オンライン麻雀 公平性検証エンジン (MFVE)

詳細設計書 - MFVE-MOD-A: 静的公平性検証モジュール

作成日: 2025 年 10 月 21 日
作成者: Gemini 2.5 Pro Canvas
バージョン: 1.0

1. 概要

1.1. 本モジュールの目的

本モジュール (ID: MFVE-MOD-A) は、オンライン麻雀プラットフォームの「静的公平性」を検証することを目的とする。具体的には、観測された実対局ログから得られる指標（配牌品質スコア HQS 等）の統計的分布が、大規模シミュレーションによって生成された「数学的に完全に公平な」理論分布と一致するかを、客観的な数値をもって判定する。

本設計は、特に以下の実務的要件を達成するものである。

- **ブラックボックス監査:** プラットフォームの内部ロジック（ソースコード）を開示することなく、公開されたアウトプット（対局ログ）のみを用いて検証する。
- **メモリ効率:** 数百万局から数十億件に及ぶ巨大なデータを扱うため、メモリオーバーフローを回避する設計とする。
- **統計的堅牢性:** 統計的有意性（P 値）と実務的有意性（効果量 D）を分離評価することで、巨大サンプルサイズに起因する誤判定を防ぐ。

1.2. 参照文書

- ブラックボックス・アルゴリズムに対する自己監査プロトコル.pdf
- オンライン麻雀ゲーム競技の信頼性向上_清田達.pdf
- 麻雀ゲームにおける運と実力の境界.pdf

2. インターフェース定義

2.1. 入力

本モジュールは、集計済みのヒストグラム（度数配列）と設定オブジェクトを入力として受け取る。

引数名	型	説明
observed_hist	Numpy ndarray (1 次元)	観測データ（実対局ログ）から集計された指標のヒストグラム。

theoretical_hist	Numpy ndarray (1 次元)	理論データ（シミュレーション）から集計された指標のヒストグラム。 ※ **observed_hist とビン定義（幅・数が完全に一致していること。**
config	辞書 (dict)	検定の挙動を制御する設定オブジェクト。

config オブジェクト詳細

キー	型	説明	設定例
alpha	float	統計的有意水準（第一種過誤の許容率）。	0.01
effect_size_threshold_D	float	効果量 D の実務的な許容閾値。この値は専門家（プロ雀士等）との協議に基づき、ドメイン知識によって決定される。	0.05
min_expected_freq	float	カイ二乗検定の統計的妥当性を担保するための、各ビンの最小期待度数。この値を下回るビンは動的にマージされる。	5.0

2.2. 出力

本モジュールは、検証結果を格納した辞書オブジェクトを返す。

キー	型	説明
status	str	総合判定。**'合格', '実用上公平', '不合格'** のいずれか。
summary	str	判定の根拠を平易に説明するサマリーテキスト。
p_value	float	カイ二乗検定によって算出された P 値（多重検定補正前）。
chi2_statistic	float	カイ二乗検定統計量。
ks_statistic_D	float	ビン化された CDF から算出した効果量（KS 統計量 D の近似値）。

merged_bins_count	int	カイ二乗検定実行時に使用された（動的マージ後の）最終的なビン数。
is_p_significant	bool	P 値が有意水準 α 以下かどうかのフラグ。
is_d_significant	bool	効果量 D が実務的閾値以上かどうかのフラグ。

3. 内部処理ロジック

本モジュールは、P 値の厳密性を担保する「**カイ二乗検定パス**」と、効果量の解像度を担保する「**KS 統計量パス**」を並行して実行し、その結果を統合して最終判定を下す。

3.1. 前処理：期待度数の算出

1. 観測データの合計度数 N_{obs} を算出する。
2. 理論データの合計度数 N_{theo} を算出する。
3. 期待度数配列 f_{exp} を算出する。

   $f_{exp} = \{theoretical\_hist / N_{theo}\} \times N_{obs}$

3.2. カイ二乗検定パス (P 値の算出)

目的: 統計的に厳密な P 値を算出するため、カイ二乗検定の前提条件（各ビンの期待度数が一定以上）を満たすようにビンを動的にマージする。

1. **動的ビンマージ (Dynamic Bin Merging) の実行:**
   - 期待度数配列 f_{exp} と観測度数配列 observed_hist のコピーを作成する。
   - 配列の両端（HQS が極端に低い/高いビン）から中央に向かってスキャンする。
   - 期待度数が config['min_expected_freq'] を下回るビンを見つけた場合、隣接するビンと統合（度数を加算）し、ビンの総数を減らす。
   - すべてのビンの期待度数が基準を満たすまで、このマージ処理を繰り返す。
2. **カイ二乗適合度検定の実行:**
   - マージ後の度数配列を用いて、scipy.stats.chisquare を実行する。
   - chi2_statistic と p_value を取得する。
   -

3.3. KS 統計量パス (効果量 D の算出)

目的: 効果量 D の近似精度を最大化するため、マージ前のオリジナル（高解像度）のヒストグラムを使用する。

1. **CDF（累積分布関数）の作成:**
   - オリジナルの observed_hist から観測 CDF (cdf_observed) を算出する。
   - オリジナルの theoretical_hist から理論 CDF (cdf_theoretical) を算出する。
2. **効果量 D の算出:**

- 2 つの CDF の差の絶対値の最大値を計算する。

  D ＝ max(|cdf_observed － cdf_theoretical |)

3.4. 総合判定 (3 値判定ロジック)

条件	判定ステータス	サマリー概要
P ＞ α	**合格**	観測分布と理論分布に統計的有意差は検出されなかった。
P ≦ α **かつ** D ＜ 閾値	**実用上公平**	統計的有意差は検出されたが、その差の大きさ（効果量）は実務上無視できる微小な範囲内である。
P ≦ α **かつ** D ≦ 閾値	**不合格**	統計的有意差が検出され、かつその差の大きさも実務上無視できないレベルであるため、要調査。

4. 理論分布生成の標準仕様

4.1. 基本方針

本検証システムにおける「理論分布」は、検証対象アプリの（未知の）乱数生成器を模倣するものではなく、そのレギュレーションにおける「数学的に完全に公平な」を定義するベンチマーク（ものさし）である。

したがって、理論分布の生成には、暗号学的に安全で、統計的な偏りがないことが証明されている標準的なアルゴリズムを採用する。

4.2. 標準アルゴリズム

- **乱数生成器 (RNG):** NIST SP 800-90A に準拠する HMAC-DRBG (SHA-256) を採用する。シード値を指定することで、生成される乱数列の完全な再現性を担保する。
- **シャッフル:** Fisher–Yates shuffle アルゴリズムを採用する。
- **整数生成のバイアス排除:** 上記シャッフル内で使用する整数は、Rejection Sampling（棄却サンプリング）により、範囲指定に伴う統計的バイアスを完全に排除する。

監査可能性への貢献:

この標準仕様は、内部的な理論分布生成に用いるだけでなく、各プラットフォーム事業者が導入を推奨される「監査可能な公平性モデル (FMSF 構想)」の技術的基盤となる。事業者がこの仕様を導入し、対局ログと共にシード値を公開することで、第三者による決定論的な結果検証（＝不正がないことの直接証明）が可能となる。

参考資料

1．ベンチマーク牌山生成モジュール

import hmac
import hashlib
from typing import List, Tuple, Optional

----- HMAC-DRBG (SHA-256) -----

class HmacDRBG:
"""
NIST SP 800-90A 準拠の簡易 HMAC-DRBG (SHA-256) 実装。
監査用に seed_material を保持し、再現性を担保。
"""
def **init**(self, seed_material: bytes):
self._K = b"¥x00" * 32 # key (32 bytes for SHA-256)
self._V = b"¥x01" * 32 # value
self.seed_material = seed_material
self._update(seed_material)

```
def _hmac(self, key: bytes, data: bytes) -> bytes:
    return hmac.new(key, data, hashlib.sha256).digest()

def _update(self, provided_data: Optional[bytes]):
    # Step 1
    self._K = self._hmac(self._K, self._V + b"¥x00" + (provided_data or b""))
    self._V = self._hmac(self._K, self._V)
    # Step 2
    if provided_data:
        self._K = self._hmac(self._K, self._V + b"¥x01" + provided_data)
        self._V = self._hmac(self._K, self._V)

def reseed(self, entropy: bytes):
    self._update(entropy)

def random_bytes(self, n: int) -> bytes:
    out = b""
    while len(out) < n:
```

```
        self._V = self._hmac(self._K, self._V)
        out += self._V
    return out[:n]


def randint(self, low: int, high: int) -> int:
    """
    Return an integer in [low, high], unbiased via rejection sampling.
    """
    assert low <= high
    span = high - low + 1
    # compute bits needed
    bits = max(1, (span - 1).bit_length())
    # rejection sampling
    while True:
        # bytes needed to cover bits
        nbytes = (bits + 7) // 8
        rb = int.from_bytes(self.random_bytes(nbytes), "big")
        r = rb & ((1 << bits) - 1)
        if r < span:
            return low + r
```

----- Mahjong tile set -----

```
def build_tiles(include_red: bool = False) -> List[str]:
"""
日本麻雀の牌セットを返す。
include_red=False: 136 枚（赤なし）
include_red=True : 136 枚 + 赤 5 (m,p,s) 各 1 枚で計 139 枚
"""
tiles = []
# 数牌: 萬(1-9)m, 筒(1-9)p, 索(1-9)s 各 4 枚
for suit in ["m", "p", "s"]:
for num in range(1, 10):
tiles.extend([f"{num}{suit}"] * 4)
# 字牌: 東南西北 白發中 各 4 枚
for honor in ["E", "S", "W", "N", "P", "F", "C"]:
tiles.extend([honor] * 4)
```

```
if include_red:
    # 赤 5: それぞれの 5 から 1 枚を赤に置換（慣習により通常は各 1 枚）
    # ここでは「5m」「5p」「5s」の 1 枚を「5mR」「5pR」「5sR」に差し替え
    def replace_one(tiles: List[str], normal: str, red: str):
        idx = tiles.index(normal)  # 少なくとも 4 枚ある前提
        tiles[idx] = red

    replace_one(tiles, "5m", "5mR")
    replace_one(tiles, "5p", "5pR")
    replace_one(tiles, "5s", "5sR")

return tiles
```

----- Fisher–Yates shuffle (unbiased) -----

def fisher_yates_shuffle(items: List[str], rng: HmacDRBG) -> None:
"""
インプレースで完全均等分布のシャッフル。
"""
n = len(items)
for i in range(n - 1, 0, -1):
j = rng.randint(0, i)
items[i], items[j] = items[j], items[i]

----- Wall building for Mahjong -----

def build_wall(shuffled_tiles: List[str]) -> List[List[str]]:
"""
牌山（34 列 x 4 枚）を構築。実物卓の「配列」概念に合わせた形式。
出力: 34 スタック（左から右）、各スタックは上から順に 2 枚+2 枚の積み上げに相当。
実運用に合わせて取り出し規約を別途定義してください。
"""
n = len(shuffled_tiles)
if n % 4 != 0:
raise ValueError("Tile count must be divisible by 4 to form stacks.")
stacks = []
for i in range(0, n, 4):

stacks.append(shuffled_tiles[i:i+4])
return stacks # 34 stacks for 136 枚（赤なし）

----- Public API -----

def shuffle_mahjong_wall(seed: bytes,
include_red: bool = False,
reseed_with: Optional[bytes] = None
) -> Tuple[List[str], List[List[str]]]:
"""
麻雀向けシャッフルの公開関数。
- seed: 監査用の初期シード（例: 卓上乱数器のログ、時刻＋卓 ID のハッシュ等）
- include_red: 赤ドラの有無
- reseed_with: 追加エントロピー（卓操作ログ等）を合成して再シード可能
戻り値: (平列化されたシャッフル済み牌リスト, 牌山スタック構造)
"""
rng = HmacDRBG(seed_material=seed)
if reseed_with:
rng.reseed(reseed_with)

```
tiles = build_tiles(include_red=include_red)
fisher_yates_shuffle(tiles, rng)
wall = build_wall(tiles)
return tiles, wall
```

----- Example usage -----

if **name** == "**main**":
# 監査可能な固定シード（例）。運用時は OS エントロピーや卓ログのハッシュを推奨。
seed = hashlib.sha256(b"table-001|2025-10-09T05:24:00+09:00|dealer:E").digest()
# 任意の追加エントロピー（卓操作ログやサイコロ写真のハッシュ等）
reseed = hashlib.sha256(b"manual-cut|dice:5,6|camera-proof").digest()

```
tiles, wall = shuffle_mahjong_wall(seed=seed, include_red=True, reseed_with=reseed)

# 平列化出力（上家からの取り出し順に使う等、規約に合わせて解釈）
print("Shuffled tiles:", tiles)
print("Wall stacks (34 x 4):")
```

```
for idx, stack in enumerate(wall, start=1):
    print(f"{idx:02d}: {stack}")
```

２．HQS の算出: 論文 に基づく HQS の計算ロジック

※シャンテン数計算ロジックは未実装

```
import numpy as np
from typing import List, Dict, Tuple

# --- 1. ユーティリティ (牌の変換) ---

def create_tile_map() -> Dict[str, int]:
    """
    2 文字の牌文字列を、シャンテン計算用のインデックス(0-33)に変換する
    辞書を作成する。
    '0m', '0p', '0s' (赤ドラ) は、通常の '5m', '5p', '5s' と同じインデックスに
    マッピングする。

    (0-8: 萬子 1-9, 9-17: 筒子 1-9, 18-26: 索子 1-9, 27-33: 字牌 1-7)
    """
    tile_map = {}
    suits = ['m', 'p', 's']
    honors = ['z']

    offset = 0
    # 萬子、筒子、索子 (1-9)
    for suit in suits:
        for num in range(1, 10):
            tile_map[f'{num}{suit}'] = offset + num - 1
        offset += 9

    # 字牌 (1-7: 東南西北白發中)
    for num in range(1, 8):
        tile_map[f'{num}{honors[0]}'] = offset + num - 1

    # 赤ドラの特殊マッピング
```

```
    tile_map['0m'] = tile_map['5m'] # 0m -> 5m (index 4)
    tile_map['0p'] = tile_map['5p'] # 0p -> 5p (index 13)
    tile_map['0s'] = tile_map['5s'] # 0s -> 5s (index 22)

    return tile_map

# 処理中に何度も使うため、グローバル定数として定義
TILE_STRING_TO_INDEX: Dict[str, int] = create_tile_map()


def convert_to_hand_vector(tile_strings: List[str]) -> List[int]:
    """
    13 枚の牌文字列リストを、34 種の牌の枚数をカウントする
    リスト（長さ 34）に変換する。
    """
    hand_vector = [0] * 34
    for tile in tile_strings:
        if tile in TILE_STRING_TO_INDEX:
            hand_vector[TILE_STRING_TO_INDEX[tile]] += 1
        # else:
            # 不明な牌（エラーハンドリング）
    return hand_vector


# --- 2. 牌山からの配牌生成 ---

def parse_paishan(wall_str: str) -> List[str]:
    """2 文字ずつの牌山文字列を、牌のリストに変換する。"""
    if len(wall_str) % 2 != 0:
        raise ValueError("牌山文字列の長さが奇数です。")

    return [wall_str[i:i+2] for i in range(0, len(wall_str), 2)]

def deal_hands(paishan: List[str]) -> Dict[str, List[str]]:
    """
    牌山リストから、指定されたロジックで 4 人分の配牌を生成する。
    (4 枚 x3 巡、その後 1 枚 x1 巡)
    """
```

```
    if len(paishan) < 52:
        raise ValueError("配牌を行うには牌山が不足しています（最低 52 枚必要）。")

    hands = {f'player_{i}': [] for i in range(4)}
    wall_index = 0

    # 1. 4 枚ずつ 3 回（計 12 枚）
    for _ in range(3):
        for i in range(4): # 4 人のプレイヤー
            hand = hands[f'player_{i}']
            hand.extend(paishan[wall_index : wall_index + 4])
            wall_index += 4

    # 2. 1 枚ずつ 1 回（計 13 枚）
    for i in range(4):
        hands[f'player_{i}'].append(paishan[wall_index])
        wall_index += 1

    return hands

# --- 3. HQS の算出（★パフォーマンス上の注意点） ---

def calculate_shanten_stub(hand_vector: List[int]) -> int:
    """
    【重要】シャンテン数計算（スタブ）

    シャンテン数の計算は、麻雀のルール（七対子、国士無双、通常手）を
    すべて考慮する必要があり、非常に複雑なアルゴリズムです。

    100 万件の処理速度を達成するには、この関数を
    C++, Rust, C# などで記述された高速な専用ライブラリに
    置き換えることが【必須】です。

    ここでは、仮のロジックとして固定値（平均的な値）を返します。
    """
    # (ここに七対子、国士無双、一般手のシャンテン数を計算する
    #  再帰的なロジックが入るが、Python では非常に遅い)
```

```
    # 仮の値を返す（例：平均 6 シャンテン）
    return 6

def calculate_hqs(tile_strings: List[str]) -> Tuple[int, int]:
    """
    論文の例に基づき、配牌品質スコア(HQS)を計算する。

    HQS 例：(14 - シャンテン数 * 2) + ドラ枚数 [cite: 174]

    Returns:
        (HQS, シャンテン数) のタプル
    """

    # 1. 赤ドラの枚数をカウント
    # (ここでは牌山固有のドラであり、局のドラ表示牌は考慮しない)
    aka_dora_count = tile_strings.count('0m') + ¥
                     tile_strings.count('0p') + ¥
                     tile_strings.count('0s')

    # 2. シャンテン計算用に手牌をベクトル化
    hand_vector = convert_to_hand_vector(tile_strings)

    # 3. シャンテン数を計算（★最重要ボトルネック）
    shanten = calculate_shanten_stub(hand_vector)

    # 4. 論文[cite: 174]の例示式に基づき HQS を算出
    hqs = (14 - shanten * 2) + aka_dora_count

    return hqs, shanten


# --- 4. メイン処理（100 万回呼び出す関数）---

def process_single_wall_for_hqs(wall_str: str) -> Dict[str, Dict]:
    """
    単一の牌山文字列を受け取り、4 人分の HQS とシャンテン数を計算する。
    この関数が 100 万回呼び出される想定。
    """
```

```
    try:
        paishan = parse_paishan(wall_str)
        hands_dict = deal_hands(paishan)

        hqs_results = {}
        for player, hand_tiles in hands_dict.items():
            hqs, shanten = calculate_hqs(hand_tiles)
            hqs_results[player] = {
                'hqs': hqs,
                'shanten': shanten,
                'hand_str': " ".join(hand_tiles) # 確認用
            }

        return hqs_results

    except ValueError as e:
        print(f"処理エラー: {e}")
        return None
    except Exception as e:
        print(f"不明なエラー: {e}")
        return None

# --- 実行例 ---
if __name__ == "__main__":

    # ユーザー提供の牌山データ
    paishan_str =
"1m7m1p3p8m1p3p4p2m3m6m7m1m9m9m6p7p2s4s4s4p7p9p2s1p0p8p9p8p3s3s5s4s7s7s2z2s5s6s2z3s6s7s9s5s3z4z6z3
z3z7z7z5z6p9m6p6m3m5m3p7m8s4p8p2m1s6z7m6s3s5p1z8m4m3m1m6p8m5p1s4p9m8m2z4m4z1z6m5z9s2p4m9p1p2z2p5m6
z1z2m5p3m3p1s0m8s1s5z1z5m6z7z9s9p2s6s7p7s4s4m6m9s2m8p3z7z2p4z2p1m5z7p8s4z0s8s"

    print("--- HQS (配牌品質スコア) の算出実行 ---")
    # この関数を 100 万回ループで呼び出す
    hqs_data = process_single_wall_for_hqs(paishan_str)

    if hqs_data:
        for player, data in hqs_data.items():
            print(f"[{player}]")
```

```
        print(f"  Hand: {data['hand_str']}")
        print(f"  Shanten: {data['shanten']} (スタブ値)")
        print(f"  HQS: {data['hqs']}")

print("¥n--- 100 万件処理のパフォーマンスについて ---")
print("上記の 'process_single_wall_for_hqs' 関数を 100 万回呼び出します。")
print("牌のパース('parse_paishan')や配牌('deal_hands')は高速です。")
print("実用上の性能は、'calculate_shanten_stub' 関数の実装に完全に依存します。")
print("100 万局を現実的な時間（数分〜数十分）で処理するには、")
print("このシャンテン計算部分を Python 以外の高速な言語（C++, Rust 等）で")
print("実装したライブラリを呼び出す必要があります。")
```