意味的ワードクラウド アルゴリズム解説

意味的ワードクラウド (Semantic Word Cloud) アルゴリズム解説

項目	内容
作成者	荻 多加之
作成支援	Claude Code (Anthropic)
作成日	2024-12-25
最終更新	2026-02-16
バージョン	1.5

v1.0: 初版作成 v1.1: レイアウトパラメータ調整、CSV自動出力、回転制限追加 v1.2: SVG出力追加、出力ファイル名の日付自動付与 v1.3: MDS配置オプション追加、CSV出力にメタ情報・座標・色を追加 v1.4: 数学的解析セクション追加（埋め込み統計、PCA分布特性、オイラー法収束） v1.5: PPTX出力対応、フォントを游ゴシック(YuGothic Bold)に統一、レイアウトパラメータ調整

このドキュメントでは、`semantic_wordcloud.py` の動作原理を詳しく解説します。

---

1. 概要

従来のワードクラウドとの違い

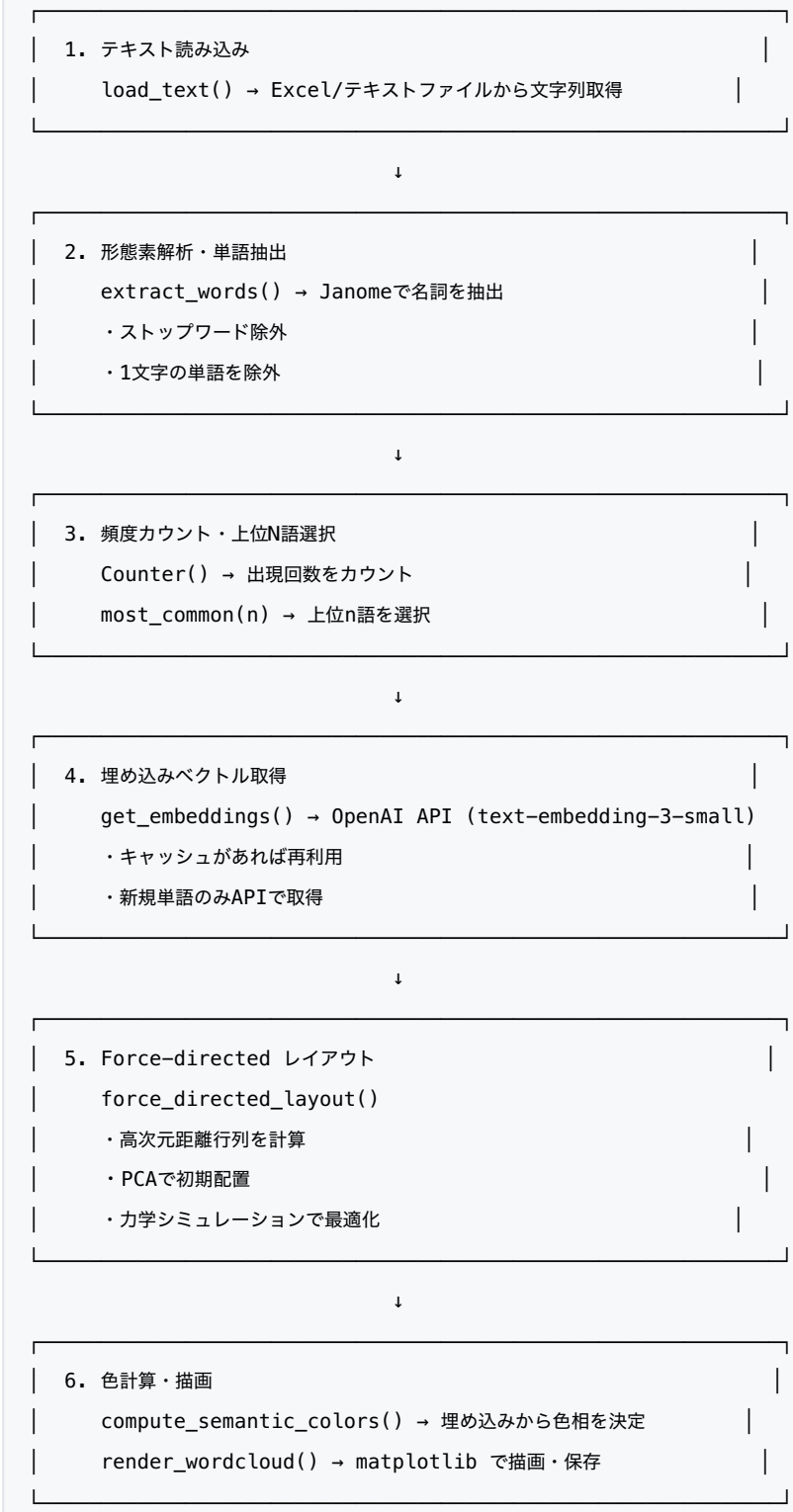
項目	従来のワードクラウド	意味的ワードクラウド
配置基準	ランダム or スパイラル	意味的類似度
単語間の関係	考慮しない	類似した単語が近くに配置
技術	単純な衝突回避	埋め込みベクトル + Force-directed layout

このプログラムが行うこと

```
入力テキスト → 形態素解析 → 頻出単語抽出 → 埋め込みベクトル取得
→ 意味的距離計算 → Force-directed配置 → 画像出力
```

---

2. 処理フロー


1. テキスト読み込み
load_text() → Excel/テキストファイルから文字列取得
2. 形態素解析・単語抽出
extract_words() → Janomeで名詞を抽出
・ストップワード除外
・1文字の単語を除外
3. 頻度カウント・上位N語選択
Counter() → 出現回数をカウント
most_common(n) → 上位n語を選択
4. 埋め込みベクトル取得
get_embeddings() → OpenAI API (text-embedding-3-small)
・キャッシュがあれば再利用
・新規単語のみAPIで取得
5. Force-directed レイアウト
force_directed_layout()
・高次元距離行列を計算
・PCAで初期配置
・力学シミュレーションで最適化
6. 色計算・描画
compute_semantic_colors() → 埋め込みから色相を決定
render_wordcloud() → matplotlib で描画・保存

---

3. キーコンセプト

3.1 埋め込みベクトル (Embedding Vector)

単語を高次元の数値ベクトルに変換したもの

```
「機械学習」 → [0.012, -0.034, 0.056, ..., 0.023]  (1536次元)
「深層学習」 → [0.015, -0.031, 0.058, ..., 0.019]  (1536次元)
「料理」     → [-0.045, 0.067, -0.012, ..., 0.089] (1536次元)
```

- 意味が似ている単語 → ベクトルも似ている（近い）
- 意味が異なる単語 → ベクトルも異なる（遠い）

OpenAI の `text-embedding-3-small` モデルは1536次元のベクトルを出力します。

3.2 コサイン類似度 (Cosine Similarity)

2つのベクトルの「向き」の類似度を測る指標

```
              A · B
cos(θ) = ──────────────
           |A| × |B|

値の範囲: -1 ～ 1
  1: 完全に同じ方向（非常に類似）
  0: 直交（関連なし）
 -1: 正反対の方向（対義的）
```

実際の単語埋め込みでは、ほとんどの値が 0.3〜0.9 の範囲に収まります。

```
# コード内での計算
sim_matrix = cosine_similarity(embeddings)  # 類似度行列
dist_matrix = 1 - sim_matrix                # 距離行列に変換
```

3.3 PCA (主成分分析)

高次元データを低次元に圧縮する手法

```
1536次元 → 2次元

目的: 初期配置の決定
方法: データの分散が最大になる方向を見つけて射影
```

```
pca = PCA(n_components=2)
initial_coords = pca.fit_transform(embeddings)
```

PCAは「できるだけ情報を保持しながら次元を減らす」ため、 意味的に近い単語は2次元でも近くに配置されます。

3.4 初期配置: PCA vs t-SNE

本プログラムではPCAを初期配置に使用していますが、t-SNEも選択肢の一つです。

比較表

項目	PCA	t-SNE
計算速度	**高速**	遅い（反復計算）
決定性	**決定的**（毎回同じ結果）	確率的（毎回異なる）
大域構造	**保持される**	失われやすい
局所構造	部分的に保持	**よく保持される**
クラスタ分離	緩やか	**明確に分離**

PCAを選んだ理由

1. **Force-directedとの相性**
   - PCAは大域的な構造を保持 → Force-directedで微調整
   - t-SNEは既に局所最適化済み → Force-directedと役割が重複
2. **決定性**
   - PCAは同じ入力に対して常に同じ出力
   - ランダムシードの管理が単純
3. **計算効率**
   - PCAは線形変換で高速
   - t-SNEは$O(N^2)$〜$O(N \log N)$の反復計算

t-SNEを使う場合のメリット・デメリット

メリット: - クラスタがより明確に分離される - 局所的な類似関係がよく表現される

デメリット: - 計算時間が長い - 大域的な距離関係が歪む（遠い点同士の距離が信頼できない） - Force-directedレイアウトの効果が薄れる（t-SNE自体が配置最適化） - perplexityパラメータの調整が必要

コード例（t-SNEを使う場合）

```
from sklearn.manifold import TSNE

# PCAの代わりにt-SNEを使用
# tsne = TSNE(n_components=2, perplexity=30, random_state=42)
# initial_coords = tsne.fit_transform(embeddings)
```

PCAの限界と、それでも問題にならない理由（重要）

PCAの本質的な限界:

1. **線形手法である**
   - 1536次元の埋め込み空間は非線形構造を持つ
   - PCAは線形射影なので、非線形な類似関係を捉えきれない
2. **最大分散 ≠ 意味的類似度**
   - PCAは「分散が最大になる方向」を探す
   - これが「意味的に近い単語を近くに配置する」と一致するとは限らない

しかし、この設計では問題にならない:

```
┌─────────────────────────────────────────────┐
│  PCAの役割 = 「だいたいの初期配置」を提供するだけ    │
│                      ↓                      │
│  Force-directed = 高次元距離を使って本格的に最適化   │
└─────────────────────────────────────────────┘
```

核心となるコード:

```
# Force-directedは高次元の距離を直接使う
ideal_distances = cosine_distance(embeddings)  # 1536次元から計算

# PCAの出力は使わない！ただの初期位置として利用
spring_force = attraction_strength * (current_dist - ideal_dist)
```

つまり： - PCAの出力 → **初期位置だけ**（500回の反復で書き換わる） - 最終配置 → **高次元コサイン距離**が決める

検証: 初期配置を完全ランダムにしても、十分な反復で同様の結果が得られる：

```
# 極端な例：完全ランダム初期配置
word.x = center_x + random.uniform(-100, 100)
word.y = center_y + random.uniform(-100, 100)
# → 反復を増やせば収束する（ただし収束が遅い）
```

まとめ:

観点	評価
PCAは最適か？	いいえ、非線形構造には不向き
問題になるか？	いいえ、Force-directedが本質的な最適化を担う
改善の余地	ある（UMAPなど）が、実用上は十分

結論: 本プログラムでは、Force-directedレイアウトが主役であり、 初期配置は「おおまかな配置」を提供すれば十分なため、高速で決定的なPCAを採用しています。 PCAの限界は、Force-directedの反復最適化によって補われます。

初期配置: PCA vs MDS の検討

MDS（多次元尺度構成法）も初期配置の候補として検討しました。

MDSの原理的な優位性:

Force-directedレイアウトでは、高次元での距離行列（`dist_matrix = 1 - cosine_similarity`）を 制約として使用しています。MDSはまさにこの距離行列を入力として2Dに射影するため、 原理的にはPCAよりマッチしている可能性がありました。

```
# PCA: 埋め込みベクトルを直接射影
pca = PCA(n_components=2)
initial_coords = pca.fit_transform(embeddings)

# MDS: 距離行列を保存するように射影
mds = MDS(n_components=2, dissimilarity='precomputed')
initial_coords = mds.fit_transform(dist_matrix)
```

実際の検証結果:

指標	PCA	MDS
初期重なり数	多い	**少ない**
最終的な配置密度	**密**	疎（隙間が多い）
色のグラデーション	**滑らか**	局所的に不連続

MDSは初期重なりが約半分と少なく、距離保存の観点では優れていました。 しかし、視覚的な結果ではPCAの方が好ましい結果となりました。

色空間の次元数と、PCAが優れていた理由:

本プログラムでは、意味的類似度を色で表現しています。

```
高次元埋め込み（1536次元）
    ↓
PCA 2D射影
    ↓
角度（1次元）→ 色相
距離（1次元）→ 彩度
```

ここで重要なのは、**1536次元の意味構造を実質1〜2次元（主に色相）で表現している** という点です。この次元の制約が、PCAとMDSの結果の違いに大きく影響しています。

PCAの特性と色との相性:

PCAは「最も分散が大きい方向」を第1主成分として選びます。 これは「最も情報量が多い軸」であり、限られた次元で情報を圧縮する際に合理的な選択です。

・第1・第2主成分は、データの主要な変動方向を捉える

- この軸に沿った角度が色相になるため、意味的な違いが色の違いとして表現されやすい
- 結果として、色のグラデーションが滑らかになる

MDSの特性と色との相性の悪さ:

MDSは「すべての点間距離を保存する」ことを目的とします。

- 高次元の複雑な距離関係を2Dで完全に再現することはできない
- MDSの2D座標における**角度には特別な意味がない**（座標系の回転は任意）
- 色相は角度で決まるため、MDSの座標と色相の間に整合性がない

具体的には、MDSで隣り合う単語でも、PCA空間では異なる角度にある場合があり、 これが「大局的には色が適切だが、局所的に不連続」という現象を引き起こします。

MDSの距離保存による副作用:

MDSは距離を忠実に保存しようとするため、別の問題も生じます。

- 高次元では互いに等距離な点の集合があり得るが、2Dでは完全に再現できない
- 距離保存を優先した結果、配置が広がり隙間が生じやすい

コード例（MDSを使う場合）:

```
# --layout-method mds オプションで切り替え可能
from sklearn.manifold import MDS

mds = MDS(n_components=2, dissimilarity='precomputed', random_state=seed)
initial_coords = mds.fit_transform(dist_matrix)
```

結論:

MDSは距離保存という点で原理的にはForce-directedと相性が良いように思えます。 しかし、色が実質1次元（色相）という制約の下では、「どの軸を選ぶか」が重要になります。

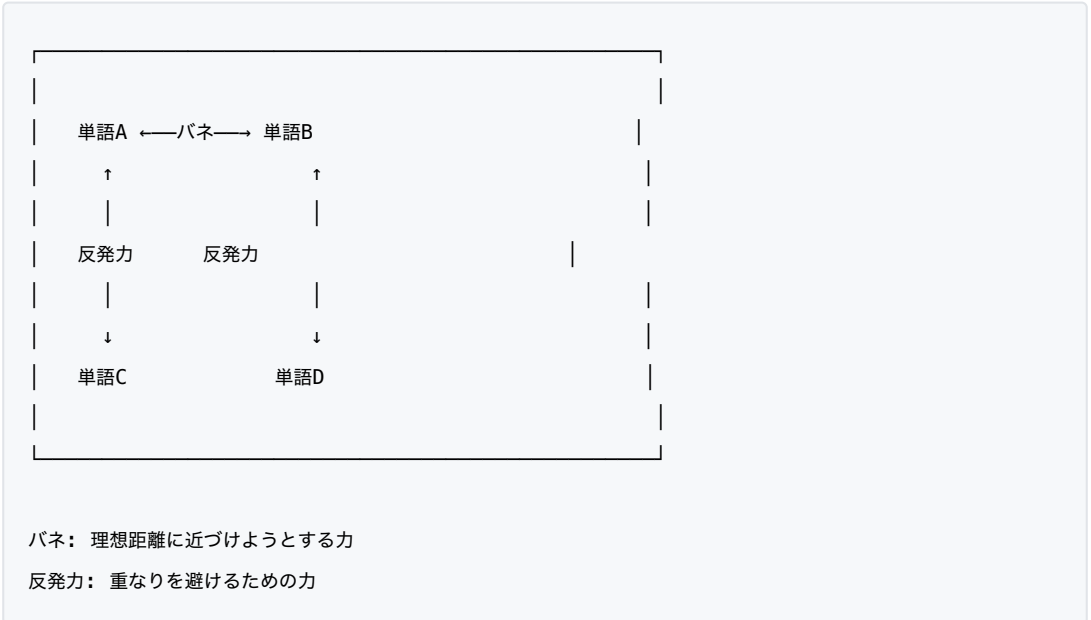
- PCAは「最も重要な軸」を選ぶ設計なので、限られた色空間を有効に使える
- MDSは距離保存が目的で、軸の選び方に意図がないため、色との相性が悪い

本プログラムではPCAをデフォルトとし、MDSはオプション（`--layout-method mds`）として 提供しています。

3.5 Force-directed Layout (力指向レイアウト)

物理シミュレーションによるグラフ配置アルゴリズム

各単語をノード、意味的関係をエッジとみなし、 バネと反発力でノードを動かして最適な配置を見つけます。



---

4. 各処理の詳細

4.1 ストップワードの読み込み

```
def load_stopwords(custom_filepath: str = "stopwords.txt") -> set:
```

3つのソースからストップワードを収集:

1. **stopwords-iso 日本語** (~134語)
   - 「これ」「それ」「ある」「いる」など
2. **stopwords-iso 英語** (~1298語)
   - “the”, “and”, “is”, “of” など
3. **カスタムファイル** (stopwords.txt)
   - ユーザー定義の除外単語

キャッシュ機構: - 初回: URLからダウンロード → `stopwords_cache.json` に保存 - 2回目以降: キャッシュから読み込み（高速）

4.2 形態素解析

```
def extract_words(text: str) -> list[str]:
    tokenizer = Tokenizer()  # Janome
    for token in tokenizer.tokenize(text):
        pos = token.part_of_speech.split(',')
        if pos[0] == '名詞' and pos[1] not in ['非自立', '代名詞', '数']:
            # 名詞かつ、非自立・代名詞・数詞でない
            ...
```

Janomeの出力例:

```
「機械学習の研究」
  ↓
機械学習  /  名詞,固有名詞,一般,*
の        /  助詞,連体化,*,*
研究      /  名詞,サ変接続,*,*
```

フィルタリング条件: - 品詞が「名詞」である - 名詞の中でも「非自立」「代名詞」「数」は除外 - 1文字の単語は除外 - ストップワードに含まれていない

4.3 埋め込み取得とキャッシュ

```
def get_embeddings(words: list[str], api_key: str = None,
                   cache_file: str = "embeddings_cache.json") -> np.ndarray:
```

キャッシュ構造:

```
{
  "機械学習": [0.012, -0.034, ...],
  "深層学習": [0.015, -0.031, ...],
  "ロボット": [0.023, 0.045, ...]
}
```

処理フロー:

1. キャッシュファイルを読み込み
2. 要求された単語のうち、キャッシュにある/ないを分離
3. キャッシュにない単語だけAPIで取得
4. 新しい単語をキャッシュに追加・保存
5. 要求された順序で埋め込みを返す

4.4 Force-directed Layout の詳細

```
def force_directed_layout(words, canvas_width, canvas_height, iterations, verbose):
```

Step 1: 異方性スケールの計算

```
aspect_ratio = canvas_width / canvas_height  # 例: 1200/900 = 1.333
scale_x = np.sqrt(aspect_ratio)              # 例: 1.155
scale_y = 1.0 / np.sqrt(aspect_ratio)        # 例: 0.866
```

これにより、正方形ではなく楕円形のレイアウトになります。

Step 2: 距離行列の計算

```
sim_matrix = cosine_similarity(embeddings)  # N×N の類似度行列
dist_matrix = 1 - sim_matrix                 # 距離に変換 (0〜2)

# 理想距離にスケーリング
ideal_distances = (dist_matrix / max_dist) * ideal_scale
```

例:

```
        機械学習  深層学習  料理
機械学習        0.0       0.15     0.85
深層学習        0.15      0.0      0.82
料理            0.85      0.82     0.0

→ 機械学習と深層学習は近く、料理は遠く配置される
```

Step 3: PCAによる初期配置

```
pca = PCA(n_components=2)
initial_coords = pca.fit_transform(embeddings)

# 異方性スケールを適用
word.x = center_x + initial_coords[i, 0] * scale * scale_x
word.y = center_y + initial_coords[i, 1] * scale * scale_y
```

Step 4: 力の計算と更新

3種類の力と計算次元（重要）:

このアルゴリズムの核心は、**高次元の意味的距離を2次元配置で再現する**ことです。 各力がどの次元で計算されるかを理解することが重要です。

力	理想/目標	現在/実際	目的
スプリング力	**1536次元**（コサイン距離）	2次元（キャンバス）	意味的関係の再現
反発力	-	2次元のみ	視覚的重なり防止
中心引力	-	2次元のみ	キャンバス内収束

1. **スプリング力 (attraction)** ← 高次元→2次元マッピングの核心

```
# 理想距離: 1536次元空間でのコサイン距離から算出
ideal_dist = ideal_distances[i, j]  # 高次元由来

# 現在距離: 2次元キャンバス上の距離
current_dist = np.sqrt(dx**2 + dy**2)  # 2次元

spring_force = attraction_strength * (current_dist - ideal_dist)
```

- 現在距離 > 理想距離 → 引き寄せる
- 現在距離 < 理想距離 → 押し離す
- **意味的に近い単語は2Dでも近くに配置される**

2. **反発力 (repulsion)** ← 純粋に2次元の視覚的問題

```
# すべて2次元で計算
min_sep = (words[i].width + words[j].width) / 2 + 5  # 単語幅（2D）
actual_dist = np.sqrt(dx**2 + dy**2)  # 2D距離

if actual_dist < min_sep:
    repulsion = repulsion_strength / (actual_dist² + 1)
```

- 単語が重なりそうなとき発生
- 距離が近いほど強い
- **意味的距離とは無関係、視覚的な配置のみ考慮**

3. **中心への引力 (center force)** ← 2次元

```
center_force = 0.002 * dist_to_center
```

- 単語が散らばりすぎないように中心に引き寄せる

シミュレーションループ:

```
for iteration in range(iterations):  # デフォルト500回
    forces = compute_forces()
    temperature = max(0.1, 1.0 - iteration / iterations)  # 冷却

    for word in words:
        word.vx = (word.vx + force) * damping * temperature
        word.vy = (word.vy + force) * damping * temperature
        word.x += word.vx
        word.y += word.vy
```

温度 (temperature): - 初期: 1.0 → 大きく動く - 最終: 0.1 → 微調整のみ - 徐々に冷却することで安定した配置に収束

4.5 色の計算

色の計算は、意味的に近い単語に似た色を付けるための処理です。

目的

```
意味的に近い単語 → 似た色
意味的に遠い単語 → 異なる色
```

これにより、ワードクラウドを見たときに「このグループは似た話題」と直感的にわかります。

処理の流れ

```
┌─────────────────────────────────────────────────────────┐
│ Step 1: 埋め込みベクトルをPCAで2次元に圧縮                    │
│         1536次元 → 2次元（x, y座標）                       │
└─────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────┐
│ Step 2: 各単語の(x, y)から「角度」を計算                      │
│         angle = atan2(y, x)  → −π 〜 +π の値              │
└─────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────┐
│ Step 3: 角度を色相(Hue)に変換                               │
│         hue = (angle + π) / (2π)  → 0.0 〜 1.0 の値       │
└─────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────┐
│ Step 4: HSV色空間からRGBに変換して描画                       │
└─────────────────────────────────────────────────────────┘
```

Step 1: PCAで2次元に圧縮

```
pca = PCA(n_components=2)
coords_2d = pca.fit_transform(embeddings)
```

1536次元の埋め込みを2次元に圧縮します。 意味的に近い単語は、2次元でも近い位置に配置されます。

```
例:
  「機械学習」→ (0.5, 0.3)
  「深層学習」→ (0.6, 0.4)   ← 近い！
  「料理」    → (−0.8, −0.2) ← 遠い
```

Step 2: 角度の計算

```
angle = np.arctan2(y, x)
```

`atan2(y, x)` は、原点から点(x, y)への角度を返します。

```
                  90° (π/2)
                    ↑
                    │
                    │ (0.5, 0.3) ← この点の角度は約31°
                    │   ╱
                    │ ╱
   180° (π) ←──────┼──────→ 0° (0)
                    │
                    │
                    │
                    ↓
                 −90° (−π/2)

戻り値の範囲: −π ～ +π （約 −3.14 ～ +3.14）
```

Step 3: 角度を色相に変換

```
hue = (angle + np.pi) / (2 * np.pi)
```

角度（-π ～ +π）を色相（0 ～ 1）に変換します。

```
変換式の意味:
  angle = −π  →  hue = (−π + π) / (2π) = 0.0  (赤)
  angle = 0   →  hue = (0 + π) / (2π)  = 0.5  (シアン)
  angle = +π  →  hue = (+π + π) / (2π) = 1.0  (赤)
```

色相環 (Hue Circle)

```
             hue=0 (赤)
                ↑
                │
   hue=0.83 ←───┼───→ hue=0.17
     (紫)       │       (黄)
                │
   hue=0.67 ←───┼───→ hue=0.33
    (青)        │      (緑)
                │
                ↓
          hue=0.5 (シアン)
```

Step 4: HSVからRGBへの変換

```
saturation = 0.65 + 0.3 * min(dist / max_dist, 1.0)  # 彩度
value = 0.85                                          # 明度
rgb = colorsys.hsv_to_rgb(hue, saturation, value)
```

HSV色空間: - **H (Hue/色相)**: 色の種類（0〜1で赤→黄→緑→シアン→青→紫→赤） - **S (Saturation/彩度)**: 色の鮮やかさ（0=灰色、1=鮮やか） - **V (Value/明度)**: 明るさ（0=黒、1=明るい）

彩度の調整:

```
saturation = 0.65 + 0.3 * min(dist / max_dist, 1.0)
```

- 原点から遠い単語（dist大）→ 彩度が高い（鮮やか）
- 原点に近い単語（dist小）→ 彩度が低い（くすんだ色）

これにより、中心にある「一般的な単語」は控えめな色、 外側にある「特徴的な単語」は鮮やかな色になります。

実例

```
PCA座標と色の対応:

 単語          (x, y)       角度    色相   色
 ──────────────────────────────────────────────
 機械学習    (0.5, 0.3)    31°      0.59    シアン系
 深層学習    (0.6, 0.4)    34°      0.59    シアン系（似た色）
 ロボット    (0.7, -0.2)   -16°     0.46    緑系
 細胞         (-0.3, 0.6)  117°      0.82     紫系
 料理         (-0.8, -0.2) -166°     0.04     赤系
```

なぜPCAを使うのか（配置と別に計算する理由）

色の計算では、**配置用のPCAとは別に**、色専用のPCAを実行しています。

```python
def compute_semantic_colors(words: list[Word]) -> list[tuple]:
    # 新たにPCAを実行（配置とは独立）
    embeddings = np.array([w.embedding for w in words])
    pca = PCA(n_components=2)
    coords_2d = pca.fit_transform(embeddings)
```

理由: - 配置は異方性スケールやForce-directedで変形されている - 色は純粋な「意味的な位置関係」を反映させたい - PCAの第1・第2主成分が意味的なクラスタを捉える

---

5. パラメータの影響

コマンドラインオプション

オプション	デフォルト	説明
`-n, --num-words`	80	表示する単語数
`--cache-words`	200	キャッシュする単語数
`--iterations`	500	シミュレーション反復回数
`--seed`	None	ランダムシード
`--layout-method`	pca	初期配置の方法（pca/mds）
`--pptx-scale`	1.0	PPTXスライドのスケール（例: 0.5で半分のサイズ）
`-o, --output`	WordCloud_YYYYMMDD.png	出力ファイル名

内部パラメータ

```
# force_directed_layout 内
attraction_strength = 0.01   # スプリングの強さ
repulsion_strength = 600     # 反発力の強さ
damping = 0.9                # 減衰係数

# 重なり解消パラメータ
padding = 1                  # 重なり検出のパディング
nudge = 4.0                  # 重なり解消時の移動量
overlap_iterations = 300     # 重なり解消の反復回数

# 回転制限
# 4文字以上の単語は回転させない（長い単語の回転はレイアウトを崩しやすい）

# キャンバス
canvas_width = 1200
canvas_height = 900
FIGSIZE_DIVISOR = 120        # figsize = (1200/120, 900/120) = (10", 7.5")

# フォントサイズ
font_size = 8 + 19 * (ratio ** 0.5)  # 8〜27の範囲
```

パラメータ調整のヒント

症状	調整
単語が散らばりすぎ	`attraction_strength` を上げる
単語が密集しすぎ	`repulsion_strength` を上げる
配置が安定しない	`iterations` を増やす
重なりが多い	`repulsion_strength`、`padding`、`nudge` を上げる
隙間が多すぎる	`repulsion_strength`、`padding`、`nudge` を下げる
長い単語の回転で崩れる	回転制限の文字数を調整

---

6. 数学的背景

6.1 コサイン類似度の幾何学的意味

2つのベクトル A, B のコサイン類似度:

```
cos(θ) = (A · B) / (|A| × |B|)

ここで:
  A · B = Σ(ai × bi)  (内積)
  |A| = √Σ(ai²)       (ノルム)
```

なぜコサイン類似度を使うのか: - ベクトルの「大きさ」ではなく「向き」を比較 - 単語の出現頻度に依存しない - 値が -1〜1 に正規化される

6.2 PCAの数学

目標: 分散を最大化する方向を見つける

```
1. データの平均を引く（中心化）
2. 共分散行列を計算: C = (1/n) × X^T × X
3. 固有値分解: C × v = λ × v
4. 最大固有値に対応する固有ベクトルが第1主成分
```

1536次元 → 2次元の場合、上位2つの固有ベクトルを使用。

6.3 Force-directed Layout の物理モデル

フックの法則 (バネ):

```
F = −k × (x − x₀)

k: バネ定数（attraction_strength）
x: 現在の距離
x₀: 理想距離（ideal_distance）
```

クーロンの法則 (反発):

```
F = k × q₁ × q₂ / r²

簡略化: F = repulsion_strength / (r² + 1)
```

運動方程式:

```
v(t+1) = (v(t) + F) × damping × temperature
x(t+1) = x(t) + v(t+1)
```

6.4 異方性スケールの意味

```
aspect_ratio = width / height = 1200 / 900 ≈ 1.333

scale_x = √(aspect_ratio) ≈ 1.155
scale_y = 1 / √(aspect_ratio) ≈ 0.866

scale_x × scale_y = 1（面積保存）
```

これにより: - x方向に1.134倍伸ばす - y方向に0.882倍縮める - 全体の「面積」は変わらない - 結果として横長の楕円形になる

付録: コード構造

```
semantic_wordcloud.py
│
├── 定数・設定
│   ├── FONT_PATHS
│   ├── JAPANESE_STOPWORDS_URL
│   ├── ENGLISH_STOPWORDS_URL
│   └── STOPWORDS_CACHE_FILE
│
├── クラス
│   └── Word（単語の状態を保持）
│       ├── text, freq, font_size, embedding
│       ├── x, y, vx, vy, rotation
│       ├── width, height（プロパティ）
│       ├── get_bbox()
│       └── overlaps()
│
├── ユーティリティ関数
│   ├── load_stopwords()
│   ├── get_font_path()
│   ├── extract_words()
│   ├── load_text_from_excel()
│   └── load_text()
│
├── コア処理
│   ├── get_embeddings()
│   ├── force_directed_layout()
│   │   ├── compute_forces()
│   │   └── count_overlaps()
│   ├── compute_semantic_colors()
│   └── render_wordcloud()
│
└── main()
    └── 引数解析・処理フロー制御
```

参考資料

- OpenAI Embeddings
- scikit-learn: PCA
- Force-directed graph drawing (Wikipedia)
- Euler method (Wikipedia) — 本実装の数値積分法

- PageRank (Wikipedia) — 反復的相互依存解決の代表例
- Perron-Frobenius theorem (Wikipedia) — PageRank収束の数学的保証
- stopwords-iso
- Janome

---

7. 関連研究と本実装の位置づけ

7.1 既存の研究・実装

意味的ワードクラウドは新しいアイデアではなく、いくつかの先行研究・実装が存在します。

学術研究

研究	手法
Semantic word cloud generation based on word embeddings (2016)	Word2Vecで意味的距離を計算、グラフベースのレイアウト
Semantic Word Clouds with t-SNE	t-SNEで意味的配置を生成
ReCloud	文法的依存関係からセマンティックグラフを構築、Force-directedレイアウト

既存ツール・実装

プロジェクト	特徴
Arizona大学 Semantic Word Cloud	Webツール、複数アルゴリズム対応
nlp-chula/swordcloud	Python、タイ語NLP研究グループ
ttavni/SemanticWordClouds	PKE + 事前学習埋め込み
WordCloud.jl	Julia、t-SNE対応

7.2 本実装の特徴

既存手法との共通点

- 埋め込みベクトルによる意味的距離の計算
- Force-directedレイアウト

・PCAによる初期配置・色付け

本実装の独自の組み合わせ

要素	本実装	一般的な既存実装
埋め込み	OpenAI text-embedding-3-small	Word2Vec, GloVe, FastText
ベクトル次元	1536次元	100-300次元
距離制約	**高次元コサイン距離を直接使用**	2D投影後の距離
レイアウト形状	異方性（楕円形）対応	多くは正方形のみ
言語対応	日本語（Janome + stopwords-iso）	英語中心が多い

技術的な差別化ポイント

1. **OpenAI Embeddings の利用**
   - 最新の大規模言語モデルベースの埋め込み
   - 1536次元の高密度な意味表現
   - 既存実装の多くはWord2Vec/GloVe（100-300次元）
2. **高次元距離の直接利用**
   - 多くの実装: 先にt-SNE/PCAで2Dに落としてから配置
   - 本実装: 高次元のコサイン距離を理想距離として保持し、Force-directedで2D再現
   - これにより、次元削減による情報損失を最小化
3. **異方性Force-directed**
   - 横長・縦長など任意のアスペクト比に対応
   - 多くの実装は正方形レイアウトのみ
4. **日本語ネイティブ対応**
   - Janomeによる形態素解析
   - stopwords-isoの日本語ストップワード
   - 英語中心の既存ツールとの差別化

7.3 結論

個々の技術（埋め込み、Force-directed、PCA）は既存のものですが、「OpenAI Embeddings + 高次元距離制約 + 異方性レイアウト + 日本語対応」 という組み合わせは、調査した範囲では前例が見つかりませんでした。

---

8. アルゴリズムの数理的考察

本セクションでは、PCA初期配置と力学シミュレーションの数理的性質を掘り下げ、 両者の相補的な役割を考察します。

8.1 埋め込みベクトルの統計的性質

ベクトルの基本特性

OpenAI `text-embedding-3-small` が返す1536次元ベクトルの性質を実データで確認しました。

```
単語数: 251, 次元数: 1536

各要素の値域: 約 −0.11 〜 +0.12（正負両方を含む）
負の要素の割合: 約50%（1536次元中769次元が負）
L2ノルム: 全ベクトルが正確に 1.000000
```

重要な点は、**全ベクトルがL2正規化されている**ことです。 すなわち、全ての単語ベクトルは1536次元空間の**単位超球面の表面**に拘束されています。

各次元の分布

各次元の周辺分布をShapiro-Wilk検定で調べた結果:

```
ランダムに選んだ100次元のうち91次元が正規分布と見なせる（p > 0.05）
```

個々の次元の周辺分布は概ね正規分布的です。しかし、多変量としては「全次元の二乗和 = 1」という超球面制約があるため、真の多変量正規分布ではありません。

PCA適用の妥当性

PCAは正規分布を前提としない手法（分散最大化の線形変換）であるため、 超球面上のデータに対しても適用可能です。さらに、L2正規化済みデータでは コサイン類似度とユークリッド距離の間に以下の関係が成り立ちます:

```
cos_sim = 1 − ||a − b||² / 2
```

実データでの相関係数は **-0.997** であり、PCA（ユークリッドベース）でも コサイン類似度に基づく意味的近さの関係がよく保存されることを確認しました。

8.2 PCA射影の分布特性と周辺部の隙間

PCA配置で周辺部に隙間が生じる理由

PCAだけの初期配置でも意味的な配置としてはそこそこ良好ですが、 **周辺部に隙間が多い**という視覚的な課題があります。 これはPCAの原理に直接起因する構造的な性質です。

原因1: 中心集中

PCAの各主成分スコアは、1536次元の値の線形結合（重み付き合計）です。 中心極限定理により、このような合計値は正規分布に近づきます。 正規分布は中心付近に密度が集中するため:

```
面積25%の中心領域（半径0.5以内）に、72%の単語が集中
周辺リング（0.75〜1.0）には26語のみ（密度は中心の約1/20）
```

原因2: 四隅の空白

PCAの2軸は無相関（直交）であるため、2次元の分布は楕円〜円形になります。 矩形キャンバスの四隅（|x| > 0.7 かつ |y| > 0.7 の領域）には**点が0個**です。 これはPCAの構造的な性質であり、データが増えても四隅は常にスカスカです。

原因3: 外れ値の影響

PC2の尖度（kurtosis）は7.62と、正規分布（3.0）よりかなり大きい値でした。 これは「極端に中心に集中し、少数の外れ値が遠方にある」分布を意味します。 外れ値がキャンバスの端に飛ぶことで、中間領域に大きな隙間が生じます。

まとめ: PCAの構造的制約

PCAの性質	ワードクラウドへの影響
主成分スコアが正規分布的	中心に密集、周辺がスカスカ
2軸が無相関（直交）	楕円分布 → 四隅が空白
分散最大化が目的	外れ値が遠方に飛び、隙間を作る

PCAは「意味的距離を保つ」目的には合っていますが、 「キャンバスを均一に埋める」こととは本質的に相反します。

8.3 PCAと力学シミュレーションの相補性

前節の分析から、PCAと力学シミュレーションの役割分担が明確になります。

ステップ	役割	得意なこと	苦手なこと
PCA	意味的な相対位置を決める	似た単語を近くに配置	空間を均一に使う
力学シミュレーション	空間を効率的に使う	隙間を埋める・重なり解消	意味的関係の発見

PCAが**トポロジー（位相的構造）**を与え、 力学シミュレーションが**レイアウト最適化**を担います。

PCAだけでもそこそこ良い配置になるのは、意味的構造という最も重要な部分を PCAが正しく捉えているからです。力学シミュレーションはそれを壊さずに、 「中心に72%集中」「四隅が空白」という問題を、反発力によって自然に解消します。

8.4 力学シミュレーションの数値計算

同期更新方式

本実装では、各反復（iteration）で**全単語を同時に更新**します。

```
1ターンの処理:
  1. 全ペア (N×(N−1)/2) の力を計算 → forces[N, 2] に蓄積
  2. 全単語の速度・位置を一斉に更新
  3. 境界チェック
```

これは分子動力学シミュレーションと同じ**同期更新（synchronous update）**方式です。

力の方向と大きさの決定

各単語には3種類の力がベクトル的に合算され、 その**合力の方向**が移動方向、**合力の大きさ**が移動量の基礎になります。

1. **バネ力**: 2単語を結ぶ線上の方向。強さは「現在の2D距離と理想距離の差」に比例
2. **近接反発力**: 相手から離れる方向。逆二乗則で近いほど急激に強くなる
3. **中心引力**: キャンバス中心に向かう方向。中心から遠いほど強い

移動距離は力を直接位置に適用するのではなく、**速度（慣性）**を経由して決まります:

```
v(t+1) = (v(t) + F(t)) * damping * temperature
x(t+1) = x(t) + v(t+1)
```

オイラー法の誤差と収束

この更新式は**オイラー法（前進差分）**と呼ばれる最も単純な数値積分法です。

力 F(t) は時刻 t の全単語の配置に基づいて計算されますが、 実際にはその力で全単語が同時に動くため、 **1ステップ後の配置はどの単語にとっても想定通りにはなりません。**

```
時刻 t:    A---[100]---B---[100]---C   ← この配置で力を計算
時刻 t+1:  A'--[??]---B'---[??]---C'   ← 全員が動いた結果、距離は想定と異なる
```

これは数値シミュレーション全般に共通する根本的な問題ですが、 本実装では3重の安定化機構により収束を保証しています:

機構	パラメータ	効果
速度減衰	damping = 0.9	毎ステップ速度を10%減らし振動を抑制
温度低下	temperature: 1.0 → 0.1	終盤は移動量が1/10になり微調整モード
速度制限	max_speed = 10 × temperature	1ステップの移動量に上限を設定

「1ステップでは不正確でも、小刻みに何度も修正すれば徐々に収束する」という戦略であり、特に温度低下の効果が大きく、 終盤はほぼ動かなくなるため不正確さの影響も十分小さくなります。

なお、より高精度な数値積分法（Verlet積分、Runge-Kutta法など）や、1単語ずつ逐次更新する方式（Gauss-Seidel法）も存在しますが、 本用途では厳密な物理精度は不要であるため、 単純なオイラー法で実用上十分な結果が得られています。

8.5 解析解と数値解: 何が解けて何が解けないか

問い: 反復せずに直接解けるか？

力学シミュレーションの収束先（全力の釣り合い状態）を、 反復を使わず解析的に求めることは可能でしょうか。

線形化すればPCA / Classical MDS に帰着する

非線形性の源を段階的に除去してみます:

Step 1: 近接反発力と中心引力を除去 — バネ力のみにする

全ペアの理想距離にできるだけ近づける問題になります:

```
minimize Σ_{i,j} (||xi − xj|| − d_ij)²
```

これは**MDS（多次元尺度構成法）**そのものですが、 ||xi - xj|| の中の平方根がまだ非線形です。

Step 2: 距離の二乗で妥協する

```
minimize Σ_{i,j} (||xi − xj||² − d_ij²)²
```

平方根が消え、**Classical MDS** となります。 これは距離行列の固有値分解で一発で解けます。

Step 3: 実はPCAと等価

L2正規化されたデータでは、Classical MDSとPCAは数学的に等価です:

手法	入力	解法	保存するもの
PCA	共分散行列	固有値分解	分散（内積の構造）
Classical MDS	距離行列	固有値分解	ペア間距離

つまり、**解析的に解けるところまで単純化すると、 現在のPCA初期配置に帰着します。**

重なり防止を入れると解析解は困難

ワードクラウドとしての実用性に不可欠な重なり防止は、本質的に以下の制約です:

```
全ペア(i, j)について: 単語iと単語jの矩形領域が重ならない
```

この制約は: - **不等式制約**である（等式ではない） - 単語ごとにサイズが異なる - 回転する単語もある - 矩形の重なり判定は非線形

これを含めた最適化は不等式制約付き非線形最適化問題となり、 解析解は知られていません。

設計の合理性

```
解析的に解ける範囲（線形）
├── 意味的な理想配置 → PCA / Classical MDS（固有値分解）
│     └── 本実装の初期配置がまさにこれ
│
解析的に解けない範囲（非線形）
├── 重なり防止 → 不等式制約
├── キャンバス境界 → 不等式制約
└── 異なるサイズの矩形パッキング → NP困難に近い
      └── 力学シミュレーションで近似的に解く
```

本実装の2段階アプローチ（PCAで解析解 → 力学シミュレーションで非線形部分を処理）は、 **解ける部分は解析的に解き、解けない部分だけ反復に頼る** という意味で、計算論的に合理的な設計です。

8.6 反復的相互依存の解決: PageRankとの類似性

相互依存問題の普遍性

力学シミュレーションの「全単語の位置が全単語の位置に依存する」という構造は、 他の分野にも共通する問題パターンです。

代表的な例が**論文の被引用評価**です。 単純に引用件数を数えるのではなく、 「多く引用されている（重要な）論文からの引用をより高く評価する」 という手法があります（Google PageRank と本質的に同じアルゴリズム）。

```
論文Aの重要度 ← 論文Bの重要度に依存（BがAを引用）
論文Bの重要度 ← 論文Cの重要度に依存（CがBを引用）
論文Cの重要度 ← 論文Aの重要度に依存（AがCを引用）
→ 循環！ 定義が循環していて決定できないように見える
```

解決法: 反復による固定点の発見

一見決定できないように見えるこの問題は、力学シミュレーションと同じ反復法で解けます:

```
1. 全論文の重要度を適当に初期化（例: 全部1.0）
2. 各論文の重要度を「引用元の現在の重要度」で再計算
3. 値が変わらなくなるまで 2 を繰り返す
```

	力学シミュレーション	**PageRank/引用評価**
求めたいもの	全単語の位置	全論文の重要度
循環依存	Aの位置はBの位置に依存し、BもAに依存	Aの重要度はBの重要度に依存し、BもAに依存
解法	全員同時に少しずつ動かして収束させる	全員同時に少しずつ更新して収束させる
1ステップの不正確さ	他も動くので想定通りにならない	他も変わるので想定通りにならない
収束する理由	減衰 + 温度低下	行列の固有ベクトルへの収束が数学的に証明されている

PageRankが解析的にも解ける理由

PageRankの場合、更新式は **r(t+1) = M × r(t)** という線形写像です。この反復は行列 M の**最大固有ベクトル**に収束します（Perron-Frobeniusの定理）。

連立方程式として書けば:

```
x = 0.5y + 0.3z
y = 0.4x + 0.2z
z = 0.1x + 0.5y
```

定義は循環していますが、連立一次方程式には解が存在します。 反復法はこの解を力ずくで見つけていると解釈できます。

しかし実用上は、反復法が選ばれます:

解法	**計算量**	**論文10万件の場合**
直接解法（固有値分解）	$O(n^3)$	10^{15} 演算（非現実的）
反復法（べき乗法）	$O(n^2 \times$ 反復回数$)$	数十回で収束（実用的）

一方、力学シミュレーションは非線形であるため、 直接解法自体が困難です（セクション8.5参照）。

共通する本質

これらの問題に共通するのは、 **「全体が全体に依存する問題を、少しずつ同時更新して解く」** という計算パラダイムです。

1ステップごとの更新は厳密ではありませんが、 適切な減衰・収束条件のもとで反復を重ねることで、 相互依存が自然に解消され、安定した解に到達します。