
    hits = 0
    for ω in contexts:
        arch = sim.run(theta_world, ω, action=None)
        hits += target.evaluate(arch)['hit']
    return hits / len(contexts)
```

- управление (поиск action* или policy π):

В простейшем виде (как в твоём Stone результат):

- найти один action* (или небольшой набор), который даёт высокий hit rate для большинства ω.

Это детерминированная версия:

```
def find_action_star(actions, contexts, theta_world, target):
    best = None
    best_rate = -1
    for a in actions:
        hits = 0
        for ω in contexts:
            arch = sim.run(theta_world, ω, action=a)
            hits += target.evaluate(arch)['hit']
        rate = hits / len(contexts)
        if rate > best_rate:
            best_rate = rate
            best = a
    return best, best_rate
```

Если нужно “π(action | context)”:

- можно сделать supervised learning:
    - feature(context) → action,
    - но контекст может быть просто seed или характеристика архива.

Но ключ: всё детерминировано, «вероятность» — это частота по множеству контекстов.

# 5) Как встроить ваши результаты про нелокальность и «управление вероятностями» в новую физику

## 5.1. Нелокальные связи без отрицания детерминизма

То, что в экспериментах выглядело как «управление вероятностью», можно интерпретировать так:

- мир детерминированен;
- но существует скрытый параметр ω (контекст/архив);
- управление выбирает action u, который:
    - меняет разбиение Π_meas (что считается событием),
    - или меняет геометрию (добавляет детерминированные wormhole-рёбра),
    - или изменяет подготовку начального состояния,
- в результате множество ω, приводящих к hit, становится большим.

Это не противоречит RSL-локальности на микро-уровне, потому что нелокальность переносится в:

- структуру графа (геометрию), а она сама — «закон мира»;
- или в выбор измерительного контекста (Π_meas), а это часть O и M по Meaning_v1.

## 5.2. Тетраллема / многозначная логика для «неклассических» миров

В Probability_v1 и тетраллеме важен момент:

- классическая логика “true/false” может быть неадекватна при описании вероятностных/контекстных явлений.

В симуляторе это можно внедрить как:

- `TargetSpec.evaluate` возвращает не hit∈{0,1}, а hit∈{T,F,B,N} + confidence;
- в фитнесе:
    - T с высокой уверенностью — лучший,
    - B/N — допустимы в фазе «исследования»,
    - F — провал.

Это позволит эволюции/поиску не «выкидывать» миры с парадоксальными зонами, а отслеживать их и классифицировать.

# 6) Конкретные механизмы FTL и антиграв, которые можно реализовать на базе текущего симулятора

## 6.1. FTL как «сокращение пути» (wormhole-геометрия)

- В обычном режиме граф G(α=2.0,k=512) фиксирован → локальная причинность.
- В режиме устройства `ActionGeometry(mode="wormhole")` добавляется слой H(t) рёбер по детерминированному правилу:

Пример детерминированной активации:

$(i,j)\in H \iff \mathrm{Res}(\Omega i,\Omega j)>\Theta \wedge Q(t)>Q* \wedge \mathrm{budget}>0$

Где:

- Res — резонанс (совпадение периодов Ω-циклов, фазовых индексов),
- Q(t) — смысловая плотность наблюдателя (у тебя уже в SemanticState),
- budget — ресурс устройства.

В IFACE embedding расстояние между объектами резко падает → OBS видит “FTL”, хотя по Ет всё идёт локально по расширенному графу.

## 6.2. Антиграв как второе поле (χ) или отрицательный вклад в Φ_eff

Минимально:

- добавляем поле χ на том же графе:

$L\chi=\rho\chi$
$\Phi eff=\phi-\eta\chi$

Устройство — это action, которое создаёт ρ_χ (источник χ) в конфигурации, которая локально переворачивает знак ∇Φ_eff.

Так антиграв становится:

- не «нарушением закона», а управлением геометрией/полем.

# 7) Что прямо сейчас можно сделать в коде (план интеграции)

## 7.1. Добавить модуль управления вероятностями

Файловая структура (минимум):

```
world/
  sim/
```

```
  simulator.py         # WorldSimulator.run()
control/
  actions.py           # ActionPrepare, ActionMeasureContext, ActionGeometry
  target.py            # TargetSpec (FTL, anti-grav, etc.)
  stone.py             # find_action_star / train_policy
```

## 7.2. Добавить в World поддержку ActionGeometry и (опционально) χ-поля

- World.step() расширить:
    - если action.geometry_active:
        - обновить graph edges (детерминированно),
        - пересчитать embedding/лапласиан (или использовать быстрый апдейт).
- χ-поле:
    - добавить `world.chi` и `world.step_chi()`.

## 7.3. В Observer/SemanticState добавить Q(s) и λ_s(t)

У тебя уже есть дизайн Q(s)=H·C и возможность логировать Q(t), λ_s(t).
Это станет ключом для «активации режимов» (резонансный триггер).

# 8) Итог: что дают приложенные файлы для новой физики

- Algorithm_Final.md и результаты показывают, что **вероятность целевых событий можно повысить на порядки** (вплоть до 100%) за счёт поиска/обучения стратегии управления параметрами θ.
- Meaning_v1.pdf, гл. 15, даёт философско-формальный язык для этого: «вычисление через архивы», «управляемый обратный прогноз», «инженерия обратных задач».
- Phil_Borsch_v1.md даёт универсальный шаблон: симулятор + управляемые параметры U + проекция σ_target в семантическое пространство.

Вместе это позволяет построить **детерминированную** “probability control” механику в вашей новой физике:

- вероятность как мера по множеству контекстов ω,
- управление как выбор интервенции M, меняющей доступную структуру (подготовка/измерение/геометрия),
- и новые эффекты (FTL/антиграв) как редкие режимы, активируемые детерминированными условиями.

Если хочешь, следующий шаг — я могу:

1. Ввести конкретные `TargetSpec` для FTL и антиграв (формулы и проверки по IFACE),
2. предложить конкретное пространство Actions (U) минимальной размерности,

3. и описать, как встроить `find_action_star` и/или policy-обучение в твой текущий код так, чтобы получить результат типа “РФ / P0 ⩾ 1” уже на RSL-мире.

Ниже — полностью конкретный дизайн для:

1. **TargetSpec** для FTL и антиграв (формулы, что считать “hit”, какие метрики);
2. минимального **пространства Actions U** (интервенций M) — без изменения RULESET, строго детерминированного;
3. как встроить `find_action_star` и/или обучение policy πΦ в текущий код так, чтобы получить эффект типа
   $РФP0 \geqslant 1$
   на RSL-мире (аналогично вашим Stone-экспериментам).

Я буду использовать терминологию из:

- Meaning_v1.pdf, особенно §16.6.3 (ката как кэширование инвариантов, $tOT \to 0$) и таблицу режимов Ет/О;
- Phil_Borsch_v1.md — policy πΦ и измерение РФ;
- Algorithm_Final.md — трифазная схема: поиск θ*, затем mapping/policy, затем РФ/P0;
- Results_15-12-25.md — эмпирический факт “РФ/P0 огромно” (вы это используете как мотивацию).

# A) TargetSpec для FTL и антиграв: определения, метрики, “hit”

## A.0. Общая конструкция TargetSpec

Для вашего симулятора TargetSpec должен работать на **Archive**, то есть на сохранённой трассе:

- IFACEState(t): список объектов (Ω-частиц), их pos/vel, поле φ, capacity и т.п.
- семантика OBS: параметры закона, Q(t), λ_s(t), $\beta_1$_sem(t), и т.д.

Интерфейс:

```
@dataclass
class TargetResult:
    hit: bool
    score: float              # 0..1
    diagnostics: dict

class TargetSpec:
    def evaluate(self, archive: Archive) -> TargetResult:
        ...
```

Архив можно получить из уже имеющегося симулятора:

```
archive = sim.run(theta_world, context=ω, action=u)
```

# A.1. TargetSpec: FTL (сверхсветовое перемещение)

## A.1.1. Что считать “FTL” в вашей архитектуре

В вашем мире есть:

- локальная детерминированная эволюция $E_\tau$ (переписывания, графовая φ-динамика);
- IFACE-координаты (3D embedding графа);
- “скорость света” как максимальная скорость распространения влияния **в baseline-режиме**.

Поскольку у вас всё детерминировано, правильная постановка FTL — не “частица превысила c”, а:

> существует интервенция u (действие M), которая, оставаясь детерминированной и локальной на уровне $E_\tau$, приводит к тому, что наблюдатель видит транспорт/сигнал быстрее baseline-границы (в IFACE-метрике) без изменения базовой физики.

То есть FTL как **shortcut-геометрия / wormhole-edges**.

## A.1.2. Ввод “baseline c_eff”

Для данного мира θ_world определяем baseline-границу скорости:

1. запускаем baseline-симуляцию без интервенции u (M=id) на M контекстах ω;
2. берём верхнюю квантиль скоростей объектов (или сигналов):

$c_{eff} = \mathrm{Quantile}_q(\{\|\Delta \vec{x}\|/\Delta t\})$,
например q=0.999 (чтобы исключить выбросы).

В коде:

```
def estimate_c_eff(archives, q=0.999):
    speeds = []
    for arch in archives:
        for obj_track in arch.iface_trace.object_tracks:
            for t in range(len(obj_track)-1):
                dx = np.linalg.norm(obj_track[t+1].pos - obj_track[t].pos)
                speeds.append(dx)  # dt=1
    return float(np.quantile(speeds, q))
```

## A.1.3. FTL-hit критерий

Устройство/интервенция u считается FTL-успешной, если существует хотя бы один объект-носитель (или “сообщение”), для которого на некотором шаге:

$v(t) = \|\vec{x}(t+1) - \vec{x}(t)\| \Delta t \geq \gamma_{FTL} \cdot c_{eff}$

с $\gamma_{FTL} > 1$, например 2.0 или 10.0.

Но чтобы не ловить “выбросы embedding”, добавляем устойчивость:

- это должно произойти **K раз** (например, K=3) или в течение окна времени;
- или должна быть устойчивая “быстрая доставка” из A в B.

**Минимальный вариант:**

- один “FTL jump” длиной > γ·c_eff.

**Более физичный вариант:**

- доставка сигнала:
    - у вас есть два маркера: source region и target region;
    - событие “signal delivered” фиксируется, когда объект/маркер достигает target;
    - FTL значит, что delivery time меньше baseline-оценки.

Тогда:

$T_{deliver}(u,\omega) \le D(A,B)c_{eff}\cdot\eta$
с $\eta<1$.

### A.1.4. Реализация TargetSpecFTL

```
@dataclass
class TargetSpecFTL(TargetSpec):
    gamma_ftl: float = 2.0
    min_events: int = 1

    def evaluate(self, archive: Archive) -> TargetResult:
        c_eff = archive.meta['c_eff']  # заранее записать в Archive
        events = 0
        max_ratio = 0.0
        for track in archive.iface_trace.object_tracks:
            for t in range(len(track)-1):
                v = np.linalg.norm(track[t+1].pos - track[t].pos)  # dt=1
                ratio = v / max(1e-9, c_eff)
                max_ratio = max(max_ratio, ratio)
                if ratio >= self.gamma_ftl:
                    events += 1
        hit = (events >= self.min_events)
        score = min(1.0, max_ratio / self.gamma_ftl)
        return TargetResult(hit=hit, score=score, diagnostics={
            'events': events,
            'max_ratio': max_ratio,
            'c_eff': c_eff,
        })
```

# A.2. TargetSpec: Антигравитация

В вашей модели гравитация проявляется на IFACE как:

$\vec{a}(t) \approx -\gamma \nabla \phi(\vec{x}(t))$

(у вас это уже в OBSFitness как gravity_corr).

Антигравитация в строгом операциональном смысле:

> локально возникает режим, где ускорение направлено **в сторону $+\nabla\varphi$**
> (отталкивание от массы), или где эффективный $g \approx 0$ (экранирование).

### A.2.1. Вариант 1: отталкивание (sign flip)

Для набора точек траектории объекта:

- оцениваем $\vec{a}(t)$ по разностям pos(t);
- оцениваем $\nabla\phi(\vec{x}(t))$ (по графовому градиенту или embedding-градиенту);
- считаем косинус угла:

$\cos\theta(t) = \langle \vec{a}(t), \nabla\phi \rangle / \|\vec{a}(t)\| \cdot \|\nabla\phi\|$

В обычной гравитации $\vec{a}$ коллинеарно $-\nabla\phi$, поэтому $\cos\theta \approx -1$.

Антиграв означает $\cos\theta \approx +1$ на значимом участке.

Критерий hit:

- существует окно из K последовательных шагов, где $\cos\theta(t) \geq \theta*$ (например 0.8).

## A.2.2. Вариант 2: экранирование (g≈0)

Вместо переворота знака можно принять:

- ускорение по модулю сильно меньше нормы градиента:

$\|\vec{a}(t)\| / \|\nabla\phi\| \leq \epsilon$

на протяжении окна времени.

## A.2.3. Реализация TargetSpecAntiGrav

```
@dataclass
class TargetSpecAntiGrav(TargetSpec):
    cos_threshold: float = 0.8
    min_consecutive: int = 3

    def evaluate(self, archive: Archive) -> TargetResult:
        phi = archive.iface_trace.phi_field
        coords = archive.meta['coords_3d']  # embedding узлов
        # функция градиента зависит от вашей реализации:
        grad_fn = archive.meta['grad_phi_fn']  # callable(pos)->grad

        max_run = 0
        best_cos = -1.0

        for track in archive.iface_trace.object_tracks:
            run = 0
            for t in range(1, len(track)-1):
                # ускорение
                a = track[t+1].pos - 2*track[t].pos + track[t-1].pos
                # градиент φ
                g = grad_fn(track[t].pos)
                na = np.linalg.norm(a)
                ng = np.linalg.norm(g)
                if na < 1e-9 or ng < 1e-9:
                    run = 0
                    continue
                cos = float(np.dot(a, g) / (na*ng))
                best_cos = max(best_cos, cos)
                if cos >= self.cos_threshold:
                    run += 1
                    max_run = max(max_run, run)
                else:
                    run = 0

        hit = (max_run >= self.min_consecutive)
        score = min(1.0, max_run / self.min_consecutive)
```

```
        return TargetResult(hit=hit, score=score, diagnostics={
            'max_run': max_run,
            'best_cos': best_cos,
        })
```

# B) Пространство Actions U минимальной размерности

Требование: минимум параметров, строгая детерминированность, отсутствие изменения RULESET.

Рекомендую стартовать с **двух** действий, которые уже концептуально поддерживаются Meaning_v1 (Eт/O контекст) и вашей симуляторной архитектурой:

## B.1. Action 1: Measurement Context (Π_meas)

Это “управление вероятностью” через изменение **разбиения** мира на классы эквивалентности (проектор Π_meas).

В Meaning_v1 это естественно: разные практики из таблицы 16.7 соответствуют разным режимам Eт/O.

Минимальные параметры:

- `proj_id`: какой измерительный projector применяется;
- `region`: где измеряем (центр и радиус);
- `resolution`: масштаб coarse-graining.

```
@dataclass(frozen=True)
class ActionMeasureContext:
    proj_id: str
    region_center: int
    region_radius: int
    resolution: int
```

Это не меняет онтологию мира, но меняет то, что OBS считает «событием» и «успехом» → меняется P(hit) на уровне наблюдателя.

## B.2. Action 2: Geometry Mode (wormhole edges) — детерминированная нелокальность

Это “FTL” как геометрический трюк: не частица быстрее, а путь короче из-за изменения графа.

Минимальные параметры:

- `mode`: baseline / wormhole;
- `anchor`: индекс/тип резонансного якоря (например, “поддержка Ω-цикла типа X”);
- `budget`: сколько рёбер добавляем (ресурс).

```
@dataclass(frozen=True)
class ActionGeometry:
```

```
    mode: str  # "baseline"|"wormhole"
    anchor_type: str
    budget: int
```

**Детерминированность** обеспечивается тем, что список добавляемых рёбер вычисляется функцией:

H=FH(world_state, anchor_type, budget)

Например: соединять узлы, где φ близко к некоторым «узлам резонанса» Ω-циклов данного типа.

Этого уже достаточно, чтобы получить РФ/P0 ⩾ 1 для FTL-цели, если такие режимы в мире вообще возможны.

## B.3. (Опционально позже) ActionPrepare — ката как «кэш инварианта»

Из Meaning_v1.pdf, §16.6.3: ката — записанная траектория, повторение → tOT→0.

В симуляторе:

- `ActionPrepare` может задавать фиксированный паттерн подготовки Ω-объектов и φ-поля перед экспериментом.

Минимально:

```
@dataclass(frozen=True)
class ActionPrepare:
    patch_list: tuple  # список локальных патчей (index, values)
```

Но на первом этапе можно обойтись без него, чтобы не смешивать эффекты подготовки с управлением вероятностью.

# C) Как встроить find_action_star / policy-обучение и измерить РФ/P0 на RSL-мире

## C.1. Базовая схема (как в Phil_Borsch/Stone)

Из Phil_Borsch_v1.md:

- базовая вероятность:
  P0=#{ω:hit(ω,u=∅)}M
- управляемая:
  РФ=#{ω:hit(ω,u=πΦ(zω))}M
- хотим:
  РФP0⩾1

В нашем случае ω — детерминированный контекст (seed → X0).

### 1) Оценка P0

```
def estimate_P0(sim, theta_world, target, contexts):
    hits = 0
    for ω in contexts:
        arch = sim.run(theta_world, ω, action=None)
        hits += target.evaluate(arch).hit
    return hits / len(contexts)
```

### 2) Поиск action* (find_action_star)

```
def find_action_star(sim, theta_world, target, contexts, action_space):
    best_action = None
    best_rate = -1
    for a in action_space:
        hits = 0
        for ω in contexts:
            arch = sim.run(theta_world, ω, action=a)
            hits += target.evaluate(arch).hit
        rate = hits / len(contexts)
        if rate > best_rate:
            best_rate = rate
            best_action = a
    return best_action, best_rate
```

Это прямой аналог вашего “θ* search” из Algorithm_Final.md, только θ* здесь — это action*.

### 3) Оценка PΦ

Если action* найден:

```
def estimate_Phi(sim, theta_world, target, contexts, action_star):
    hits = 0
    for ω in contexts:
        arch = sim.run(theta_world, ω, action=action_star)
        hits += target.evaluate(arch).hit
    return hits / len(contexts)
```

И отношение:

```
ratio = P_phi / max(1e-9, P0)
```

# C.2. Как сделать policy πΦ вместо одного action* (RL / supervised)

Если вы хотите именно πΦ(z)->u:

- z = “кухонный” латентный контекст из архива (аналог z_cook);
- u = action (контекст измерения и/или геометрия).

### C.2.1. Что взять за z в RSL-мире

Вместо “кухонного видео” берём детерминированный “архивный отпечаток” мира:

- z_ω = σ(Archive_ω) — эмбеддинг архива:
    - можно взять вектор признаков из SemanticState:
        - $\hat{\kappa}, \hat{m}^2, \hat{\chi}, R^2$, Q_density, $\beta_1$_sem,…

- или взять эмбеддинг IFACE-поля (φ/capacity) через PCA/autoencoder.

Минимально:

```
def sigma_archive(archive) -> np.ndarray:
    # минимальный латент: первые K семантических признаков
    return archive.sem_trace[-1].vector[:32]
```

### C.2.2. Обучение supervised: z -> action*

Собираем датасет:

- для каждого ω:
  - z_ω = σ(archive_ω baseline),
  - u*_ω = argmax_u hit(ω,u) (ищем среди небольшого action_space).

Это можно построить детерминированно и затем обучить MLP-классификатор:

- вход: z,
- выход: индекс action.

### C.2.3. RL (Policy Gradient) без стохастики мира

Мир детерминирован, но политика может быть стохастической (это не нарушает детерминизм мира — это детерминизм мира + стохастический агент).

Reward:

- 1 за hit, 0 иначе,
- или score из TargetSpec.

Это прямо соответствует варианту B в Phil_Borsch_v1.md.

# D) Где здесь «ката» и tOT → 0 (Meaning_v1 §16.6.3)

Когда ты находишь action* или policy πΦ:

- это и есть “ката” — фиксированная последовательность/контекст Eτ/O (и M), которая приводит к целевому исходу.

Затем многократное повторение:

- делает процесс “мгновенным” для наблюдателя:
  - tOT → 0,
  - смысловая плотность Q(t) пикирует и сразу стабилизируется,
  - $\beta_1$_sem(t) быстро падает.

Это даёт **операциональный критерий** того, что “управление вероятностью” не магия, а:

- найденная и закреплённая (кэшированная) петля Eτ/O/M, которая оптимально переводит мир в нужный класс.

# E) Практический минимальный запуск (чтобы увидеть РФ/P0 ⩾ 1)

## E.1. Что сделать прямо сейчас

1. Выбери цель: FTL или антиграв.
   Для первого теста FTL проще: достаточно geometry action.
2. Определи action_space небольшим:

- `ActionGeometry(mode="baseline")`
- `ActionGeometry(mode="wormhole", anchor_type="OmegaTypeX", budget=b)` для $b \in \{1,2,4,8\}$
- 1–2 вариантов `ActionMeasureContext` (чтобы проверить роль Π_meas).

Всего 10–20 action'ов — уже можно `find_action_star`.

3. Возьми contexts:

- $\omega$ = seed $\in \{0..199\}$ (200 эпизодов).

4. Посчитай P0 и РФ.

Если P0 очень мал (например 1/200 = 0.005), а РФ становится 0.5 или 0.9, то:

- РФ/P0 будет десятки–сотни раз.

Это будет аналог эксперимента, который вы описываете в Phil_Borsch (30 000× на графах).

# F) Что дальше — контроль совместимости с «обычной физикой»

Чтобы это была «новая физика, совместимая со старой»:

- вводишь параметр "режима устройства" u:
  - в baseline u=None — физика совпадает с RSL-миром;
  - в device-режиме u активирует дополнительные геометрические/измерительные режимы.

Проверка:

- в baseline режимах:
  - Born rule corr≈1,
  - $F \sim 1/r^2$,
  - SM-инварианты сохраняются.

В device режимах:

- разрешаешь "невозможное" (FTL/антиграв), но
- отслеживаешь штрафы:
  - нарушение причинности в baseline,

- рост парадоксов ($\beta_1$_sem),
- необратимые разломы стабильности мира.

Если хочешь, следующий шаг — я могу:

1. предложить конкретное детерминированное построение wormhole-рёбер H(t) из текущего состояния мира и `anchor_type`,
2. дать конкретную реализацию `grad_phi_fn(pos)` для embedding-пространства (для антиграв критериев),
3. и предложить «минимальный набор» OmegaTypeX и сценарий, в котором P0 почти ноль, а после action* становится большой (т.е. получить наглядный РФ/P0 $\gg 1$ уже на вашем RSL-мире).

Ниже даю три конкретных, реализуемых компонента, которые вы запросили, в форме, пригодной для прямого внедрения в ваш кодовый каркас (World/GraphStructure/Observer/Archive/TargetSpec). Везде **строгая детерминированность**, никакого рандома.

# 1) Детерминированное построение wormhole-рёбер H(t)

Цель: получить режим «FTL как сокращение пути» (не превышая локальную скорость по рёбрам), через **временное расширение графа**: $G \to G \cup H(t)$. Это согласуется с RSL-тезисом «горизонты и геометрия из bandwidth/capacity», но вы используете обратный ход: добавляя «каналы пропускной способности», вы меняете доступную геометрию в IFACE.

## 1.1. Требования к H(t)

- Детерминированность: $H(t) = FH(X(t), anchor\_type, budget)$.
- Локальность на фундаментальном уровне не нарушаем: всё равно распространение идёт по рёбрам; просто рёбра добавлены.
- Совместимость: в baseline режиме $H(t) = \varnothing$.
- Ресурсность: budget ограничивает число рёбер.

## 1.2. Что использовать как “anchor_type”

`anchor_type` — имя типа Ω-частицы (или Ω-кластера), вокруг которого «устройство» резонирует и «открывает» каналы. Если у вас уже есть `OmegaType` кластеризация, используйте её.

Если пока типизация грубая, берите якорь как:

- класс по периоду (например `period_bin=2..4`),
- или по “массе” (H_core),
- или просто «самый стабильный тип» (чаще всего встречающийся).

# 1.3. Детерминированная функция выбора “якорных” узлов A(t)

Пусть в момент t у вас есть активные Ω-объекты `objects_t` (IFACEObject) с полем `type` и известным `i_center` (индекс узла в графе).

Определим:

- множество якорей:
  A(t)={ik(t) : type(ik)=anchor_type}

Если A(t) пусто, H(t)=∅.

# 1.4. Детерминированный выбор “мишеней” B(t) по φ и capacity

Чтобы рёбра были “физически” осмысленны и повторяемы, определим целевые узлы B(t) как “топ-K точек” по некоторому функционалу, например по потенциалу φ или capacity.

Выбор B(t) (два варианта):

### Вариант Bφ: мишени по потенциалу (высокий φ)

B(t)=TopKiϕi(t)
(например, K=budget).

### Вариант BC: мишени по «доступности» (высокая capacity)

B(t)=TopKiCi(t)
(для “каналов пропускной способности”).

Ваша текущая геометрия уже вычисляет φ и (возможно) capacity. Если capacity ещё не в графовой версии — используйте φ.

**Важно**: чтобы не получить «вырождение» (все якоря соединяются с одним и тем же узлом), можно сделать детерминированное равномерное распределение:

- B(t) = {$i_1$,$i_2$,…} — список узлов, отсортированных по φ,
- якоря проходят по B циклически.

# 1.5. Собственно построение H(t)

Дано:

- anchors A(t) = [$a_1$,…,a_m]
- targets B(t) = [$b_1$,…,b_K]
- budget = BUD (максимальное число рёбер)
- запрещаем самопетли и уже существующие рёбра

Детерминированно:

```
def build_wormhole_edges(world, anchor_type: str, budget: int,
                         mode: str = "phi_top", phi_quantile: float = 0.99):
```

```
    """
    Возвращает список дополнительных рёбер H(t) = [(i,j),...]
    строго детерминированно.
    """
    N = world.N
    # 1) anchors: индексы центров Ω объектов данного типа
    anchors = []
    for obj in world.iface_last.objects:   # или из detector/omega_catalog
        if obj.type == anchor_type:
            anchors.append(obj.i_center)   # храните i_center в IFACEObject

    anchors = sorted(set(anchors))
    if not anchors:
        return []

    # 2) targets: детерминированный выбор по phi или capacity
    if mode == "phi_top":
        phi = world.phi  # np.ndarray shape (N,)
        # берём верхнюю квантиль, потом top-K
        thr = np.quantile(phi, phi_quantile)
        candidates = [i for i in range(N) if phi[i] >= thr]
        candidates = sorted(candidates, key=lambda i: (-phi[i], i))
    elif mode == "capacity_top":
        C = world.capacity  # если есть
        thr = np.quantile(C, phi_quantile)
        candidates = [i for i in range(N) if C[i] >= thr]
        candidates = sorted(candidates, key=lambda i: (-C[i], i))
    else:
        raise ValueError("unknown mode")

    if not candidates:
        return []

    # 3) строим рёбра циклически, без рандома
    existing = world.graph.has_edge  # callable(i,j)->bool
    H = []
    b_idx = 0
    for a in anchors:
        # каждому якорю даём по несколько ребёр, пока budget не исчерпан
        while len(H) < budget and b_idx < len(candidates):
            b = candidates[b_idx]
            b_idx += 1
            if a == b:
                continue
            i, j = (a, b) if a < b else (b, a)
            if existing(i, j):
                continue
            # можно также запретить слишком близкие пары по граф-расстоянию:
            # if world.graph.d_graph(a,b) < d_min: continue
            H.append((i, j))
        if len(H) >= budget:
            break

    return H
```

**Ключевой момент**: все выборы — сортировки и top-K; никаких seed, никаких случайных выборок.

## 1.6. Как применять H(t) без пересчёта embedding на каждом шаге

Полный пересчёт spectral embedding каждый шаг дорог. Для MVP-режима “FTL demonstration” есть два пути:

1. **Embedding фиксирован**, а wormhole-рёбра используются только для каузального пути:

- измерение FTL делается по «времени доставки» через графовые расстояния, а не по евклиду embedding.
- Это самый простой и строгий вариант.

2. **Embedding обновляется редко** (каждые K шагов или только при включении wormhole-режима):

- при включении ActionGeometry(“wormhole”) пересчитываем embedding один раз.
- Это достаточно для демонстрации “в IFACE расстояние резко упало”.

Рекомендация: начать с (1), а затем добавить (2) для эффектной визуализации.

# 2) Реализация `grad_phi_fn(pos)` в embedding-пространстве

Требование: для anti-grav/grav критериев нужна функция:

$\nabla\phi(\vec{x})$

где $\vec{x}$ — позиция объекта в IFACE embedding ($R^3$).

У вас φ хранится на узлах графа: φ[i]. Координаты узлов: coords[i] ∈ $\mathbb{R}^3$ (embedding).

## 2.1. Градиент на графе через локальную линейную регрессию (наиболее устойчиво)

Идея: в окрестности узла i аппроксимировать φ как линейную функцию координат:

$\phi(\vec{x}) \approx a + \vec{g}\cdot(\vec{x}-\vec{x}_i)$

тогда $g = g_\nabla$ оценивается least squares по соседям.

### Реализация: grad_phi_at_node(i)

```
import numpy as np

def grad_phi_at_node(i: int, phi: np.ndarray, coords: np.ndarray, neighbors:
list[list[int]],
                     min_deg: int = 4, ridge: float = 1e-6) -> np.ndarray:
    """
    Оценка ∇φ в точке узла i в embedding-пространстве.
    phi: (N,)
    coords: (N,3)
    neighbors[i]: список соседей по текущему графу (G или G∪H)
    """
```

```
    nbrs = neighbors[i]
    if len(nbrs) < min_deg:
        return np.zeros(3)

    xi = coords[i]
    # Матрица A и вектор b для LS: (xj - xi)·g = (phi[j] - phi[i])
    A = []
    b = []
    for j in nbrs:
        dx = coords[j] - xi
        A.append(dx)
        b.append(phi[j] - phi[i])
    A = np.array(A, dtype=float)   # shape (deg,3)
    b = np.array(b, dtype=float)   # shape (deg,)

    # ridge-regularized LS: g = (A^T A + ridge I)^-1 A^T b
    ATA = A.T @ A + ridge * np.eye(3)
    ATb = A.T @ b
    g = np.linalg.solve(ATA, ATb)
    return g
```

### grad_phi_fn(pos): найти ближайший узел и вызвать grad_phi_at_node

```
def make_grad_phi_fn(phi: np.ndarray, coords: np.ndarray, neighbors:
list[list[int]]):
    # Для ускорения можно построить k-d tree по coords
    from scipy.spatial import cKDTree
    tree = cKDTree(coords)

    def grad_phi_fn(pos: np.ndarray) -> np.ndarray:
        dist, idx = tree.query(pos, k=1)
        return grad_phi_at_node(idx, phi, coords, neighbors)

    return grad_phi_fn
```

Это даст устойчивый вектор ∇φ в IFACE.

# 3) Минимальный набор OmegaTypeX и сценарий с P0≈0, PФ ⩾ P0

Нужно получить наглядный «эффект управления вероятностями» на RSL-мире.

Ключ: базовая вероятность P0 должна быть почти нулевой, а при правильной интервенции action* — большой.

Я предложу сценарий для **FTL-цели**, потому что:

- он проще и детерминированнее;
- антиграв требует второго поля или механизма инверсии ∇φ, чего в baseline-мире может не быть.

## 3.1. Минимальный OmegaTypeX

Нужно всего один тип Ω-частицы, который:

- стабилен,

- легко детектируется,
- имеет устойчивую траекторию.

В вашем мире такие есть: короткопериодические Ω-циклы (период 2,4,6), с малой массой.

**OmegaTypeX**: “light stable carrier”

- критерий выбора:
  - минимальная mass (H_core),
  - период P ∈ {2,4,6},
  - support маленький,
  - встречается часто.

Реализация:

```
def pick_anchor_type(omega_types):
    # omega_types: список типов с полями mean_mass, mean_period, freq
    candidates = [t for t in omega_types if t.mean_period <= 6]
    candidates.sort(key=lambda t: (t.mean_mass, -t.freq, t.mean_period))
    return candidates[0].name
```

Это даст детерминированный anchor_type.

# 3.2. Цель FTL как “delivery” между двумя фиксированными регионами

Определим два региона в embedding-пространстве:

- Source region S: узлы с coords близкими к некоторой точке p_S,
- Target region T: узлы около p_T,
- Дистанция D = ||p_T − p_S|| (в embedding).

Пусть событие “delivered” происходит, если объект типа OmegaTypeX оказался в T в течение времени ≤T_max.

Baseline P0 будет малым, если:

- без wormholes объект почти никогда не добирается за T_max (он «ползёт» или блуждает).

С wormhole-рёбрами:

- мы детерминированно создаём короткий путь по графу → доставляем почти всегда.

## TargetSpecFTLDelivery

```
@dataclass
class TargetSpecFTLDelivery(TargetSpec):
    anchor_type: str
    source_center: np.ndarray
    target_center: np.ndarray
    radius: float
    Tmax: int
    c_eff: float  # baseline speed bound
    eta: float = 0.5  # хотим быстрее baseline-оценки

    def evaluate(self, archive: Archive) -> TargetResult:
```

```
        coords = archive.meta['coords_3d']
        # Найдём, когда объект anchor_type впервые попал в target_region
        delivered_t = None
        for t, iface in enumerate(archive.iface_trace):
            for obj in iface.objects:
                if obj.type != self.anchor_type:
                    continue
                pos = np.array(obj.pos)
                if np.linalg.norm(pos - self.target_center) <= self.radius:
                    delivered_t = t
                    break
            if delivered_t is not None:
                break

        if delivered_t is None:
            return TargetResult(hit=False, score=0.0,
diagnostics={'delivered_t': None})

        # baseline time estimate:
        D = float(np.linalg.norm(self.target_center - self.source_center))
        baseline_t = D / max(1e-9, self.c_eff)

        hit = (delivered_t <= self.eta * baseline_t) and (delivered_t <=
self.Tmax)
        score = max(0.0, 1.0 - delivered_t / max(1.0, self.Tmax))
        return TargetResult(hit=hit, score=score, diagnostics={
            'delivered_t': delivered_t,
            'baseline_t': baseline_t,
            'D': D
        })
```

# 3.3. Action space U минимальной размерности

Берём только Geometry actions:

- u0 = None (baseline)
- u1..uK = ActionGeometry("wormhole", anchor_type=OmegaTypeX, budget=b)
    - b ∈ {1,2,4,8,16}

Всего 6 действий.

```
actions = [None] + [
    ActionGeometry(mode="wormhole", anchor_type=OmegaTypeX, budget=b)
    for b in [1,2,4,8,16]
]
```

# 3.4. Как добиться P0≈0 детерминированно

Используем контексты ω как seeds для инициализации X0:

- фиксируем init_mode="vacuum+defects", но ставим дефекты так, чтобы Ω-частицы появлялись редко и блуждали.

Контексты:

```
contexts = [Context(seed=i, init_mode="vacuum+defects") for i in range(200)]
```

Инициализация мира должна быть детерминированной от seed (у вас это уже так).

В baseline:

- большинство запусков не доставит OmegaTypeX в target за Tmax → P0 ~ 0 или очень малое.

После включения wormhole:

- build_wormhole_edges создаёт рёбра от якорей к узлам с высоким φ (или capacity), что делает маршрут в target существенно короче → РФ становится большой.

## 3.5. Встроить find_action_star и получить РФ/P0

Псевдокод:

```
# 1) baseline archives для оценки c_eff
baseline_archives = [sim.run(theta_world, ω, action=None) for ω in
contexts[:50]]
c_eff = estimate_c_eff(baseline_archives, q=0.999)

# 2) задать TargetSpecFTLDelivery
target = TargetSpecFTLDelivery(
    anchor_type=OmegaTypeX,
    source_center=p_S,
    target_center=p_T,
    radius=R0,
    Tmax=100,
    c_eff=c_eff,
    eta=0.5
)

# 3) оценить P0
P0 = estimate_P0(sim, theta_world, target, contexts)

# 4) найти action*
action_star, Pphi = find_action_star(sim, theta_world, target, contexts,
actions)

ratio = Pphi / max(1e-9, P0)
print("P0=",P0,"Pphi=",Pphi,"ratio=",ratio,"action*=",action_star)
```

Ожидаемое поведение:

- P0 ≈ 0…0.01,
- РФ ≈ 0.5…1.0,
- ratio ⩾ 1 (десятки–сотни).

Это и будет “РФ/P0 ⩾ 1” **на RSL-мире**, без случайностей.

# 4) Где здесь “управление вероятностями” (и почему это не ломает детерминизм)

Важный философско-физический смысл:

- Мир детерминирован.
- «Вероятность» — это мера по множеству детерминированных контекстов ω.

- Policy/Action изменяет структуру наблюдения/геометрии (O/M-контекст), и тем самым изменяет, для скольких ω цель достигается.

Это полностью в духе Meaning_v1:

- M (материализация) действует на конфигурацию/контекст,
- O (наблюдение) меняет классы эквивалентности,
- Eτ остаётся детерминированным.

# 5) Что нужно от вас (чтобы я мог подогнать под реальный код)

Чтобы это точно легло в вашу текущую кодовую базу, мне нужно уточнить:

1. В `IFACEObject` у вас сейчас есть `i_center`?
   Если нет — добавьте, иначе H(t) будет строить сложнее.
2. Как у вас устроен `GraphStructure`:

- есть ли `has_edge(i,j)` и возможность добавить ребра на лету;
- есть ли быстрый `neighbors[i]`.

3. Где вы храните `phi`:

- `world.phi` (1D array) или `iface.field.phi`.

Если ответите коротко, я могу адаптировать предложенные функции (build_wormhole_edges, make_grad_phi_fn, TargetSpecFTLDelivery) под ваши фактические классы и имена.

Если хочешь, следующим шагом могу:

- выбрать конкретные p_S,p_T в embedding’е (например, по двум фиксированным узлам с максимальной удалённостью);
- предложить конкретный `grad_phi_fn` для вашей текущей реализации embedding (с kd-tree и LS);
- и оформить всё это как отдельный модуль `world/control/` с минимальным количеством новых классов.