

# 感染症数理解析と予測モデリング: Python による実践入門

---

このガイドは、疫学や数理モデリングの経験が全くない技術者でも、感染症数理解析の基礎から中級レベルの手法まで『最短』で理解し実装できるように作成した学習教材です。Google Colaboratory を使って、実際に手を動かしながら学んでいきましょう。

## 目次

1. 感染症数理解析とは何か？
  2. 感染症と伝播の基礎知識
  3. 実行環境の準備
  4. データの取得と前処理
  5. 基本的な時系列解析
  6. SIR モデルとその実装
  7. 年齢構造を持つ SIR モデル
  8. 空間構造を持つモデル
  9. 確率論的モデルの導入
  10. パラメータ推定
  11. 機械学習を用いた予測
  12. 再生産数と流行の閾値
  13. 介入効果のシミュレーション
  14. 実データを用いた検証
  15. 実践的なミニプロジェクト
  16. よくある質問と回答
  17. 次のステップ
- 

## 1. 感染症数理解析とは何か？

感染症数理解析は、数学的モデルを用いて感染症の発生、拡大、そして収束のプロセスを理解・分析・予測するための学問分野です。

### 感染症数理解析の目的

- **流行予測:** 感染症の拡大速度や規模を予測
- **介入効果の評価:** ワクチン接種や行動制限などの効果を定量的に分析
- **リスク評価:** 特定の人口集団や地域における流行リスクの評価
- **資源配分の最適化:** 医療資源や予防対策の効率的な配分を支援

### 感染症数理解析の応用分野

- **公衆衛生政策:** 政策立案者への科学的根拠の提供
- **医療計画:** 病床や医療従事者の需要予測
- **ワクチン戦略:** 接種スケジュールや対象グループの最適化
- **パンデミック対応:** 新興感染症への迅速な対応計画の策定

- **経済影響分析:** 感染症流行の経済的影響の予測と緩和策の検討

なぜ感染症数理解析を学ぶべきか

- **科学的根拠に基づく意思決定:** 直感ではなくデータと数学に基づいた判断が可能に
  - **複雑な相互作用の理解:** 人口構造や接触パターンなど多要因を考慮した分析
  - **未来予測と「もし~だったら」分析:** 様々なシナリオでの流行予測が可能
  - **データサイエンスの実践的応用:** 統計学や機械学習の疫学的応用力が養える
- 

## 2. 感染症と伝播の基礎知識

感染症数理モデルを構築する前に、感染症と伝播に関する基本的な知識を理解しておくことが重要です。

感染症の基本特性

- **潜伏期間:** 感染から症状発現までの期間
- **感染期間:** 他者に感染させうる期間
- **基本再生産数 ( $R_0$ ):** 一人の感染者が生み出す二次感染者数の平均値
- **実効再生産数 ( $R_t$ ):** 特定の時点での再生産数（集団免疫や介入効果を反映）
- **世代時間:** 感染者が別の人を感染させるまでの平均時間

感染経路の種類

1. **飛沫感染:** 咳やくしゃみによる飛沫を介した感染
2. **接触感染:** 汚染表面や感染者との直接接触による感染
3. **空気感染:** より小さな粒子（エアロゾル）による遠距離感染
4. **媒介物感染:** 食品、水、生物媒介（ベクター）による感染

人口集団と感染症伝播

- **均質集団と非均質集団:** 人口集団内の接触や感受性の均一性・不均一性
- **年齢構造:** 年齢によって異なる感受性や接触率
- **空間構造:** 地理的分布と移動パターン
- **社会ネットワーク:** 人々の接触関係のネットワーク構造

集団免疫と閾値

- **集団免疫:** 感染に対する免疫を持つ個体の割合が閾値を超えることで、未免疫の個体も間接的に保護される現象
  - **ワクチンによる集団免疫:** 予防接種による人為的な集団免疫の獲得
  - **自然免疫:** 感染回復による免疫獲得
  - **感染症の閾値定理:**  $R_0 > 1$  のとき流行が発生し、 $R_0 < 1$  のとき流行は収束する
- 

## 3. 実行環境の準備

おすすめ環境

1. **Google Colab** (初心者向け) : インストール不要ですぐに始められる

## 2. Jupyter Notebook (慣れてきたら) : ローカル環境での開発に適している

### Google Colab の使い方

1. ブラウザで <https://colab.research.google.com/> を開く
2. 「新しいノートブック」をクリック
3. 以下のパッケージをインストール

```
# 必要なライブラリのインストール
!pip install numpy matplotlib scipy pandas seaborn scikit-learn epyestim
statsmodels networkx pystan (or cmdstanpy) pyro-ppl tqdm
```

### 必要なライブラリのインポート

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as spi
import pandas as pd
import seaborn as sns
from sklearn.metrics import r2_score, mean_squared_error
from statsmodels.tsa.seasonal import seasonal_decompose
import networkx as nx
from tqdm.notebook import tqdm
```

⌚ 初心者向けヒント : SciPy は微分方程式を解くための強力なライブラリで、感染症モデルの実装に重要です。Pandas は疫学データの操作と分析に便利です。

## 4. データの取得と前処理

感染症数理解析の最初のステップは、適切なデータの取得と前処理です。

### 感染症データの取得

```
def load_epidemic_data(source='jhu', disease='covid-19', region='Japan'):
    """感染症データを読み込む関数"""
    if source == 'jhu':
        # JHU CSSEのCOVID-19データの読み込み
        url = "https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/csse_covid_19_data/csse_covid_19_time_series/time_series_covid19_confirmed_global.csv"
        df = pd.read_csv(url)

        # 特定の国データを抽出
        country_data = df[df['Country/Region'] == region]

        # 日付列のみを抽出して転置
```

```
date_cols = country_data.columns[4:]
cases = country_data[date_cols].values[0]
dates = pd.to_datetime(date_cols, format='%m/%d/%y')

# データフレームに整形
data = pd.DataFrame({
    'date': dates,
    'cumulative_cases': cases
})

# 日次新規症例数を計算
data['new_cases'] = data['cumulative_cases'].diff().fillna(0)

elif source == 'sample':
    # サンプルの流行曲線データを生成
    np.random.seed(42)
    days = 120
    t = np.arange(days)

    # SIR風の流行曲線を生成
    peak = 30
    data = pd.DataFrame({
        'date': pd.date_range(start='2023-01-01', periods=days),
        'new_cases': 100 * np.exp(0.2 * t) / (1 + np.exp(0.2 * (t - peak)))
    })
    # ランダムなノイズを加える
    data['new_cases'] = data['new_cases'] * (1 + 0.2 * np.random.randn(days))
    data['new_cases'] = data['new_cases'].round().astype(int)
    data['cumulative_cases'] = data['new_cases'].cumsum()

else:
    raise ValueError("sourceは'jhu'または'sample'を指定してください")

# 7日間移動平均の計算
data['new_cases_7day_avg'] = data['new_cases'].rolling(window=7,
center=False).mean()

return data

# 例: サンプルデータを読み込む
epidemic_data = load_epidemic_data(source='sample')

# データの最初の数行を表示
print(epidemic_data.head())

# 流行曲線のプロット
plt.figure(figsize=(14, 6))
plt.plot(epidemic_data['date'], epidemic_data['new_cases'], 'o-', alpha=0.6,
label='日次新規症例数')
plt.plot(epidemic_data['date'], epidemic_data['new_cases_7day_avg'], 'r-',
linewidth=2, label='7日間移動平均')
plt.title('流行曲線', fontsize=15)
plt.xlabel('日付')
plt.ylabel('新規症例数')
```

```
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
plt.tight_layout()
plt.show()
```

## データのクリーニングと前処理

```
def preprocess_epidemic_data(data, smooth=True, remove_outliers=False,
fill_missing=True):
    """感染症データの前処理を行う関数"""

    # データのコピーを作成
    processed_data = data.copy()

    # 1. 外れ値の処理（オプション）
    if remove_outliers:
        # IQRを使用した外れ値の検出と処理
        Q1 = processed_data['new_cases'].quantile(0.25)
        Q3 = processed_data['new_cases'].quantile(0.75)
        IQR = Q3 - Q1

        # 外れ値の閾値
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        # 外れ値を特定
        outliers = (processed_data['new_cases'] < lower_bound) |
(processed_data['new_cases'] > upper_bound)

        if outliers.sum() > 0:
            print(f"検出された外れ値の数: {outliers.sum()}")

        # 外れ値を前後の値の平均で置換
        for idx in processed_data.index[outliers]:
            if idx > 0 and idx < len(processed_data) - 1:
                prev_val = processed_data.loc[idx-1, 'new_cases']
                next_val = processed_data.loc[idx+1, 'new_cases']
                processed_data.loc[idx, 'new_cases'] = (prev_val + next_val) / 2

    # 2. 欠損値の補完（オプション）
    if fill_missing and processed_data['new_cases'].isna().any():
        # 線形補間で欠損値を埋める
        processed_data['new_cases'] =
processed_data['new_cases'].interpolate(method='linear')

    # 3. 平滑化（オプション）
    if smooth:
        # 7日間移動平均による平滑化
        processed_data['new_cases_smooth'] =
processed_data['new_cases'].rolling(window=7, center=True).mean()
```

```

# 端部の処理
processed_data['new_cases_smooth'] =
processed_data['new_cases_smooth'].fillna(processed_data['new_cases'])

# 4. 累積症例数の再計算
processed_data['cumulative_cases'] = processed_data['new_cases'].cumsum()

return processed_data

# 例: データの前処理を実行
processed_data = preprocess_epidemic_data(epidemic_data, smooth=True,
remove_outliers=True)

# 前処理前後のデータの比較
plt.figure(figsize=(14, 8))

plt.subplot(2, 1, 1)
plt.plot(epidemic_data['date'], epidemic_data['new_cases'], 'o-', alpha=0.6,
label='元データ')
plt.title('前処理前の新規症例数', fontsize=15)
plt.ylabel('症例数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(processed_data['date'], processed_data['new_cases'], 'o-', alpha=0.6,
label='前処理後')
plt.plot(processed_data['date'], processed_data['new_cases_smooth'], 'r-',
linewidth=2, label='平滑化曲線')
plt.title('前処理後の新規症例数', fontsize=15)
plt.xlabel('日付')
plt.ylabel('症例数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

plt.tight_layout()
plt.show()

```

## 年齢構造データの準備

```

def prepare_age_structured_data(n_age_groups=9, use_real=False):
    """年齢構造を持つデータを準備する関数"""

    if use_real:
        # 実際の年齢別人口構成 (日本2020年、100万人単位)
        age_population = pd.DataFrame({
            'age_group': ['0-9', '10-19', '20-29', '30-39', '40-49', '50-59', '60-69',
            '70-79', '80+'],
            'population': [10.3, 10.8, 12.3, 14.2, 18.1, 16.8, 15.6, 11.7, 9.3]
        })
    else:

```

```

# サンプルの年齢構成を生成
age_groups = [f"{i*10}-{(i+1)*10-1}" for i in range(n_age_groups-1)]
age_groups.append(f"({n_age_groups-1}*10)+")

# 年齢分布（若年層と高齢層で人口が多い二峰性の分布）
population = 10 + 5 * np.sin(np.linspace(0, np.pi, n_age_groups)) +
np.random.rand(n_age_groups)

age_population = pd.DataFrame({
    'age_group': age_groups,
    'population': population
})

# 人口の合計が1になるように正規化
total_population = age_population['population'].sum()
age_population['proportion'] = age_population['population'] / total_population

return age_population

# 年齢構造データの準備
age_data = prepare_age_structured_data(use_real=True)

# 年齢構成の可視化
plt.figure(figsize=(12, 6))
plt.bar(age_data['age_group'], age_data['population'], color='steelblue',
alpha=0.7)
plt.title('年齢層別人口構成', fontsize=15)
plt.xlabel('年齢層')
plt.ylabel('人口 (100万人)')
plt.grid(True, linestyle='--', alpha=0.7, axis='y')
plt.tight_layout()
plt.show()

```

💡 **初心者向けヒント**：感染症データは日々更新されるため、常に最新のソースを参照することが重要です。また、報告遅延や週末効果などの影響を考慮した前処理も必要になることがあります。

## 5. 基本的な時系列解析

感染症データは本質的に時系列データであり、時系列分析手法を適用することで重要な洞察が得られます。

### トレンドと季節性の分解

```

def decompose_time_series(data, column='new_cases_smooth', period=7):
    """時系列データをトレンド、季節性、残差に分解する関数"""

    # 季節分解（加法モデル）
    decomposition = seasonal_decompose(data[column], model='additive',
period=period)

    # 結果の表示

```

```

plt.figure(figsize=(14, 10))

plt.subplot(4, 1, 1)
plt.plot(data['date'], data[column], 'o-', markersize=3)
plt.title('元の時系列', fontsize=15)
plt.grid(True, linestyle='--', alpha=0.7)

plt.subplot(4, 1, 2)
plt.plot(data['date'], decomposition.trend, 'r-')
plt.title('トレンド成分', fontsize=15)
plt.grid(True, linestyle='--', alpha=0.7)

plt.subplot(4, 1, 3)
plt.plot(data['date'], decomposition.seasonal, 'g-')
plt.title('季節性成分', fontsize=15)
plt.grid(True, linestyle='--', alpha=0.7)

plt.subplot(4, 1, 4)
plt.plot(data['date'], decomposition.resid, 'b-')
plt.title('残差成分', fontsize=15)
plt.grid(True, linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()

return decomposition

```

# 時系列分解の実行

```

decomposition = decompose_time_series(processed_data, column='new_cases_smooth',
period=7)

```

## 成長率の計算と分析

```

def analyze_growth_rate(data, window=7):
    """感染の成長率を計算して分析する関数"""

    # データのコピーを作成
    growth_data = data.copy()

    # 単純な日次成長率 (t日の症例数 / t-1日の症例数)
    growth_data['daily_growth_rate'] = growth_data['new_cases'] /
growth_data['new_cases'].shift(1)

    # 指数成長率 (log(症例数)の差分)
    growth_data['log_cases'] = np.log(growth_data['new_cases'].replace(0, 0.1))  #
0を0.1に置換してlogを計算
    growth_data['exponential_growth_rate'] = growth_data['log_cases'].diff()

    # n日間の移動平均成長率
    growth_data[f'{window}day_avg_growth'] =
growth_data['new_cases'].pct_change(periods=window) / window

```

```
# 倍加時間の計算 (log(2) / 成長率)
growth_data['doubling_time'] = np.log(2) /
growth_data[f'{window}day_avg_growth']

# 成長率の可視化
plt.figure(figsize=(14, 10))

plt.subplot(3, 1, 1)
plt.plot(growth_data['date'], growth_data['new_cases_smooth'], 'b-', linewidth=2)
plt.title('新規症例数 (平滑化)', fontsize=15)
plt.grid(True, linestyle='--', alpha=0.7)
plt.ylabel('症例数')

plt.subplot(3, 1, 2)
plt.plot(growth_data['date'], growth_data[f'{window}day_avg_growth'], 'r-', linewidth=2)
plt.axhline(y=0, color='k', linestyle='--', alpha=0.3)
plt.title(f'{window}日間の平均成長率', fontsize=15)
plt.grid(True, linestyle='--', alpha=0.7)
plt.ylabel('成長率 (日平均)')

plt.subplot(3, 1, 3)
# 倍加時間はマイナスになるとグラフが見にくくなるので、適切な範囲でクリップ
doubling_time = growth_data['doubling_time'].clip(-30, 30)
plt.plot(growth_data['date'], doubling_time, 'g-', linewidth=2)
plt.axhline(y=0, color='k', linestyle='--', alpha=0.3)
plt.title('倍加時間 (日数)', fontsize=15)
plt.grid(True, linestyle='--', alpha=0.7)
plt.ylabel('日数')

plt.tight_layout()
plt.show()

return growth_data

# 成長率分析の実行
growth_analysis = analyze_growth_rate(processed_data, window=7)
```

## ホットスポット分析

```
def hotspot_analysis(data, threshold=0.1, window=7):
    """急速に症例が増加している期間（ホットスポット）を検出する関数"""

    # 成長率の計算 (n日間の変化率)
    growth_rate = data['new_cases_smooth'].pct_change(periods=window)

    # ホットスポットの定義：成長率が閾値を超える期間
    hotspots = growth_rate > threshold
```

```
# 連続した日のホットスポットをグループ化
hotspot_groups = []
current_group = None

for idx, (date, is_hotspot) in enumerate(zip(data['date'], hotspots)):
    if is_hotspot:
        if current_group is None:
            current_group = {'start_idx': idx, 'start_date': date}
        elif current_group is not None:
            current_group['end_idx'] = idx - 1
            current_group['end_date'] = data['date'].iloc[idx - 1]
            hotspot_groups.append(current_group)
            current_group = None

# 最後のグループを処理
if current_group is not None:
    current_group['end_idx'] = len(data) - 1
    current_group['end_date'] = data['date'].iloc[-1]
    hotspot_groups.append(current_group)

# ホットスポットの可視化
plt.figure(figsize=(14, 6))

plt.plot(data['date'], data['new_cases_smooth'], 'b-', linewidth=2)

# 各ホットスポットを強調表示
for group in hotspot_groups:
    start_idx = group['start_idx']
    end_idx = group['end_idx']
    plt.axvspan(data['date'].iloc[start_idx], data['date'].iloc[end_idx],
                color='red', alpha=0.3)

plt.title(f'感染症ホットスポット分析 (成長率閾値: {threshold*100:.1f}%)',
          fontsize=15)
plt.xlabel('日付')
plt.ylabel('新規症例数 (平滑化)')
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

# ホットスポット期間の詳細を返す
hotspot_details = []
for group in hotspot_groups:
    start_date = group['start_date']
    end_date = group['end_date']
    duration = (end_date - start_date).days + 1

    # この期間の平均成長率
    period_growth = data['new_cases_smooth'].iloc[group['end_idx']] /
    data['new_cases_smooth'].iloc[group['start_idx']]
    daily_growth = period_growth ** (1/duration) - 1

    hotspot_details.append({
        'start_date': start_date,
```

```

        'end_date': end_date,
        'duration': duration,
        'avg_daily_growth': daily_growth
    })

return pd.DataFrame(hotspot_details)

# ホットスポット分析の実行
hotspots = hotspot_analysis(processed_data, threshold=0.05, window=7)

# 検出されたホットスポットの詳細を表示
if not hotspots.empty:
    print("検出されたホットスポット期間:")
    for _, row in hotspots.iterrows():
        print(f"期間: {row['start_date'].date()} から {row['end_date'].date()}")
        print(f"({row['duration']}日間)")
        print(f"平均日次成長率: {row['avg_daily_growth']*100:.2f}%")
        print("-" * 40)
else:
    print("ホットスポットは検出されませんでした。")

```

💡 **初心者向けヒント** : 成長率は感染症の拡大速度を把握するための重要な指標です。特に指数関数的成長の初期段階では、倍加時間（症例数が 2 倍になる時間）は直感的に理解しやすい指標となります。

## 6. SIR モデルとその実装

SIR (Susceptible-Infected-Recovered) モデルは、感染症数理モデルの基本となるコンパートメントモデルです。

### 基本的な SIR モデル

```

def sir_model(y, t, beta, gamma, N):
    """基本的なSIRモデルの微分方程式"""
    S, I, R = y

    # dS/dt = -beta * S * I / N
    dSdt = -beta * S * I / N

    # dI/dt = beta * S * I / N - gamma * I
    dIdt = beta * S * I / N - gamma * I

    # dR/dt = gamma * I
    dRdt = gamma * I

    return [dSdt, dIdt, dRdt]

def simulate_sir(N, beta, gamma, I0, days):
    """SIRモデルのシミュレーションを実行する関数"""

```

```
# 初期条件
S0 = N - I0
R0 = 0
y0 = [S0, I0, R0]

# 時間軸
t = np.linspace(0, days, days + 1)

# SIRモデルを解く
solution = spi.solve_ivp(
    fun=lambda t, y: sir_model(y, t, beta, gamma, N),
    t_span=[t[0], t[-1]],
    y0=y0,
    t_eval=t
)

# 結果の整形
S = solution.y[0]
I = solution.y[1]
R = solution.y[2]

# 日次の新規感染者数を計算 (S の減少分)
new_cases = np.zeros_like(S)
new_cases[1:] = -(S[1:] - S[:-1])

results = {
    't': t,
    'S': S,
    'I': I,
    'R': R,
    'new_cases': new_cases
}

return results

def plot_sir_simulation(results, N, beta, gamma):
    """SIRシミュレーション結果をプロットする関数"""

    t = results['t']
    S = results['S']
    I = results['I']
    R = results['R']
    new_cases = results['new_cases']

    # 基本再生産数の計算
    R0 = beta / gamma

    plt.figure(figsize=(15, 10))

    # SIR曲線のプロット
    plt.subplot(2, 1, 1)
    plt.plot(t, S, 'b-', label='感受性者 (S)', linewidth=2)
    plt.plot(t, I, 'r-', label='感染者 (I)', linewidth=2)
    plt.plot(t, R, 'g-', label='回復者 (R)', linewidth=2)
```

```

plt.title(f'SIRモデルシミュレーション (人口={N:,}, R0={R0:.2f})', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('人数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# 新規感染者数のプロット
plt.subplot(2, 1, 2)
plt.bar(t, new_cases, color='orange', alpha=0.6, label='日次新規感染者数')
plt.title('日次新規感染者数', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('人数')
plt.grid(True, linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()

# 主要な結果を表示
max_infected = np.max(I)
max_infected_day = t[np.argmax(I)]
total_infected = R[-1]

print(f"基本再生産数 (R0): {R0:.2f}")
print(f"最大感染者数: {max_infected:.0f} 人 (全人口の{max_infected/N*100:.2f}%)")
print(f"感染ピーク日: {max_infected_day:.0f} 日目")
print(f"最終的な累積感染者数: {total_infected:.0f} 人 (全人口の{total_infected/N*100:.2f}%)")

# SIRモデルのシミュレーション実行
N = 100000 # 人口
beta = 0.3 # 感染率
gamma = 0.1 # 回復率
I0 = 10 # 初期感染者数
days = 180 # シミュレーション日数

sir_results = simulate_sir(N, beta, gamma, I0, days)

# シミュレーション結果のプロット
plot_sir_simulation(sir_results, N, beta, gamma)

```

## パラメータの影響分析

```

def analyze_sir_parameters(N, I0, days, beta_range, gamma_range):
    """異なるSIRパラメータの影響を分析する関数"""

    # パラメータの組み合わせを生成
    beta_values = np.linspace(beta_range[0], beta_range[1], 5)
    gamma_values = np.linspace(gamma_range[0], gamma_range[1], 3)

```

```
# 結果を格納する配列
results = []

# 各パラメータの組み合わせでシミュレーション
for beta in beta_values:
    for gamma in gamma_values:
        R0 = beta / gamma

        # シミュレーション実行
        sim_results = simulate_sir(N, beta, gamma, I0, days)

        # 主要な結果を記録
        max_infected = np.max(sim_results['I'])
        max_infected_day = sim_results['t'][np.argmax(sim_results['I'])]
        total_infected = sim_results['R'][-1]

        results.append({
            'beta': beta,
            'gamma': gamma,
            'R0': R0,
            'max_infected': max_infected,
            'max_infected_day': max_infected_day,
            'total_infected': total_infected,
            'sim_results': sim_results
        })

# R0別の感染曲線を可視化
plt.figure(figsize=(15, 10))

# 感染者数の時間推移
plt.subplot(2, 1, 1)
for res in results:
    if res['gamma'] == gamma_values[1]: # 中央のgamma値に対してのみプロット
        plt.plot(res['sim_results']['t'], res['sim_results']['I'],
                  label=f"R₀={res['R0']:.2f} (β={res['beta']:.2f}, γ={res['gamma']:.2f})")

plt.title('R₀が感染者数の推移に与える影響', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# R0とピーク感染者数の関係
plt.subplot(2, 1, 2)
R0_values = [res['R0'] for res in results]
peak_values = [res['max_infected'] for res in results]

plt.scatter(R0_values, peak_values, c=peak_values, cmap='viridis', s=100,
alpha=0.7)
plt.colorbar(label='ピーク感染者数')

plt.title('R₀とピーク感染者数の関係', fontsize=15)
plt.xlabel('基本再生産数 (R₀)')
```

```

plt.ylabel('ピーク感染者数')
plt.grid(True, linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()

# 結果をデータフレームにまとめて返す
results_df = pd.DataFrame([
    {k: v for k, v in res.items() if k != 'sim_results'}
    for res in results
])

return results_df

# パラメータの影響を分析
param_analysis = analyze_sir_parameters(
    N=100000,
    I0=10,
    days=180,
    beta_range=[0.1, 0.5],
    gamma_range=[0.05, 0.25]
)

# 分析結果の確認
print(param_analysis[['beta', 'gamma', 'R0', 'max_infected',
    'max_infected_day']].sort_values('R0'))

```

## SEIR モデル（潜伏期間を考慮）

```

def seir_model(y, t, beta, sigma, gamma, N):
    """SEIRモデルの微分方程式
    S: 感受性者, E: 潜伏期間中の感染者, I: 感染性のある感染者, R: 回復者
    """
    S, E, I, R = y

    # dS/dt = -beta * S * I / N
    dSdt = -beta * S * I / N

    # dE/dt = beta * S * I / N - sigma * E
    dEdt = beta * S * I / N - sigma * E

    # dI/dt = sigma * E - gamma * I
    dIdt = sigma * E - gamma * I

    # dR/dt = gamma * I
    dRdt = gamma * I

    return [dSdt, dEdt, dIdt, dRdt]

def simulate_seir(N, beta, sigma, gamma, E0, I0, days):
    """SEIRモデルのシミュレーションを実行する関数"""

```

```
# 初期条件
S0 = N - E0 - I0
R0 = 0
y0 = [S0, E0, I0, R0]

# 時間軸
t = np.linspace(0, days, days + 1)

# SEIRモデルを解く
solution = spi.solve_ivp(
    fun=lambda t, y: seir_model(y, t, beta, sigma, gamma, N),
    t_span=[t[0], t[-1]],
    y0=y0,
    t_eval=t
)

# 結果の整形
S = solution.y[0]
E = solution.y[1]
I = solution.y[2]
R = solution.y[3]

# 日次の新規感染者（E）を計算（S の減少分）
new_exposures = np.zeros_like(S)
new_exposures[1:] = -(S[1:] - S[:-1])

# 日次の新規発症者（I）を計算（E → I の遷移）
new_cases = np.zeros_like(S)
new_cases[1:] = sigma * E[:-1] # 近似計算

results = {
    't': t,
    'S': S,
    'E': E,
    'I': I,
    'R': R,
    'new_exposures': new_exposures,
    'new_cases': new_cases
}

return results

def plot_seir_simulation(results, N, beta, sigma, gamma):
    """SEIRシミュレーション結果をプロットする関数"""

    t = results['t']
    S = results['S']
    E = results['E']
    I = results['I']
    R = results['R']
    new_cases = results['new_cases']

    # 基本再生産数と平均潜伏期間の計算
```

```
R0 = beta / gamma
incubation_period = 1 / sigma
infectious_period = 1 / gamma

plt.figure(figsize=(15, 12))

# SEIR曲線のプロット
plt.subplot(3, 1, 1)
plt.plot(t, S, 'b-', label='感受性者 (S)', linewidth=2)
plt.plot(t, E, 'y-', label='潜伏期間中 (E)', linewidth=2)
plt.plot(t, I, 'r-', label='感染者 (I)', linewidth=2)
plt.plot(t, R, 'g-', label='回復者 (R)', linewidth=2)

title = f'SEIRモデルシミュレーション\n'
title += f'人口={N:,}, R₀={R0:.2f}, 潜伏期間={incubation_period:.1f}日, 感染期間={infectious_period:.1f}日'
plt.title(title, fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('人数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# E+I ( 総感染者数 ) のプロット
plt.subplot(3, 1, 2)
plt.plot(t, E + I, 'purple', label='総感染者数 (E+I)', linewidth=2)
plt.plot(t, E, 'y--', label='潜伏期間中 (E)', linewidth=1.5)
plt.plot(t, I, 'r--', label='感染者 (I)', linewidth=1.5)

plt.title('総感染者数の推移', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('人数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# 新規発症者数のプロット
plt.subplot(3, 1, 3)
plt.bar(t, new_cases, color='orange', alpha=0.6, label='日次新規発症者数')
plt.title('日次新規発症者数', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('人数')
plt.grid(True, linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()

# 主要な結果を表示
max_infected = np.max(E + I)
max_infected_day = t[np.argmax(E + I)]
total_infected = R[-1]

print(f"基本再生産数 (R₀): {R0:.2f}")
print(f"平均潜伏期間: {incubation_period:.1f} 日")
print(f"平均感染期間: {infectious_period:.1f} 日")
print(f"最大感染者数 (E+I): {max_infected:.0f} 人 (全人口の")
```

```

{max_infected/N*100:.2f}%)")
print(f"感染ピーク日: {max_infected_day:.0f} 日目")
print(f"最終的な累積感染者数: {total_infected:.0f} 人 (全人口の
{total_infected/N*100:.2f}%)")

# SEIRモデルのシミュレーション実行
N = 100000    # 人口
beta = 0.3     # 感染率
sigma = 1/5.0   # 潜伏期間の逆数 ( 1/平均潜伏期間 )
gamma = 1/7.0   # 回復率 ( 1/平均感染期間 )
E0 = 10         # 初期潜伏期間中の感染者数
I0 = 5          # 初期感染者数
days = 180       # シミュレーション日数

seir_results = simulate_seir(N, beta, sigma, gamma, E0, I0, days)

# シミュレーション結果のプロット
plot_seir_simulation(seir_results, N, beta, sigma, gamma)

```

⌚ **初心者向けヒント** : SIR モデルは感染症の流行動態を理解するための基本的なフレームワークです。実際の感染症に適用する際は、より複雑なモデル (SEIR, SEIRS, SIRD など) が必要になることもあります。SEIR モデルは COVID-19 などの潜伏期間のある感染症のモデル化に適しています。

## 7. 年齢構造を持つ SIR モデル

実際の人口は均質ではなく、年齢による感受性や接触パターンの違いがあります。年齢構造を持つモデルはより現実的な予測を可能にします。

### 接触行列の作成

```

def create_contact_matrix(n_age_groups, contact_type='assortative'):
    """年齢層間の接触行列を作成する関数"""

    if contact_type == 'uniform':
        # 均一な接触パターン (全ての年齢層間で同じ接触率)
        contact_matrix = np.ones((n_age_groups, n_age_groups))

    elif contact_type == 'assortative':
        # 同年齢層との接触が多いパターン
        contact_matrix = np.zeros((n_age_groups, n_age_groups))
        for i in range(n_age_groups):
            for j in range(n_age_groups):
                # 年齢層の差に基づいて接触率を設定
                distance = abs(i - j)
                contact_matrix[i, j] = np.exp(-distance / 2.0)

    elif contact_type == 'realistic':
        # より現実的な接触パターン (例: 多世代家族や学校環境を反映)
        contact_matrix = np.zeros((n_age_groups, n_age_groups))

```

```
# 同年齢層との接触（対角要素）
for i in range(n_age_groups):
    contact_matrix[i, i] = 5.0

# 近い年齢層との接触（オフ対角要素）
for i in range(n_age_groups):
    for j in range(n_age_groups):
        if i != j:
            distance = abs(i - j)
            contact_matrix[i, j] = 3.0 * np.exp(-distance / 1.5)

# 特定のパターンを追加

# 子供と親の世代の接触を強化
for i in range(2): # 0-19歳
    for j in range(3, 6): # 30-59歳
        contact_matrix[i, j] *= 2.0
        contact_matrix[j, i] *= 2.0

# 高齢者のケアによる接触
for i in range(n_age_groups - 2, n_age_groups): # 70歳以上
    for j in range(4, 7): # 40-69歳
        contact_matrix[i, j] *= 1.5
        contact_matrix[j, i] *= 1.5

else:
    raise ValueError("contact_type は 'uniform', 'assortative', 'realistic' のいずれかを指定してください")

return contact_matrix

def plot_contact_matrix(contact_matrix, age_groups):
    """接觸行列を可視化する関数"""

    plt.figure(figsize=(10, 8))

    sns.heatmap(contact_matrix, annot=True, fmt=".2f", cmap="YlOrRd",
                xticklabels=age_groups, yticklabels=age_groups)

    plt.title('年齢層間の接觸行列', fontsize=15)
    plt.xlabel('接觸される年齢層')
    plt.ylabel('接觸する年齢層')

    plt.tight_layout()
    plt.show()

# 年齢層と人口比率の定義
age_data = prepare_age_structured_data(use_real=True)
age_groups = age_data['age_group'].tolist()
n_age_groups = len(age_groups)

# 接触行列の作成と可視化
contact_matrix = create_contact_matrix(n_age_groups, contact_type='realistic')
plot_contact_matrix(contact_matrix, age_groups)
```

## 年齢別の感受性と重症度

```
def define_age_specific_parameters(age_groups, baseline_beta, baseline_gamma):
    """年齢別のパラメータを定義する関数"""

    n_groups = len(age_groups)

    # 年齢による感受性の違い（若年層の感受性を低く、高齢層の感受性を高く設定）
    susceptibility = np.ones(n_groups)
    for i in range(n_groups):
        # 年齢が上がるにつれて感受性が上昇
        age_midpoint = int(age_groups[i].split('-')[0]) if '-' in age_groups[i]
    else 80
        susceptibility[i] = 0.5 + 1.0 * (age_midpoint / 80)

    # 年齢による重症度（回復率の逆数）の違い
    severity = np.ones(n_groups)
    for i in range(n_groups):
        # 年齢が上がるにつれて重症度が上昇（回復率が低下）
        age_midpoint = int(age_groups[i].split('-')[0]) if '-' in age_groups[i]
    else 80
        severity[i] = 0.5 + 1.5 * (age_midpoint / 80)

    # 年齢別のbetaとgammaの計算
    beta_by_age = baseline_beta * susceptibility
    gamma_by_age = baseline_gamma / severity

    # 結果をデータフレームにまとめる
    params_df = pd.DataFrame({
        'age_group': age_groups,
        'susceptibility': susceptibility,
        'severity': severity,
        'beta': beta_by_age,
        'gamma': gamma_by_age,
        'R0': beta_by_age / gamma_by_age
    })

    return params_df

# 年齢別パラメータの定義
age_params = define_age_specific_parameters(age_groups, baseline_beta=0.3,
baseline_gamma=0.1)

# パラメータの可視化
plt.figure(figsize=(14, 10))

plt.subplot(2, 1, 1)
plt.bar(age_params['age_group'], age_params['susceptibility'], color='blue',
alpha=0.7)
plt.title('年齢層別の相対的感受性', fontsize=15)
```

```

plt.ylabel('相対感受性')
plt.grid(True, linestyle='--', alpha=0.7, axis='y')

plt.subplot(2, 1, 2)
plt.bar(age_params['age_group'], age_params['severity'], color='red', alpha=0.7)
plt.title('年齢層別の相対的重症度', fontsize=15)
plt.ylabel('相対重症度')
plt.grid(True, linestyle='--', alpha=0.7, axis='y')

plt.tight_layout()
plt.show()

# 年齢別のR₀を確認
plt.figure(figsize=(10, 6))
plt.bar(age_params['age_group'], age_params['R₀'], color='purple', alpha=0.7)
plt.title('年齢層別の基本再生産数 (R₀)', fontsize=15)
plt.ylabel('R₀')
plt.grid(True, linestyle='--', alpha=0.7, axis='y')
plt.tight_layout()
plt.show()

```

## 年齢構造化 SIR モデル

```

def age_structured_sir_model(y, t, params):
    """年齢構造を持つSIRモデルの微分方程式"""
    n_groups = params['n_groups']
    N = params['N']
    beta = params['beta']
    gamma = params['gamma']
    cm = params['contact_matrix']
    susceptibility = params['susceptibility']

    # 状態変数の分解
    S = y[:n_groups]
    I = y[n_groups:2*n_groups]
    R = y[2*n_groups:]

    # 各コンパートメントの変化率を格納する配列
    dSdt = np.zeros(n_groups)
    dIdt = np.zeros(n_groups)
    dRdt = np.zeros(n_groups)

    # 各年齢層について計算
    for i in range(n_groups):
        # 他の全年齢層からの感染力
        infection_force = 0
        for j in range(n_groups):
            infection_force += beta[i] * susceptibility[i] * cm[i, j] * I[j] /
N[j]

        # dS/dt = -S[i] * infection_force
        dSdt[i] = -infection_force * S[i]
        dIdt[i] = infection_force * S[i]
        dRdt[i] = gamma * I[i]

```

```
dSdt[i] = -S[i] * infection_force

# dI/dt = S[i] * infection_force - gamma[i] * I[i]
dIdt[i] = S[i] * infection_force - gamma[i] * I[i]

# dR/dt = gamma[i] * I[i]
dRdt[i] = gamma[i] * I[i]

return np.concatenate([dSdt, dIdt, dRdt])

def simulate_age_structured_sir(age_data, age_params, contact_matrix,
initial_infected, days):
    """年齢構造化SIRモデルのシミュレーションを実行する関数"""

n_groups = len(age_data)

# 年齢層別の人団
population_by_age = age_data['population'].values
total_population = population_by_age.sum()

# 各年齢層の人口比率に基づいて初期感染者を配分
if isinstance(initial_infected, int):
    I0 = population_by_age / total_population * initial_infected
    I0 = np.round(I0).astype(int)
    # 合計が元の値になるように調整
    while I0.sum() != initial_infected:
        if I0.sum() < initial_infected:
            idx = np.random.choice(range(n_groups))
            I0[idx] += 1
        else:
            idx = np.random.choice(range(n_groups))
            if I0[idx] > 0:
                I0[idx] -= 1
else:
    # 年齢層別に初期感染者数が指定されている場合
    I0 = np.array(initial_infected)

# 初期条件の設定
S0 = population_by_age - I0
R0 = np.zeros(n_groups)
y0 = np.concatenate([S0, I0, R0])

# モデルパラメータの設定
params = {
    'n_groups': n_groups,
    'N': population_by_age,
    'beta': age_params['beta'].values,
    'gamma': age_params['gamma'].values,
    'contact_matrix': contact_matrix,
    'susceptibility': age_params['susceptibility'].values
}

# 時間軸
t = np.linspace(0, days, days + 1)
```

```
# 微分方程式を解く
solution = spi.solve_ivp(
    fun=lambda t, y: age_structured_sir_model(y, t, params),
    t_span=[t[0], t[-1]],
    y0=y0,
    t_eval=t
)

# 結果の整形
S = solution.y[:n_groups]
I = solution.y[n_groups:2*n_groups]
R = solution.y[2*n_groups:]

# 日次新規感染者数の計算
new_cases = np.zeros((n_groups, len(t)))
for i in range(n_groups):
    new_cases[i, 1:] = -(S[i, 1:] - S[i, :-1])

results = {
    't': t,
    'S': S,
    'I': I,
    'R': R,
    'new_cases': new_cases,
    'age_groups': age_data['age_group'].values,
    'population': population_by_age
}

return results

def plot_age_structured_results(results, age_params):
    """年齢構造化SIRモデルの結果を可視化する関数"""

    t = results['t']
    S = results['S']
    I = results['I']
    R = results['R']
    new_cases = results['new_cases']
    age_groups = results['age_groups']
    population = results['population']

    n_groups = len(age_groups)

    # カラーマップの設定
    colors = plt.cm.viridis(np.linspace(0, 1, n_groups))

    # 感染者数の推移
    plt.figure(figsize=(15, 10))

    plt.subplot(2, 1, 1)
    for i in range(n_groups):
        plt.plot(t, I[i], color=colors[i], label=f'{age_groups[i]}', linewidth=2)
```

```
plt.title('年齢層別の感染者数の推移', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(title='年齢層')

# 人口に対する感染率の推移
plt.subplot(2, 1, 2)
for i in range(n_groups):
    plt.plot(t, I[i] / population[i] * 100, color=colors[i],
label=f'{age_groups[i]}', linewidth=2)

plt.title('年齢層別の人団に占める感染率の推移', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('感染率 (%)')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(title='年齢層')

plt.tight_layout()
plt.show()

# 累積感染率の年齢層別比較
plt.figure(figsize=(10, 6))

final_attack_rate = R[:, -1] / population * 100

plt.bar(age_groups, final_attack_rate, color=colors, alpha=0.7)
plt.title('年齢層別の最終累積感染率', fontsize=15)
plt.xlabel('年齢層')
plt.ylabel('累積感染率 (%)')
plt.grid(True, linestyle='--', alpha=0.7, axis='y')

plt.tight_layout()
plt.show()

# 時間経過に伴う感染の波を可視化
plt.figure(figsize=(15, 10))

plt.subplot(2, 1, 1)
total_new_cases = np.sum(new_cases, axis=0)
plt.bar(t, total_new_cases, color='orange', alpha=0.6)
plt.title('全年齢層の日次新規感染者数', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('新規感染者数')
plt.grid(True, linestyle='--', alpha=0.7)

plt.subplot(2, 1, 2)
# ヒートマップでの可視化
plt.pcolormesh(t, np.arange(n_groups), new_cases, cmap='YlOrRd',
shading='auto')
plt.colorbar(label='新規感染者数')
plt.yticks(np.arange(n_groups) + 0.5, age_groups)
plt.title('年齢層別・日次新規感染者数', fontsize=15)
plt.xlabel('経過日数')
```

```

plt.ylabel('年齢層')

plt.tight_layout()
plt.show()

# R0などの主要な指標を出力
print("年齢層別の主要指標:")
for i in range(n_groups):
    peak_infected = np.max(I[i])
    peak_infected_day = t[np.argmax(I[i])]
    total_infected = R[i, -1]
    attack_rate = total_infected / population[i] * 100

    print(f"\n{age_groups[i]}:")
    print(f" R0: {age_params['R0'].iloc[i]:.2f}")
    print(f" 最大感染者数: {peak_infected:.0f} 人 (人口の
{peak_infected/population[i]*100:.2f}%)")
    print(f" 感染ピーク日: {peak_infected_day:.0f} 日目")
    print(f" 最終累積感染者数: {total_infected:.0f} 人 (人口の
{attack_rate:.2f}%)")

# 年齢構造化SIRモデルのシミュレーション実行
total_population = age_data['population'].sum()
initial_infected = 100 # 初期感染者総数

age_structured_results = simulate_age_structured_sir(
    age_data,
    age_params,
    contact_matrix,
    initial_infected,
    days=180
)

# シミュレーション結果のプロット
plot_age_structured_results(age_structured_results, age_params)

```

💡 **初心者向けヒント**： 年齢構造化モデルは、ワクチン接種戦略の最適化や、学校閉鎖などの年齢層別介入の効果を評価するのに役立ちます。さらに複雑なモデルでは、家庭・学校・職場といった接触の場所も区別することができます。

## 8. 空間構造を持つモデル

感染症は地理的空間の中で拡散するため、空間構造を考慮したモデルが重要です。

メタポピュレーションモデル

```

def create_spatial_network(n_regions, network_type='grid', params=None):
    """複数地域間のネットワークを作成する関数"""

    if network_type == 'grid':

```

```
# 格子状のネットワーク（隣接した地域のみ接続）
grid_size = int(np.ceil(np.sqrt(n_regions)))
G = nx.grid_2d_graph(grid_size, grid_size)

# 1次元のノード番号に変換
mapping = {(i, j): i * grid_size + j for i, j in G.nodes() if i * grid_size + j < n_regions}
G = nx.relabel_nodes(G, mapping)

# 範囲外のノードを削除
nodes_to_remove = [node for node in G.nodes() if node >= n_regions]
G.remove_nodes_from(nodes_to_remove)

elif network_type == 'random':
    # ランダムグラフ（エルデシュ・レニイモデル）
    p = params.get('p', 0.1) if params else 0.1
    G = nx.erdos_renyi_graph(n_regions, p)

elif network_type == 'small_world':
    # スモールワールドネットワーク（ワット・ストロガツモデル）
    k = params.get('k', 4) if params else 4
    p = params.get('p', 0.1) if params else 0.1
    G = nx.watts_strogatz_graph(n_regions, k, p)

elif network_type == 'scale_free':
    # スケールフリーネットワーク（バラバシ・アルバートモデル）
    m = params.get('m', 2) if params else 2
    G = nx.barabasi_albert_graph(n_regions, m)

elif network_type == 'realistic':
    # より現実的なネットワーク（距離に基づく接続確率）
    positions = np.random.rand(n_regions, 2) # 2D平面上のランダムな位置

    G = nx.Graph()
    G.add_nodes_from(range(n_regions))

    # 各ノードに位置情報付与
    for i in range(n_regions):
        G.nodes[i]['pos'] = positions[i]

    # 距離に反比例する確率で接続
    distance_scale = params.get('distance_scale', 0.2) if params else 0.2
    for i in range(n_regions):
        for j in range(i+1, n_regions):
            distance = np.linalg.norm(positions[i] - positions[j])
            p_connect = np.exp(-distance / distance_scale)

            if np.random.random() < p_connect:
                G.add_edge(i, j, weight=p_connect)

    else:
        raise ValueError("network_type は 'grid', 'random', 'small_world', 'scale_free', 'realistic' のいずれかを指定してください")
```

```
return G

def plot_spatial_network(G, node_attribute=None, title=None):
    """空間ネットワークを可視化する関数"""

    plt.figure(figsize=(12, 10))

    # ノードの位置情報
    if 'pos' in G.nodes[node] for node in G.nodes):
        pos = nx.get_node_attributes(G, 'pos')
    else:
        pos = nx.spring_layout(G, seed=42)

    # エッジの描画
    if 'weight' in G.edges[list(G.edges())[0]]:
        weights = [G[u][v]['weight'] * 3 for u, v in G.edges()]
        nx.draw_networkx_edges(G, pos, width=weights, alpha=0.3,
edge_color='gray')
    else:
        nx.draw_networkx_edges(G, pos, width=1, alpha=0.3, edge_color='gray')

    # ノードの描画
    if node_attribute:
        node_values = [G.nodes[node].get(node_attribute, 0) for node in G.nodes]
        nodes = nx.draw_networkx_nodes(G, pos, node_color=node_values,
                                         cmap=plt.cm.viridis, node_size=300)
        plt.colorbar(nodes, label=node_attribute)
    else:
        nx.draw_networkx_nodes(G, pos, node_color='skyblue', node_size=300)

    # ノード番号のラベル
    nx.draw_networkx_labels(G, pos, font_size=10)

    plt.title(title or '地域間ネットワーク', fontsize=15)
    plt.axis('off')
    plt.tight_layout()
    plt.show()

def metapopulation_sir_model(y, t, params):
    """メタポピュレーションSIRモデルの微分方程式"""
    G = params['network']
    n_regions = params['n_regions']
    beta = params['beta']
    gamma = params['gamma']
    N = params['N']
    mobility = params['mobility']

    # 状態変数の分解
    S = y[:n_regions]
    I = y[n_regions:2*n_regions]
    R = y[2*n_regions:]

    # 各コンパートメントの変化率を格納する配列
    dSdt = np.zeros(n_regions)
```

```

dIdt = np.zeros(n_regions)
dRdt = np.zeros(n_regions)

# 各地域における感染ダイナミクス
for i in range(n_regions):
    # 地域内の感染ダイナミクス
    dSdt[i] = -beta * S[i] * I[i] / N[i]
    dIdt[i] = beta * S[i] * I[i] / N[i] - gamma * I[i]
    dRdt[i] = gamma * I[i]

# 地域間の人口移動
for i, j in G.edges():
    # エッジの重みを取得 (デフォルトは1.0)
    weight = G[i][j].get('weight', 1.0)

    # 地域i->jおよびj->iへの人の流れ
    flow_rate = mobility * weight

    # 人口移動による各コンパートメントの変化率の調整
    # 感受性者の移動
    dSdt[i] -= flow_rate * S[i] / N[i]
    dSdt[j] += flow_rate * S[i] / N[i]

    dSdt[j] -= flow_rate * S[j] / N[j]
    dSdt[i] += flow_rate * S[j] / N[j]

    # 感染者の移動
    dIdt[i] -= flow_rate * I[i] / N[i]
    dIdt[j] += flow_rate * I[i] / N[i]

    dIdt[j] -= flow_rate * I[j] / N[j]
    dIdt[i] += flow_rate * I[j] / N[j]

    # 回復者の移動
    dRdt[i] -= flow_rate * R[i] / N[i]
    dRdt[j] += flow_rate * R[i] / N[i]

    dRdt[j] -= flow_rate * R[j] / N[j]
    dRdt[i] += flow_rate * R[j] / N[j]

return np.concatenate([dSdt, dIdt, dRdt])

def simulate_metapopulation_sir(G, region_populations, beta, gamma,
initial_infected, mobility, days):
    """メタポピュレーションSIRモデルのシミュレーションを実行する関数"""

n_regions = len(region_populations)

# 初期条件の設定
S0 = np.array(region_populations) - np.array(initial_infected)
I0 = np.array(initial_infected)
R0 = np.zeros(n_regions)

y0 = np.concatenate([S0, I0, R0])

```

```
# モデルパラメータの設定
params = {
    'network': G,
    'n_regions': n_regions,
    'beta': beta,
    'gamma': gamma,
    'N': np.array(region_populations),
    'mobility': mobility
}

# 時間軸
t = np.linspace(0, days, days + 1)

# 微分方程式を解く
solution = spi.solve_ivp(
    fun=lambda t, y: metapopulation_sir_model(y, t, params),
    t_span=[t[0], t[-1]],
    y0=y0,
    t_eval=t
)

# 結果の整形
S = np.reshape(solution.y[:n_regions], (n_regions, -1))
I = np.reshape(solution.y[n_regions:2*n_regions], (n_regions, -1))
R = np.reshape(solution.y[2*n_regions:], (n_regions, -1))

# 日次新規感染者数の計算
new_cases = np.zeros((n_regions, len(t)))
for i in range(n_regions):
    new_cases[i, 1:] = -(S[i, 1:] - S[i, :-1])

results = {
    't': t,
    'S': S,
    'I': I,
    'R': R,
    'new_cases': new_cases,
    'region_populations': region_populations
}

return results

def plot_metapopulation_results(results, G):
    """メタポピュレーションSIRモデルの結果を可視化する関数"""

    t = results['t']
    I = results['I']
    new_cases = results['new_cases']
    region_populations = results['region_populations']

    n_regions = len(region_populations)

    # 時間経過に伴う感染者数の空間分布
```

```
time_points = [0, 30, 60, 90, 120]
fig, axes = plt.subplots(1, len(time_points), figsize=(20, 5))

# ノードの位置情報
if all('pos' in G.nodes[node] for node in G.nodes):
    pos = nx.get_node_attributes(G, 'pos')
else:
    pos = nx.spring_layout(G, seed=42)

for i, day in enumerate(time_points):
    if day < len(t):
        # 感染率（人口に対する感染者の比率）を計算
        infection_rates = [I[region, day] / region_populations[region] * 100
for region in range(n_regions)]

    ax = axes[i]

    # エッジの描画
    if 'weight' in G.edges[list(G.edges())[0]]:
        weights = [G[u][v]['weight'] * 3 for u, v in G.edges()]
        nx.draw_networkx_edges(G, pos, width=weights, alpha=0.3,
edge_color='gray', ax=ax)
    else:
        nx.draw_networkx_edges(G, pos, width=1, alpha=0.3,
edge_color='gray', ax=ax)

    # ノードの描画（サイズと色は感染率に基づく）
    nodes = nx.draw_networkx_nodes(G, pos, node_color=infection_rates,
                                    cmap=plt.cm.YlOrRd,
                                    node_size=[max(300, rate * 30) for rate
in infection_rates],
                                    ax=ax)

    ax.set_title(f'Day {day}', fontsize=12)
    ax.axis('off')

plt.colorbar(nodes, ax=axes, label='感染率 (%)')
fig.suptitle('時間経過に伴う感染拡大の空間分布', fontsize=15)
plt.tight_layout()
plt.show()

# 地域別の感染曲線
plt.figure(figsize=(15, 10))

# カラーマップの設定
colors = plt.cm.viridis(np.linspace(0, 1, n_regions))

plt.subplot(2, 1, 1)
for i in range(n_regions):
    plt.plot(t, I[i], color=colors[i], label=f'Region {i}', linewidth=2)

plt.title('地域別の感染者数の推移', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('感染者数')
```

```
plt.grid(True, linestyle='--', alpha=0.7)

if n_regions <= 10: # 地域数が多すぎる場合は凡例を省略
    plt.legend(title='地域')

plt.subplot(2, 1, 2)
for i in range(n_regions):
    plt.plot(t, I[i] / region_populations[i] * 100, color=colors[i],
label=f'Region {i}', linewidth=2)

plt.title('地域別の人口に占める感染率の推移', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('感染率 (%)')
plt.grid(True, linestyle='--', alpha=0.7)

if n_regions <= 10:
    plt.legend(title='地域')

plt.tight_layout()
plt.show()

# ヒートマップによる可視化
plt.figure(figsize=(15, 8))

plt.subplot(1, 2, 1)
plt.pcolormesh(t, np.arange(n_regions), I, cmap='YlOrRd', shading='auto')
plt.colorbar(label='感染者数')
plt.title('地域別・感染者数の推移', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('地域')

plt.subplot(1, 2, 2)
infection_rates = I / np.array(region_populations)[:, np.newaxis] * 100
plt.pcolormesh(t, np.arange(n_regions), infection_rates, cmap='YlOrRd',
shading='auto')
plt.colorbar(label='感染率 (%)')
plt.title('地域別・感染率の推移', fontsize=15)
plt.xlabel('経過日数')
plt.ylabel('地域')

plt.tight_layout()
plt.show()

# メタポピュレーションSIRモデルの実行
n_regions = 20 # 地域数

# 地域間ネットワークの作成
G = create_spatial_network(n_regions, network_type='small_world', params={'k': 4,
'p': 0.2})

# ネットワークの可視化
plot_spatial_network(G, title='地域間移動ネットワーク')

# 地域別人口の設定 (ランダムな分布)
```

```

np.random.seed(42)
total_population = 1000000
region_populations = np.random.poisson(total_population / n_regions, n_regions)

# シミュレーションパラメータ
beta = 0.3          # 感染率
gamma = 0.1          # 回復率
mobility = 0.05      # 地域間移動率

# 初期感染者（特定の1地域でのみ感染が始まる）
initial_region = 0
initial_infected = np.zeros(n_regions, dtype=int)
initial_infected[initial_region] = 20

# シミュレーション実行
metapop_results = simulate_metapopulation_sir(
    G,
    region_populations,
    beta,
    gamma,
    initial_infected,
    mobility,
    days=180
)

# 結果の可視化
plot_metapopulation_results(metapop_results, G)

```

💡 **初心者向けヒント**： メタポピュレーションモデルは、感染症の地理的な拡散パターンを理解し、地域間の移動制限などの介入効果を評価するのに役立ちます。特に都市間の交通量データと組み合わせることで、より現実的なシミュレーションが可能になります。

## 9. 確率論的モデルの導入

決定論的モデルでは捉えられない確率的な変動を考慮するため、確率論的モデルが重要です。特に小さな集団や流行初期段階では確率的変動が大きな影響を与えます。

### 確率論的モデルの基礎

```

def stochastic_sir_model(S0, I0, R0, beta, gamma, T, num_simulations=100):
    """確率論的SIRモデルのシミュレーション関数"""
    N = S0 + I0 + R0  # 総人口

    # 結果を格納する配列
    all_results = []

    for sim in tqdm(range(num_simulations)):
        # 各コンパートメントの初期値
        S = [S0]
        I = [I0]

```

```
R = [R0]

# 現在の状態
current_S = S0
current_I = I0
current_R = R0

# 時間ステップ
dt = 0.1 # 小さな時間ステップ
t_steps = int(T / dt)

for t in range(t_steps):
    # 感染と回復の確率を計算
    p_infection = beta * current_S * current_I / N * dt
    p_recovery = gamma * current_I * dt

    # 感染と回復のイベント数を確率的に決定
    infection_events = np.random.binomial(current_S, p_infection / current_S) if current_S > 0 else 0
    recovery_events = np.random.binomial(current_I, p_recovery / current_I) if current_I > 0 else 0

    # 状態の更新
    current_S -= infection_events
    current_I += infection_events - recovery_events
    current_R += recovery_events

    # 整数値に丸める
    current_S = max(0, int(current_S))
    current_I = max(0, int(current_I))
    current_R = max(0, int(current_R))

    # 1日ごとにデータを記録 (dt=0.1の場合、10ステップごと)
    if t % int(1/dt) == 0:
        S.append(current_S)
        I.append(current_I)
        R.append(current_R)

    # 結果の保存
    all_results.append({
        'S': np.array(S),
        'I': np.array(I),
        'R': np.array(R),
        'sim_id': sim
    })

return all_results

# モデルパラメータ
total_population = 10000
initial_infected = 10
initial_recovered = 0
initial_susceptible = total_population - initial_infected - initial_recovered
beta = 0.3 # 感染率
```

```
gamma = 0.1 # 回復率
T = 150 # シミュレーション期間(日)
num_simulations = 30 # シミュレーション回数

# 確率論的SIRモデルの実行
stochastic_results = stochastic_sir_model(
    S0=initial_susceptible,
    I0=initial_infected,
    R0=initial_recovered,
    beta=beta,
    gamma=gamma,
    T=T,
    num_simulations=num_simulations
)

# 結果の可視化
plt.figure(figsize=(14, 8))

# 各シミュレーションの結果をプロット
for sim_result in stochastic_results:
    plt.plot(range(len(sim_result['I'])), sim_result['I'], 'lightblue', alpha=0.5)

# 全シミュレーションの平均値
mean_I = np.mean([sim_result['I'] for sim_result in stochastic_results], axis=0)
plt.plot(range(len(mean_I)), mean_I, 'b-', linewidth=2, label='平均値')

# 決定論的SIRモデルの結果も比較用に計算
def deterministic_sir(y, t, N, beta, gamma):
    S, I, R = y
    dSdt = -beta * S * I / N
    dIdt = beta * S * I / N - gamma * I
    dRdt = gamma * I
    return [dSdt, dIdt, dRdt]

t = np.linspace(0, T, T+1)
y0 = [initial_susceptible, initial_infected, initial_recovered]
result = spi.odeint(deterministic_sir, y0, t, args=(total_population, beta, gamma))
plt.plot(t, result[:, 1], 'r--', linewidth=2, label='決定論的モデル')

plt.title('確率論的SIRモデルによる感染者数の予測', fontsize=15)
plt.xlabel('時間(日)')
plt.ylabel('感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
plt.tight_layout()
plt.show()
```

## 確率論的モデルの分析

```
def analyze_stochastic_results(stochastic_results, confidence_level=0.95):
    """確率論的シミュレーション結果の統計分析を行う関数"""

    # 感染者数の時系列を全シミュレーションから抽出
    all_I = np.array([sim_result['I'] for sim_result in stochastic_results])

    # 平均値
    mean_I = np.mean(all_I, axis=0)

    # 標準偏差
    std_I = np.std(all_I, axis=0)

    # 信頼区間の計算
    alpha = 1 - confidence_level
    lower_percentile = alpha / 2 * 100
    upper_percentile = (1 - alpha / 2) * 100

    lower_bound = np.percentile(all_I, lower_percentile, axis=0)
    upper_bound = np.percentile(all_I, upper_percentile, axis=0)

    # 最大感染者数の分布
    peak_values = np.max(all_I, axis=1)
    peak_times = np.argmax(all_I, axis=1)

    # 結果の可視化
    plt.figure(figsize=(18, 12))

    # サブプロット1: 信頼区間付きの感染者数推移
    plt.subplot(2, 2, 1)
    t = np.arange(len(mean_I))
    plt.plot(t, mean_I, 'b-', linewidth=2, label='平均値')
    plt.fill_between(t, lower_bound, upper_bound, color='blue', alpha=0.2,
                     label=f'{confidence_level*100:.0f}%信頼区間')
    plt.title('確率論的シミュレーションの感染者数推移', fontsize=15)
    plt.xlabel('時間（日）')
    plt.ylabel('感染者数')
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.legend()

    # サブプロット2: 変動係数の推移
    plt.subplot(2, 2, 2)
    cv = std_I / mean_I
    cv[np.isnan(cv)] = 0 # 0除算エラーの処理
    plt.plot(t, cv, 'g-', linewidth=2)
    plt.title('感染者数の変動係数の推移', fontsize=15)
    plt.xlabel('時間（日）')
    plt.ylabel('変動係数 (CV)')
    plt.grid(True, linestyle='--', alpha=0.7)

    # サブプロット3: 最大感染者数のヒストグラム
    plt.subplot(2, 2, 3)
    plt.hist(peak_values, bins=15, color='orange', alpha=0.7)
    plt.axvline(np.mean(peak_values), color='r', linestyle='--',
```

```

        label=f'平均: {np.mean(peak_values):.0f}')
plt.title('最大感染者数の分布', fontsize=15)
plt.xlabel('最大感染者数')
plt.ylabel('頻度')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# サブプロット4: ピーク到達時間のヒストグラム
plt.subplot(2, 2, 4)
plt.hist(peak_times, bins=15, color='purple', alpha=0.7)
plt.axvline(np.mean(peak_times), color='r', linestyle='--',
            label=f'平均: {np.mean(peak_times):.1f}日')
plt.title('ピーク到達時間の分布', fontsize=15)
plt.xlabel('ピーク到達日')
plt.ylabel('頻度')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

plt.tight_layout()
plt.show()

# 結果の要約統計量を辞書で返す
summary_stats = {
    'mean_peak_value': np.mean(peak_values),
    'std_peak_value': np.std(peak_values),
    'mean_peak_time': np.mean(peak_times),
    'std_peak_time': np.std(peak_times),
    'extinction_probability': np.mean(np.max(all_I, axis=1) <
initial_infected)
}

return summary_stats

# 分析の実行
stats = analyze_stochastic_results(stochastic_results, confidence_level=0.95)

# 結果の表示
print("確率論的SIRモデルの統計分析結果:")
print(f"平均最大感染者数: {stats['mean_peak_value']:.0f} ± {stats['std_peak_value']:.0f}")
print(f"平均ピーク到達時間: {stats['mean_peak_time']:.1f} ± {stats['std_peak_time']:.1f} 日")
print(f"初期段階で流行が消滅する確率: {stats['extinction_probability']*100:.2f}%")

```

## 確率微分方程式(SDE)モデル

```

def sde_sir_model(S0, I0, R0, beta, gamma, T, sigma_I=0.1, num_simulations=100):
    """確率微分方程式を用いたSIRモデルの実装"""

    N = S0 + I0 + R0 # 総人口
    dt = 0.01 # 時間ステップ

```

```
t_steps = int(T / dt)
t_record = np.arange(0, T+1)
record_steps = [int(t / dt) for t in t_record]

all_results = []

for sim in tqdm(range(num_simulations)):
    # 各コンパートメントの初期値
    S = np.zeros(len(t_record))
    I = np.zeros(len(t_record))
    R = np.zeros(len(t_record))

    S[0] = S0
    I[0] = I0
    R[0] = R0

    # 現在の状態
    current_S = S0
    current_I = I0
    current_R = R0

    record_idx = 1

    for t in range(1, t_steps + 1):
        # 決定論的項
        dS_det = -beta * current_S * current_I / N * dt
        dI_det = (beta * current_S * current_I / N - gamma * current_I) * dt
        dR_det = gamma * current_I * dt

        # 確率論的項（感染者数にのみノイズを加える）
        dw = np.random.normal(0, np.sqrt(dt))
        dI_stoch = sigma_I * current_I * dw

        # 状態の更新
        current_S += dS_det
        current_I += dI_det + dI_stoch
        current_R += dR_det

        # 物理的制約を適用（負の値や総人口を超える値を防ぐ）
        current_S = max(0, min(current_S, N))
        current_I = max(0, min(current_I, N))
        current_R = max(0, min(current_R, N))

        # 総人口を保存するように調整
        total = current_S + current_I + current_R
        if total != N:
            scale_factor = N / total
            current_S *= scale_factor
            current_I *= scale_factor
            current_R *= scale_factor

        # 記録ステップでデータを保存
        if t in record_steps:
            S[record_idx] = current_S
```

```

        I[record_idx] = current_I
        R[record_idx] = current_R
        record_idx += 1

    all_results.append({
        'S': S,
        'I': I,
        'R': R,
        'sim_id': sim
    })

    return all_results, t_record

# SDEモデルのパラメータ
sigma_I = 0.1 # 確率的変動の大きさ

# SDEベースのSIRモデルの実行
sde_results, t_sde = sde_sir_model(
    S0=initial_susceptible,
    I0=initial_infected,
    R0=initial_recovered,
    beta=beta,
    gamma=gamma,
    T=T,
    sigma_I=sigma_I,
    num_simulations=num_simulations
)

# 結果の可視化
plt.figure(figsize=(14, 8))

# 各シミュレーションの結果をプロット
for sim_result in sde_results:
    plt.plot(t_sde, sim_result['I'], 'lightgreen', alpha=0.3)

# 全シミュレーションの平均値
mean_I_sde = np.mean([sim_result['I'] for sim_result in sde_results], axis=0)
plt.plot(t_sde, mean_I_sde, 'g-', linewidth=2, label='SDEモデル平均')

# 決定論的モデルとの比較
plt.plot(t, result[:, 1], 'r--', linewidth=2, label='決定論的モデル')

plt.title('確率微分方程式(SDE)によるSIRモデルシミュレーション', fontsize=15)
plt.xlabel('時間(日)')
plt.ylabel('感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
plt.tight_layout()
plt.show()

```

⌚ 初心者向けヒント： 確率論的モデルは、特に小規模集団や感染初期段階での予測に重要です。確率的変動を考慮することで、最悪のシナリオや流行が自然消滅する可能性なども評価できます。

## 10. パラメータ推定

感染症モデルの精度は、適切なパラメータ推定に大きく依存します。実データからモデルパラメータを推定するさまざまな手法を学びましょう。

### 最小二乗法によるパラメータ推定

```
def estimate_params_least_squares(data, initial_guess, population_size):
    """最小二乗法によるSIRモデルのパラメータ推定"""

    # 観測データ
    observed_cases = data['new_cases_smooth'].values
    cumulative_cases = data['cumulative_cases'].values
    times = np.arange(len(observed_cases))

    # 最適化する目的関数
    def objective_function(params):
        beta, gamma = params
        # 初期状態 (S0, I0, R0) の設定
        I0 = observed_cases[0]
        R0 = 0
        S0 = population_size - I0 - R0

        # SIRモデルの実行
        def sir_model(y, t, N, beta, gamma):
            S, I, R = y
            dSdt = -beta * S * I / N
            dIdt = beta * S * I / N - gamma * I
            dRdt = gamma * I
            return [dSdt, dIdt, dRdt]

        y0 = [S0, I0, R0]
        result = spi.odeint(sir_model, y0, times, args=(population_size, beta,
                                                       gamma))

        # 予測された感染者数
        predicted_I = result[:, 1]

        # 二乗誤差
        mse = np.mean((predicted_I - observed_cases) ** 2)
        return mse

    # 最適化
    bounds = [(0, 2), (0, 1)] # beta と gamma の範囲
    result = spi.minimize(objective_function, initial_guess, bounds=bounds,
                          method='L-BFGS-B')

    # 最適化されたパラメータ
    estimated_beta, estimated_gamma = result.x

    # 最適パラメータによるモデル予測
```

```
def sir_model(y, t, N, beta, gamma):
    S, I, R = y
    dSdt = -beta * S * I / N
    dIdt = beta * S * I / N - gamma * I
    dRdt = gamma * I
    return [dSdt, dIdt, dRdt]

I0 = observed_cases[0]
R0 = 0
S0 = population_size - I0 - R0

y0 = [S0, I0, R0]
fitted_result = spi.odeint(sir_model, y0, times, args=(population_size,
estimated_beta, estimated_gamma))

# 結果の可視化
plt.figure(figsize=(14, 8))

plt.plot(times, observed_cases, 'o', markersize=4, alpha=0.6, label='観測データ')
plt.plot(times, fitted_result[:, 1], 'r-', linewidth=2, label='最適化モデル')

plt.title('最小二乗法によるSIRモデルのパラメータ推定', fontsize=15)
plt.xlabel('時間(日)')
plt.ylabel('感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
plt.tight_layout()
plt.show()

# 基本再生産数の計算
R0_value = estimated_beta / estimated_gamma

# 結果のまとめ
results = {
    'beta': estimated_beta,
    'gamma': estimated_gamma,
    'R0': R0_value,
    'infectious_period': 1 / estimated_gamma,
    'fitted_values': fitted_result,
    'rmse': np.sqrt(result.fun)
}

return results

# パラメータ推定の実行
population_size = 10000 # 仮の人口規模
initial_guess = [0.3, 0.1] # beta と gamma の初期推測値

estimation_results = estimate_params_least_squares(
    data=processed_data,
    initial_guess=initial_guess,
    population_size=population_size
)
```

```
# 結果の表示
print("最小二乗法によるパラメータ推定結果:")
print(f"推定された感染率 (β): {estimation_results['beta']:.4f}")
print(f"推定された回復率 (γ): {estimation_results['gamma']:.4f}")
print(f"基本再生産数 (R₀): {estimation_results['R0']:.2f}")
print(f"平均感染期間: {estimation_results['infectious_period']:.1f} 日")
print(f"RMSE: {estimation_results['rmse']:.2f}")
```

## ベイズ推定によるパラメータ推定

```
def estimate_params_bayesian(data, population_size, num_samples=1000):
    """Pyroを用いたベイズ推定によるSIRモデルのパラメータ推定"""
    import pyro
    import pyro.distributions as dist
    from pyro.infer import MCMC, NUTS
    import torch

    # データの準備
    observed_cases = torch.tensor(data['new_cases_smooth'].values,
                                    dtype=torch.float)
    times = torch.arange(len(observed_cases), dtype=torch.float)

    # SIRモデル (PyTorch実装)
    def sir_model(S0, I0, R0, beta, gamma, times):
        N = S0 + I0 + R0

        S = torch.zeros(len(times))
        I = torch.zeros(len(times))
        R = torch.zeros(len(times))

        S[0] = S0
        I[0] = I0
        R[0] = R0

        for t in range(1, len(times)):
            dS = -beta * S[t-1] * I[t-1] / N
            dI = beta * S[t-1] * I[t-1] / N - gamma * I[t-1]
            dR = gamma * I[t-1]

            S[t] = S[t-1] + dS
            I[t] = I[t-1] + dI
            R[t] = R[t-1] + dR

        return S, I, R

    # ベイズモデルの定義
    def sir_model_pyro():
        # 事前分布
        beta = pyro.sample('beta', dist.Uniform(0.01, 1.0))
        gamma = pyro.sample('gamma', dist.Uniform(0.01, 0.5))
```

```
sigma = pyro.sample('sigma', dist.Uniform(0.1, 100.0))

# 初期値
I0 = observed_cases[0]
R0 = 0.0
S0 = population_size - I0 - R0

# モデルの実行
_, I_pred, _ = sir_model(S0, I0, R0, beta, gamma, times)

# 観測モデル（正規分布）
with pyro.plate('data', len(observed_cases)):
    pyro.sample('obs', dist.Normal(I_pred, sigma), obs=observed_cases)

# MCMCの実行
pyro.clear_param_store()
nuts_kernel = NUTS(sir_model_pyro)
mcmc = MCMC(nuts_kernel, num_samples=num_samples, warmup_steps=200)
mcmc.run()

# サンプルの取得
samples = mcmc.get_samples()

# 結果の可視化
plt.figure(figsize=(18, 12))

# サブプロット1: ベータの事後分布
plt.subplot(2, 3, 1)
plt.hist(samples['beta'].numpy(), bins=30, alpha=0.7, color='blue')
plt.axvline(samples['beta'].mean().item(), color='r', linestyle='--',
            label=f'平均: {samples["beta"].mean().item():.4f}')
plt.title('感染率(β)の事後分布', fontsize=15)
plt.xlabel('β')
plt.ylabel('頻度')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# サブプロット2: ガンマの事後分布
plt.subplot(2, 3, 2)
plt.hist(samples['gamma'].numpy(), bins=30, alpha=0.7, color='green')
plt.axvline(samples['gamma'].mean().item(), color='r', linestyle='--',
            label=f'平均: {samples["gamma"].mean().item():.4f}')
plt.title('回復率(γ)の事後分布', fontsize=15)
plt.xlabel('γ')
plt.ylabel('頻度')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# サブプロット3: R0の事後分布
r0_samples = samples['beta'] / samples['gamma']
plt.subplot(2, 3, 3)
plt.hist(r0_samples.numpy(), bins=30, alpha=0.7, color='orange')
plt.axvline(r0_samples.mean().item(), color='r', linestyle='--',
            label=f'平均: {r0_samples.mean().item():.2f}'')
```

```
plt.title('基本再生産数( $R_0$ )の事後分布', fontsize=15)
plt.xlabel('R0')
plt.ylabel('頻度')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# サブプロット4: 事後予測確認
plt.subplot(2, 1, 2)

# 事後分布からのサンプルを使用してモデル予測を生成
num_pred_samples = 100
pred_indices = np.random.choice(len(samples['beta']), num_pred_samples,
replace=False)

I0 = observed_cases[0].item()
R0 = 0
S0 = population_size - I0 - R0

# 各事後サンプルに基づく予測曲線をプロット
for idx in pred_indices:
    beta_sample = samples['beta'][idx].item()
    gamma_sample = samples['gamma'][idx].item()

    _, I_pred, _ = sir_model(S0, I0, R0, beta_sample, gamma_sample, times)
    plt.plot(times.numpy(), I_pred.numpy(), 'b-', alpha=0.1)

# 観測データをプロット
plt.plot(times.numpy(), observed_cases.numpy(), 'ro', markersize=4, label='観測データ')

# 平均予測曲線
beta_mean = samples['beta'].mean().item()
gamma_mean = samples['gamma'].mean().item()
_, I_mean, _ = sir_model(S0, I0, R0, beta_mean, gamma_mean, times)
plt.plot(times.numpy(), I_mean.numpy(), 'g-', linewidth=2, label='平均予測')

plt.title('ベイズ推定によるSIRモデルの事後予測', fontsize=15)
plt.xlabel('時間(日)')
plt.ylabel('感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

plt.tight_layout()
plt.show()

# 結果のまとめ
results = {
    'beta_mean': samples['beta'].mean().item(),
    'beta_std': samples['beta'].std().item(),
    'gamma_mean': samples['gamma'].mean().item(),
    'gamma_std': samples['gamma'].std().item(),
    'R0_mean': r0_samples.mean().item(),
    'R0_std': r0_samples.std().item(),
    'infectious_period': 1 / samples['gamma'].mean().item(),
```

```

        'samples': samples
    }

    return results

# ベイズ推定の実行（注：PyroとTorchが必要）
try:
    bayesian_results = estimate_params_bayesian(
        data=processed_data,
        population_size=population_size,
        num_samples=1000
    )

    # 結果の表示
    print("\nベイズ推定によるパラメータ推定結果:")
    print(f"推定された感染率 (β): {bayesian_results['beta_mean']:.4f} ± {bayesian_results['beta_std']:.4f}")
    print(f"推定された回復率 (γ): {bayesian_results['gamma_mean']:.4f} ± {bayesian_results['gamma_std']:.4f}")
    print(f"基本再生産数 (R₀): {bayesian_results['R0_mean']:.2f} ± {bayesian_results['R0_std']:.2f}")
    print(f"平均感染期間: {bayesian_results['infectious_period']:.1f} 日")

except ImportError:
    print("Pyro または PyTorch がインストールされていません。")
    print("以下のコマンドでインストールできます:")
    print("!pip install pyro-ppl torch")

```

## 時変パラメータの推定

```

def estimate_time_varying_params(data, window_size=14, population_size=10000):
    """時間変動するパラメータの推定を行う関数"""

    # データの準備
    observed_cases = data['new_cases_smooth'].values
    dates = data['date'].values
    times = np.arange(len(observed_cases))

    # 結果を格納する配列
    beta_values = []
    gamma_values = []
    r0_values = []
    dates_list = []

    # ウィンドウをスライドさせながら各時点でのパラメータを推定
    for start_idx in range(0, len(observed_cases) - window_size, 5):  # 5日ごとに推定
        end_idx = start_idx + window_size

        # ウィンドウ内のデータ
        window_cases = observed_cases[start_idx:end_idx]

```

```
window_times = times[start_idx:end_idx] - times[start_idx] # 相対時間に変換

# 最適化する目的関数
def objective_function(params):
    beta, gamma = params
    # 初期状態 (S0, I0, R0) の設定
    I0 = window_cases[0]
    R0 = 0
    S0 = population_size - I0 - R0

    # SIRモデルの実行
    def sir_model(y, t, N, beta, gamma):
        S, I, R = y
        dSdt = -beta * S * I / N
        dIdt = beta * S * I / N - gamma * I
        dRdt = gamma * I
        return [dSdt, dIdt, dRdt]

    y0 = [S0, I0, R0]
    result = spi.odeint(sir_model, y0, window_times, args=(population_size, beta, gamma))

    # 予測された感染者数
    predicted_I = result[:, 1]

    # 二乗誤差
    mse = np.mean((predicted_I - window_cases) ** 2)
    return mse

# 最適化
bounds = [(0, 2), (0, 1)] # beta と gamma の範囲
initial_guess = [0.3, 0.1] # beta と gamma の初期推測値

try:
    result = spi.minimize(objective_function, initial_guess,
                           bounds=bounds, method='L-BFGS-B')

    # 推定されたパラメータ
    estimated_beta, estimated_gamma = result.x
    estimated_r0 = estimated_beta / estimated_gamma

    # 結果を保存
    beta_values.append(estimated_beta)
    gamma_values.append(estimated_gamma)
    r0_values.append(estimated_r0)
    dates_list.append(dates[start_idx + window_size // 2]) # ウィンドウの中央の日付

except:
    print(f"ウィンドウ {start_idx}-{end_idx} でのパラメータ推定に失敗しました。")

# 結果の可視化
plt.figure(figsize=(16, 12))
```

```
# サブプロット1: 時変感染率
plt.subplot(3, 1, 1)
plt.plot(dates_list, beta_values, 'b-o', markersize=4)
plt.title('時間変動する感染率 ( $\beta$ )', fontsize=15)
plt.ylabel('β')
plt.grid(True, linestyle='--', alpha=0.7)

# サブプロット2: 時変回復率
plt.subplot(3, 1, 2)
plt.plot(dates_list, gamma_values, 'g-o', markersize=4)
plt.title('時間変動する回復率 ( $\gamma$ )', fontsize=15)
plt.ylabel('γ')
plt.grid(True, linestyle='--', alpha=0.7)

# サブプロット3: 時変基本再生産数
plt.subplot(3, 1, 3)
plt.plot(dates_list, r0_values, 'r-o', markersize=4)
plt.axhline(y=1, color='k', linestyle='--', label=' $R_0=1$  (閾値)')
plt.title('時間変動する基本再生産数 ( $R_0$ )', fontsize=15)
plt.xlabel('日付')
plt.ylabel('R₀')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

plt.tight_layout()
plt.show()

# 結果のデータフレームを返す
time_varying_params = pd.DataFrame({
    'date': dates_list,
    'beta': beta_values,
    'gamma': gamma_values,
    'R0': r0_values
})

return time_varying_params

# 時間変動パラメータの推定を実行
time_varying_results = estimate_time_varying_params(
    data=processed_data,
    window_size=14,
    population_size=population_size
)

# インターベンション効果の分析
intervention_dates = pd.to_datetime(['2023-02-15', '2023-03-10']) # 仮想的な介入日

plt.figure(figsize=(14, 6))
plt.plot(time_varying_results['date'], time_varying_results['R0'], 'r-o',
markersize=4)
plt.axhline(y=1, color='k', linestyle='--', label=' $R_0=1$  (閾値)')

# 介入時点を表示
```

```
for date in intervention_dates:
    plt.axvline(x=date, color='blue', linestyle='--', alpha=0.7)
    plt.text(date, plt.ylim()[1]*0.9, '介入', rotation=90,
verticalalignment='top')

plt.title('介入前後の実効再生産数の変化', fontsize=15)
plt.xlabel('日付')
plt.ylabel('実効再生産数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
plt.tight_layout()
plt.show()
```

💡 **初心者向けヒント**：パラメータ推定は感染症モデリングの中でも最も重要かつ難しい部分です。データの質と量、モデルの複雑さ、使用するアルゴリズムによって推定精度が大きく変わります。時変パラメータの推定は、政策介入効果の評価や流行の異なるフェーズの分析に特に有用です。

## 11. 機械学習を用いた予測

機械学習を活用することで、複雑なパターンを捉えた予測モデルを構築できます。ここでは機械学習を感染症予測に応用する方法を学びます。

### 時系列予測モデル

```
def train_prophet_model(data, forecast_periods=30):
    """Prophet モデルを用いた感染症時系列予測"""
    try:
        from prophet import Prophet

        # Prophet用にデータを整形
        df_prophet = pd.DataFrame({
            'ds': data['date'],
            'y': data['new_cases_smooth']
        })

        # モデルの定義とフィッティング
        model = Prophet(
            yearly_seasonality=False,
            weekly_seasonality=True,
            daily_seasonality=False,
            changepoint_prior_scale=0.05
        )
        model.fit(df_prophet)

        # 将来予測用のデータフレーム
        future = model.make_future_dataframe(periods=forecast_periods)

        # 予測の実行
        forecast = model.predict(future)
```

```
# 結果の可視化
plt.figure(figsize=(14, 8))

# 学習データと予測結果のプロット
train_dates = df_prophet['ds'].values
test_dates = forecast['ds'].values[-forecast_periods:]

plt.plot(train_dates, df_prophet['y'], 'b.', alpha=0.6, label='学習データ')
plt.plot(forecast['ds'], forecast['yhat'], 'r-', linewidth=2, label='予測
    値')
plt.fill_between(forecast['ds'],
                 forecast['yhat_lower'],
                 forecast['yhat_upper'],
                 color='red',
                 alpha=0.2,
                 label='95%信頼区間')

# 学習データと予測データの境界線
plt.axvline(x=train_dates[-1], color='gray', linestyle='--', alpha=0.7)
plt.text(train_dates[-1], plt.ylim()[1]*0.9, '予測開始', rotation=90,
verticalalignment='top')

plt.title('Prophet による感染症時系列予測', fontsize=15)
plt.xlabel('日付')
plt.ylabel('新規感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
plt.tight_layout()
plt.show()

# モデルのコンポーネント分解
plt.figure(figsize=(12, 10))
model.plot_components(forecast)
plt.tight_layout()
plt.show()

return model, forecast

except ImportError:
    print("Prophet がインストールされていません。")
    print("以下のコマンドでインストールできます:")
    print("!pip install prophet")
    return None, None

# Prophet モデルの学習と予測
prophet_model, prophet_forecast = train_prophet_model(processed_data,
forecast_periods=30)
```

## LSTM による感染症予測

```
def prepare_lstm_data(data, n_lag=14, n_ahead=7, test_size=0.2):
    """LSTM モデル用にデータを準備する関数"""
    try:
        import tensorflow as tf
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import LSTM, Dense, Dropout
        from sklearn.preprocessing import MinMaxScaler
        from sklearn.model_selection import train_test_split

        # データの正規化
        scaler = MinMaxScaler(feature_range=(0, 1))
        scaled_cases =
scaler.fit_transform(data['new_cases_smooth'].values.reshape(-1, 1))

        # ラグ特徴量とターゲットの作成
        X, y = [], []
        for i in range(len(scaled_cases) - n_lag - n_ahead + 1):
            X.append(scaled_cases[i:(i + n_lag), 0])
            y.append(scaled_cases[(i + n_lag):(i + n_lag + n_ahead), 0])

        X, y = np.array(X), np.array(y)

        # トレーニングデータとテストデータに分割
        X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=test_size, shuffle=False)

        # LSTM用にデータを3次元に変換
        X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
        X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))

        return X_train, X_test, y_train, y_test, scaler

    except ImportError:
        print("TensorFlow がインストールされていません。")
        print("以下のコマンドでインストールできます:")
        print("!pip install tensorflow")
        return None, None, None, None, None

def build_and_train_lstm(X_train, y_train, X_test, y_test, epochs=50,
batch_size=16):
    """LSTM モデルの構築と学習を行う関数"""
    try:
        import tensorflow as tf
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import LSTM, Dense, Dropout

        # モデルの構築
        model = Sequential()
        model.add(LSTM(50, return_sequences=True, input_shape=(X_train.shape[1],
1)))
        model.add(Dropout(0.2))
        model.add(LSTM(50))
        model.add(Dropout(0.2))
```

```
model.add(Dense(y_train.shape[1]))  
  
# モデルのコンパイル  
model.compile(optimizer='adam', loss='mse')  
  
# モデルの学習  
history = model.fit(  
    X_train, y_train,  
    epochs=epochs,  
    batch_size=batch_size,  
    validation_data=(X_test, y_test),  
    verbose=1  
)  
  
# 学習曲線の可視化  
plt.figure(figsize=(10, 6))  
plt.plot(history.history['loss'], label='Train Loss')  
plt.plot(history.history['val_loss'], label='Validation Loss')  
plt.title('LSTM モデルの学習曲線', fontsize=15)  
plt.xlabel('Epoch')  
plt.ylabel('Loss')  
plt.legend()  
plt.grid(True, linestyle='--', alpha=0.7)  
plt.tight_layout()  
plt.show()  
  
return model, history  
  
except ImportError:  
    print("TensorFlow がインストールされていません。")  
    return None, None  
  
def evaluate_lstm_forecast(model, X_test, y_test, scaler, data, n_lag, n_ahead):  
    """LSTM モデルによる予測の評価を行う関数"""  
    try:  
        # テストデータの予測  
        y_pred = model.predict(X_test)  
  
        # スケールを元に戻す  
        y_test_rescaled = scaler.inverse_transform(y_test)  
        y_pred_rescaled = scaler.inverse_transform(y_pred)  
  
        # 評価メトリクス  
        rmse = np.sqrt(mean_squared_error(y_test_rescaled, y_pred_rescaled))  
        r2 = r2_score(y_test_rescaled.reshape(-1), y_pred_rescaled.reshape(-1))  
  
        # 予測結果の可視化  
        plt.figure(figsize=(14, 8))  
  
        # 最後のテストデータの実際の値と予測値をプロット  
        last_test_idx = len(data) - n_ahead  
        test_dates = data['date'].values[last_test_idx:last_test_idx + n_ahead]  
  
        plt.plot(test_dates, y_test_rescaled[-1], 'b-o', markersize=4, label='実際')  
    
```

```
の値')
plt.plot(test_dates, y_pred_rescaled[-1], 'r-o', markersize=4, label='予測
値')
plt.fill_between(test_dates,
                 y_pred_rescaled[-1] - y_pred_rescaled[-1] * 0.2,
                 y_pred_rescaled[-1] + y_pred_rescaled[-1] * 0.2,
                 color='red',
                 alpha=0.2,
                 label='信頼区間(仮)')
plt.title(f'LSTM による {n_ahead} 日先予測 (RMSE={rmse:.2f}, R2={r2:.2f})',
          fontsize=15)
plt.xlabel('日付')
plt.ylabel('新規感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
plt.tight_layout()
plt.show()

return {'rmse': rmse, 'r2': r2, 'y_pred': y_pred_rescaled, 'y_test':
y_test_rescaled}

except ImportError:
    print("TensorFlow がインストールされていません。")
    return None

# LSTM モデルのデータ準備
X_train, X_test, y_train, y_test, scaler = prepare_lstm_data(
    data=processed_data,
    n_lag=14, # 14日間のデータから
    n_ahead=7 # 7日先を予測
)

if X_train is not None:
    # LSTM モデルの構築と学習
    lstm_model, lstm_history = build_and_train_lstm(
        X_train, y_train, X_test, y_test,
        epochs=50, batch_size=16
    )

    # LSTM モデルの評価
    if lstm_model is not None:
        lstm_evaluation = evaluate_lstm_forecast(
            model=lstm_model,
            X_test=X_test,
            y_test=y_test,
            scaler=scaler,
            data=processed_data,
            n_lag=14,
            n_ahead=7
        )
```

## アンサンブル予測モデル

```
def ensemble_forecasts(data, forecast_horizon=14):
    """複数の予測モデルを組み合わせたアンサンブル予測を行う関数"""

    # 1. Prophet モデル（前の関数を使用）
    prophet_model, prophet_forecast = train_prophet_model(data,
    forecast_periods=forecast_horizon)

    # 2. 指数平滑法予測
    from statsmodels.tsa.holtwinters import ExponentialSmoothing

    # モデルの準備
    hw_model = ExponentialSmoothing(
        data['new_cases_smooth'].values,
        seasonal_periods=7,
        trend='add',
        seasonal='add'
    ).fit()

    # 予測の実行
    hw_forecast = hw_model.forecast(forecast_horizon)

    # 3. 自己回帰移動平均 (ARIMA) モデル
    from statsmodels.tsa.arima.model import ARIMA

    # モデルの準備
    arima_model = ARIMA(data['new_cases_smooth'].values, order=(7, 1, 7)).fit()

    # 予測の実行
    arima_forecast = arima_model.forecast(forecast_horizon)

    # 予測結果の可視化
    plt.figure(figsize=(14, 8))

    # 学習データのプロット
    plt.plot(data['date'], data['new_cases_smooth'], 'b.', alpha=0.6, label='実測データ')

    # 予測開始日
    last_date = data['date'].iloc[-1]
    forecast_dates = pd.date_range(start=last_date, periods=forecast_horizon + 1)
    [1:]

    # 各モデルの予測をプロット
    plt.plot(forecast_dates,
    prophet_forecast['yhat'].tail(forecast_horizon).values,
    'r-', linewidth=2, label='Prophet')
    plt.plot(forecast_dates, hw_forecast, 'g-', linewidth=2, label='Holt-Winters')
    plt.plot(forecast_dates, arima_forecast, 'm-', linewidth=2, label='ARIMA')

    # アンサンブル（平均）予測
```

```

ensemble_forecast = (prophet_forecast['yhat'].tail(forecast_horizon).values +
                     hw_forecast +
                     arima_forecast) / 3

plt.plot(forecast_dates, ensemble_forecast, 'k-', linewidth=3, label='アンサンブル (平均)')

# 学習データと予測データの境界線
plt.axvline(x=last_date, color='gray', linestyle='--', alpha=0.7)
plt.text(last_date, plt.ylim()[1]*0.9, '予測開始', rotation=90,
         verticalalignment='top')

plt.title('複数モデルによるアンサンブル予測', fontsize=15)
plt.xlabel('日付')
plt.ylabel('新規感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
plt.tight_layout()
plt.show()

# 結果をデータフレームにまとめる
forecast_df = pd.DataFrame({
    'date': forecast_dates,
    'prophet': prophet_forecast['yhat'].tail(forecast_horizon).values,
    'holt_winters': hw_forecast,
    'arima': arima_forecast,
    'ensemble': ensemble_forecast
})

return forecast_df

# アンサンブル予測の実行
try:
    ensemble_results = ensemble_forecasts(processed_data, forecast_horizon=14)
    print("アンサンブル予測結果:")
    print(ensemble_results.head())
except Exception as e:
    print(f"アンサンブル予測実行中にエラーが発生しました: {e}")
    print("必要なライブラリが正しくインストールされているか確認してください。")

```

⌚ **初心者向けヒント**：機械学習モデルは複雑なパターンを捉えることができますが、疫学的解釈が難しいという欠点があります。一方、メカニズムモデル（SIRなど）は解釈しやすいものの、複雑な関係性を捉えきれないことがあります。両者の長所を組み合わせたハイブリッドモデルも研究されています。

## 12. 再生産数と流行の閾値

感染症の拡大を理解する上で最も重要な指標の一つが「再生産数」です。この章では再生産数の概念と計算方法、そしてそれに基づく流行閾値について学びます。

基本再生産数  $R_0$  と実効再生産数  $R_t$

```
def calculate_reproduction_number(data, method='exponential_growth',
                                  serial_interval_mean=4.7, serial_interval_sd=2.9,
                                  window_size=7):
    """
    様々な方法で再生産数を計算する関数

    Parameters:
    -----
    data : pandas.DataFrame
        新規症例数データを含むデータフレーム
    method : str
        計算方法 ('exponential_growth', 'epiestim', 'sequential_bayesian')
    serial_interval_mean : float
        平均世代時間 (感染から次の感染までの平均時間)
    serial_interval_sd : float
        世代時間の標準偏差
    window_size : int
        推定のための時間窓サイズ

    Returns:
    -----
    pd.DataFrame
        日付と再生産数を含むデータフレーム
    """
    import warnings
    warnings.filterwarnings('ignore')

    # データのコピーを作成
    result = data[['date', 'new_cases']].copy()

    if method == 'exponential_growth':
        # 指数成長法による簡易的な R 推定
        # log(症例数)の差分から成長率を計算
        smooth_cases = data['new_cases_smooth'].replace(0, 0.1) # 0を0.1に置換
        log_cases = np.log(smooth_cases)
        growth_rate = log_cases.diff().rolling(window=window_size).mean()

        # 成長率から再生産数を計算 (R = 1 + r * τ、τは平均世代時間)
        result['R'] = 1 + growth_rate * serial_interval_mean

    elif method == 'epiestim':
        try:
            import epyestim
            import epyestim.covid19 as covid19

            # EpiEstimを使用した実効再生産数の推定
            # 世代時間分布の設定
            generation_time = covid19.generate_standard_gt()

            # 実効再生産数の計算
            R_values = epyestim.estimate_R(data['new_cases'].values,
                                            gt_distribution=generation_time,
```

```
smoothing_window=window_size)

# 結果をデータフレームに格納
result['R'] = R_values

except ImportError:
    print("epyestimライブラリがインストールされていません。代わりに指数成長法を使用します。")
    # 代替として指数成長法を使用
    return calculate_reproduction_number(data,
method='exponential_growth',

serial_interval_mean=serial_interval_mean,
                                window_size=window_size)

elif method == 'sequential_bayesian':
    try:
        import pyro
        import pyro.distributions as dist
        from pyro.infer import MCMC, NUTS

        # 簡略化したベイズモデルを使用
        # 実際の実装では詳細なモデルが必要です
        def bayes_R_model(cases):
            # シンプルなベイズモデル
            R = pyro.sample('R', dist.LogNormal(0.0, 1.0))

            # 期待される新規症例数
            lam = R * cases[:-1] / serial_interval_mean

            # 觀測データの尤度
            with pyro.plate('data', len(cases)-1):
                pyro.sample('obs', dist.Poisson(lam), obs=cases[1:])

        return R

        # ウィンドウごとにベイズ推定を実行
        Rs = []
        for i in range(len(data) - window_size):
            # 部分時系列の抽出
            window_cases = data['new_cases'].iloc[i:i+window_size].values

            # MCMCサンプラーの設定
            nuts_kernel = NUTS(bayes_R_model)
            mcmc = MCMC(nuts_kernel, num_samples=500, warmup_steps=100)

            # MCMCによる推定
            mcmc.run(window_cases)

            # 事後分布の平均を取得
            posterior_samples = mcmc.get_samples()
            R_mean = posterior_samples['R'].mean().item()

            Rs.append(R_mean)
```

```

# 結果をデータフレームに格納 ( 最初の window_size 日分はNaN)
result['R'] = [np.nan] * window_size + Rs

except ImportError:
    print("pyroライブラリがインストールされていません。代わりに指数成長法を使用します。")
    # 代替として指数成長法を使用
    return calculate_reproduction_number(data,
method='exponential_growth',

serial_interval_mean=serial_interval_mean,
                                         window_size=window_size)
else:
    raise ValueError(f"未知の方法: {method}。'exponential_growth', 'epiestim',
'sequential_bayesian'のいずれかを指定してください。")

return result

# 再生産数の計算
reproduction = calculate_reproduction_number(processed_data,
method='exponential_growth',
                                         serial_interval_mean=4.7, window_size=7)

# 結果の可視化
plt.figure(figsize=(14, 8))

# 上: 新規症例数
plt.subplot(2, 1, 1)
plt.plot(processed_data['date'], processed_data['new_cases_smooth'], 'b-', linewidth=2)
plt.title('平滑化された新規症例数', fontsize=15)
plt.ylabel('症例数')
plt.grid(True, linestyle='--', alpha=0.7)

# 下: 再生産数
plt.subplot(2, 1, 2)
plt.plot(reproduction['date'], reproduction['R'], 'r-', linewidth=2)
plt.axhline(y=1, color='k', linestyle='-', alpha=0.3)
plt.title('時間変化する実効再生産数 (Rt)', fontsize=15)
plt.xlabel('日付')
plt.ylabel('再生産数')
plt.ylim(0, max(3, reproduction['R'].max() + 0.5))
plt.grid(True, linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()

```

## 年齢構造を持つ再生産数

```

def calculate_age_structured_reproduction_number(contact_matrix, susceptibility,
infectivity, recovery_rate):
    """

```

年齢構造を持つ再生産数を計算する関数

Parameters:

```
-----
contact_matrix : numpy.ndarray
    年齢層間の接触行列
susceptibility : numpy.ndarray
    各年齢層の感受性（相対値）
infectivity : numpy.ndarray
    各年齢層の感染性（相対値）
recovery_rate : float or numpy.ndarray
    回復率（1/感染期間）
```

Returns:

```
-----
float
    基本再生産数  $R_0$ 
numpy.ndarray
    年齢層別の寄与度
"""
# 行列サイズ（年齢層数）
n = len(susceptibility)

# 次世代行列の計算
if isinstance(recovery_rate, (int, float)):
    # 回復率が全年齢層で同じ場合
    recovery_rate_vector = np.ones(n) * recovery_rate
else:
    # 年齢層ごとに回復率が異なる場合
    recovery_rate_vector = recovery_rate

# 対角行列への変換
S_diag = np.diag(susceptibility)
I_diag = np.diag(infectivity)
G_diag = np.diag(1 / recovery_rate_vector)

# 次世代行列の計算 (NGM: Next Generation Matrix)
NGM = S_diag @ contact_matrix @ I_diag @ G_diag

# 最大固有値（基本再生産数）の計算
eigenvalues, eigenvectors = np.linalg.eig(NGM)
dominant_idx = np.argmax(np.real(eigenvalues))
R0 = np.real(eigenvalues[dominant_idx])

# 各年齢層の寄与度（固有ベクトルの要素）
contribution = np.abs(eigenvectors[:, dominant_idx])
# 寄与度を正規化
contribution = contribution / np.sum(contribution)

return R0, contribution

# サンプルの接触行列とパラメータの設定
def generate_sample_contact_matrix(n_age_groups=9, assortative_factor=0.3):
    """年齢層間の接触行列を生成する関数"""

```

```
# 基本的な接触率（全年齢層で同じと仮定）
base_contact = np.ones((n_age_groups, n_age_groups))

# 同年齢層での接触を強める（assortative mixing）
for i in range(n_age_groups):
    base_contact[i, i] += assortative_factor

# 隣接する年齢層での接触も若干強める
for i in range(n_age_groups-1):
    base_contact[i, i+1] += assortative_factor * 0.5
    base_contact[i+1, i] += assortative_factor * 0.5

# 学校や職場での接触パターンを模倣
# 子供同士（0-1群）の接触
base_contact[0:2, 0:2] += 0.2
# 勤労世代（2-5群）の接触
base_contact[2:6, 2:6] += 0.1

return base_contact

# 年齢層別のパラメータ設定
n_age_groups = 9
age_groups = [f"{i*10}-{(i+1)*10-1}" for i in range(n_age_groups-1)]
age_groups.append(f"({n_age_groups-1}*10)+")

# 接触行列の生成
contact_matrix = generate_sample_contact_matrix(n_age_groups)

# 年齢別感受性：子供が低く、高齢者が高いと仮定
susceptibility = np.array([0.4, 0.6, 0.8, 0.9, 1.0, 1.2, 1.5, 1.8, 2.0])

# 年齢別感染性：全年齢同じと仮定
infectivity = np.ones(n_age_groups)

# 回復率（1/感染期間）：平均5日間の感染期間と仮定
recovery_rate = 1/5

# 年齢構造を持つ再生産数の計算
R0, age_contribution = calculate_age_structured_reproduction_number(
    contact_matrix, susceptibility, infectivity, recovery_rate)

# 結果の表示
print(f"基本再生産数 R₀: {R0:.2f}")

# 年齢層別寄与度の可視化
plt.figure(figsize=(12, 6))
plt.bar(age_groups, age_contribution, color='steelblue', alpha=0.7)
plt.title(f'基本再生産数 R₀ = {R0:.2f} への年齢層別寄与度', fontsize=15)
plt.xlabel('年齢層')
plt.ylabel('寄与度')
plt.grid(True, linestyle='--', alpha=0.7, axis='y')
plt.tight_layout()
plt.show()
```

```
# 接触行列のヒートマップ表示
plt.figure(figsize=(10, 8))
sns.heatmap(contact_matrix, annot=True, fmt=".1f", cmap='YlOrRd',
            xticklabels=age_groups, yticklabels=age_groups)
plt.title('年齢層間の接触行列', fontsize=15)
plt.xlabel('感染者の年齢層')
plt.ylabel('被接触者の年齢層')
plt.tight_layout()
plt.show()
```

## 流行閾値と集団免疫

```
def calculate_herd_immunity_threshold(R0, heterogeneous=False,
susceptibility=None):
    """
    集団免疫の閾値を計算する関数

    Parameters:
    -----
    R0 : float
        基本再生産数
    heterogeneous : bool
        不均一な感受性を考慮するかどうか
    susceptibility : numpy.ndarray, optional
        各集団の感受性（相対値）と人口割合のタプルのリスト

    Returns:
    -----
    float
        集団免疫の閾値（必要な免疫保有者の割合）

    """
    if not heterogeneous or susceptibility is None:
        # 均一な集団における閾値:  $1 - 1/R_0$ 
        threshold = 1 - 1/R0
    else:
        # 不均一な集団における閾値
        # 感受性の2次モーメントの計算
        s_array = np.array([s for s, _ in susceptibility])
        p_array = np.array([p for _, p in susceptibility])

        mean_s = np.sum(s_array * p_array)
        mean_s2 = np.sum((s_array**2) * p_array)

        # CV2 (変動係数の2乗) : (分散)/(平均の2乗)
        cv2 = (mean_s2 - mean_s**2) / mean_s**2

        # 不均一性を考慮した閾値:  $1 - 1/(R_0 * (1 + CV^2))$ 
        threshold = 1 - 1/(R0 * (1 + cv2))

    return threshold
```

```

# 均一な集団での集団免疫閾値の計算
hit_homogeneous = calculate_herd_immunity_threshold(R0)

# 不均一な集団（年齢による感受性の違い）を考慮した集団免疫閾値の計算
# 感受性と人口割合のリスト
heterogeneous_susceptibility = [(s, 1/n_age_groups) for s in susceptibility]
hit_heterogeneous = calculate_herd_immunity_threshold(
    R0, heterogeneous=True, susceptibility=heterogeneous_susceptibility)

# 結果の表示
print(f"基本再生産数: R₀ = {R0:.2f}")
print(f"均一な集団での集団免疫閾値: {hit_homogeneous:.2%}")
print(f"不均一な集団（年齢構造）での集団免疫閾値: {hit_heterogeneous:.2%}")

# R₀と集団免疫閾値の関係の可視化
r0_values = np.linspace(1, 10, 100)
hit_values = [calculate_herd_immunity_threshold(r) for r in r0_values]

plt.figure(figsize=(10, 6))
plt.plot(r0_values, hit_values, 'b-', linewidth=2)
plt.axvline(x=R0, color='r', linestyle='--', label=f'計算されたR₀ = {R0:.2f}')
plt.axhline(y=hit_homogeneous, color='r', linestyle='--')

# 主要感染症のR₀値を追加
diseases = {
    '季節性インフルエンザ': 1.3,
    'SARS': 3.0,
    'COVID-19（初期株）': 2.5,
    'COVID-19（Delta株）': 5.0,
    '麻疹': 12.0,
}

# 主要感染症をプロット
for disease, r0 in diseases.items():
    if r0 <= 10: # 表示範囲内の場合のみプロット
        hit = 1 - 1/r0
        plt.plot(r0, hit, 'o', markersize=8, label=f'{disease}: R₀ = {r0}')

plt.title('基本再生産数と集団免疫閾値の関係', fontsize=15)
plt.xlabel('基本再生産数 (R₀)')
plt.ylabel('集団免疫閾値 (割合)')
plt.grid(True, linestyle='--', alpha=0.7)
plt.xlim(1, 10)
plt.ylim(0, 1)
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()

```

💡 初心者向けヒント：再生産数は感染症の拡大力を表す最も基本的な指標です。 $R > 1$  の場合は流行が拡大し、 $R < 1$  の場合は収束します。年齢構造や空間分布を考慮することで、より現実的なモデルを構築できます。

## 13. 介入効果のシミュレーション

感染症対策の効果を予測するためには、様々な介入施策をモデルに組み込む必要があります。この章では行動制限、ワクチン接種、検査・隔離などの介入効果をシミュレーションする方法を学びます。

### 社会的距離の確保と接触削減

```
def simulate_social_distancing(params, intervention_time, contact_reduction,
duration=None):
    """
    社会的距離の確保（接触削減）の効果をシミュレーションする関数

    Parameters:
    -----
    params : dict
        SIRモデルのパラメータ（ $\beta$ 、 $\gamma$ など）
    intervention_time : int
        介入開始日（シミュレーション開始からの日数）
    contact_reduction : float
        接触削減率（0から1の範囲、1で100%削減）
    duration : int, optional
        介入期間（日数）。Noneの場合は終了まで継続

    Returns:
    -----
    numpy.ndarray
        時点t、状態i、シナリオjでの人口
    """

    # パラメータの展開
    beta = params['beta']
    gamma = params['gamma']
    population = params['N']
    I0 = params['I0']
    t_max = params['t_max']

    # 初期状態
    S0 = population - I0
    R0 = 0

    # 時間配列
    t = np.arange(t_max)

    # 介入の有無による2つのシナリオ
    scenarios = ['ベースライン（介入なし）', '社会的距離の確保']

    # シミュレーション結果の格納用配列（時点、SIR状態、シナリオ）
    results = np.zeros((t_max, 3, 2))

    # ODEシステムの定義（ベースライン）
    def sir_model_baseline(y, t, beta, gamma):
        S, I, R = y
        dSdt = -beta * S * I / population
        dIdt = beta * S * I / population - gamma * I
        dRdt = gamma * I
        return [dSdt, dIdt, dRdt]
```

```
dIdt = beta * S * I / population - gamma * I
dRdt = gamma * I
return [dSdt, dIdt, dRdt]

# ODEシステムの定義（介入あり）
def sir_model_intervention(y, t, beta, gamma, intervention_time,
contact_reduction, duration):
    S, I, R = y

    # 介入期間中は接触率が減少
    if t >= intervention_time and (duration is None or t < intervention_time + duration):
        effective_beta = beta * (1 - contact_reduction)
    else:
        effective_beta = beta

    dSdt = -effective_beta * S * I / population
    dIdt = effective_beta * S * I / population - gamma * I
    dRdt = gamma * I
    return [dSdt, dIdt, dRdt]

# ベースラインシナリオのシミュレーション
y0 = [S0, I0, R0]
solution = spi.odeint(sir_model_baseline, y0, t, args=(beta, gamma))
results[:, :, 0] = solution

# 介入シナリオのシミュレーション
solution = spi.odeint(sir_model_intervention, y0, t, args=(beta, gamma,
intervention_time, contact_reduction, duration))
results[:, :, 1] = solution

# 結果の可視化
plt.figure(figsize=(15, 10))

# 感染者数の比較
plt.subplot(2, 1, 1)
for i, scenario in enumerate(scenarios):
    plt.plot(t, results[:, 1, i], label=scenario, linewidth=2)

# 介入期間の表示
if duration is None:
    plt.axvspan(intervention_time, t_max, color='gray', alpha=0.3, label='介入期間')
else:
    plt.axvspan(intervention_time, intervention_time + duration, color='gray',
alpha=0.3, label='介入期間')

plt.title('社会的距離の確保による介入効果（感染者数）', fontsize=15)
plt.xlabel('シミュレーション日数')
plt.ylabel('感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# 累積感染者数 (S0 - S) の比較
```

```
plt.subplot(2, 1, 2)
for i, scenario in enumerate(scenarios):
    plt.plot(t, S0 - results[:, 0, i], label=scenario, linewidth=2)

# 介入期間の表示
if duration is None:
    plt.axvspan(intervention_time, t_max, color='gray', alpha=0.3, label='介入期間')
else:
    plt.axvspan(intervention_time, intervention_time + duration, color='gray', alpha=0.3, label='介入期間')

plt.title('社会的距離の確保による介入効果 ( 累積感染者数 )', fontsize=15)
plt.xlabel('シミュレーション日数')
plt.ylabel('累積感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

plt.tight_layout()
plt.show()

# 介入効果の定量的評価
max_infected_baseline = np.max(results[:, 1, 0])
max_infected_intervention = np.max(results[:, 1, 1])

total_infected_baseline = S0 - results[-1, 0, 0]
total_infected_intervention = S0 - results[-1, 0, 1]

peak_reduction = (max_infected_baseline - max_infected_intervention) / max_infected_baseline
total_reduction = (total_infected_baseline - total_infected_intervention) / total_infected_baseline

print(f"ピーク時感染者数の削減効果: {peak_reduction:.2%}")
print(f"累積感染者数の削減効果: {total_reduction:.2%}")

return results

# SIRモデルのパラメータ設定
params = {
    'beta': 0.3,          # 感染率
    'gamma': 0.1,         # 回復率 ( 1/感染期間 )
    'N': 100000,          # 人口
    'I0': 100,            # 初期感染者数
    't_max': 200          # シミュレーション期間 ( 日数 )
}

# 介入効果のシミュレーション
# 30日目から60%の接触削減を30日間実施
social_distancing_results = simulate_social_distancing(
    params, intervention_time=30, contact_reduction=0.6, duration=30)
```

## ワクチン接種戦略のシミュレーション

```
def simulate_vaccination_strategies(params, strategies, coverage=0.7):
    """
    異なるワクチン接種戦略の効果をシミュレーションする関数

    Parameters:
    -----
    params : dict
        SIRモデルのパラメータ
    strategies : list of dict
        シミュレーションする接種戦略のリスト
    coverage : float
        目標接種率 ( 0から1の範囲 )

    Returns:
    -----
    numpy.ndarray
        シミュレーション結果
    """

    # パラメータの展開
    beta = params['beta']
    gamma = params['gamma']
    population = params['N']
    I0 = params['I0']
    t_max = params['t_max']

    # 初期状態
    S0 = population - I0
    R0 = 0
    V0 = 0 # 初期ワクチン接種者

    # 時間配列
    t = np.arange(t_max)

    # 結果の格納用配列 ( 時点、SIRV状態、戦略 )
    n_strategies = len(strategies) + 1 # ベースライン + 各戦略
    results = np.zeros((t_max, 4, n_strategies))

    # SIRV (Susceptible-Infected-Recovered-Vaccinated) モデルの定義
    def sirv_model(y, t, beta, gamma, vaccination_rate, start_day, duration):
        S, I, R, V = y

        # ワクチン接種による免疫獲得 ( 簡略化モデル )
        if t >= start_day and (duration is None or t < start_day + duration):
            # 未感染者へのワクチン接種
            vac_S = min(vaccination_rate * S, S)
        else:
            vac_S = 0

        # 感染ダイナミクス
        infection = beta * S * I / population
```

```
dSdt = -infection - vac_S
dIdt = infection - gamma * I
dRdt = gamma * I
dVdt = vac_S

return [dSdt, dIdt, dRdt, dVdt]

# ベースラインシナリオ（ワクチンなし）のシミュレーション
y0 = [S0, I0, R0, V0]
solution = spi.odeint(sirv_model, y0, t, args=(beta, gamma, 0, 0, None))
results[:, :, 0] = solution

# 各戦略のシミュレーション
for i, strategy in enumerate(strategies):
    start_day = strategy['start_day']
    duration = strategy.get('duration')
    daily_rate = strategy.get('daily_rate', 0.01) # デフォルトは人口の1%/日

    y0 = [S0, I0, R0, V0]
    solution = spi.odeint(sirv_model, y0, t, args=(beta, gamma, daily_rate,
start_day, duration))
    results[:, :, i+1] = solution

# 結果の可視化
plt.figure(figsize=(15, 12))

# 1. 感染者数の比較
plt.subplot(3, 1, 1)
plt.plot(t, results[:, 1, 0], 'r-', label='ベースライン（ワクチンなし）', linewidth=2)

for i, strategy in enumerate(strategies):
    plt.plot(t, results[:, 1, i+1], '--', label=strategy['name'], linewidth=2)

plt.title('異なるワクチン接種戦略の効果（感染者数）', fontsize=15)
plt.xlabel('シミュレーション日数')
plt.ylabel('感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# 2. 累積感染者数の比較
plt.subplot(3, 1, 2)
plt.plot(t, S0 - results[:, 0, 0] - results[:, 3, 0], 'r-', label='ベースライン（ワクチンなし）', linewidth=2)

for i, strategy in enumerate(strategies):
    plt.plot(t, S0 - results[:, 0, i+1] - results[:, 3, i+1], '--',
label=strategy['name'], linewidth=2)

plt.title('異なるワクチン接種戦略の効果（累積感染者数）', fontsize=15)
plt.xlabel('シミュレーション日数')
plt.ylabel('累積感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
```

```
# 3. ワクチン接種率の比較
plt.subplot(3, 1, 3)

for i, strategy in enumerate(strategies):
    plt.plot(t, results[:, 3, i+1] / population, '--', label=strategy['name'],
linewidth=2)

plt.axhline(y=coverage, color='k', linestyle='--', label=f'目標接種率
({coverage*100:.0f}%)')

plt.title('ワクチン接種率の時間的推移', fontsize=15)
plt.xlabel('シミュレーション日数')
plt.ylabel('接種率 (人口比)')
plt.grid(True, linestyle='--', alpha=0.7)
plt.ylim(0, 1)
plt.legend()

plt.tight_layout()
plt.show()

# 各戦略の効果の定量的評価
print("各戦略の効果比較:")
print("-" * 60)
print("          | ピーク感染者数削減 | 最終累積感染者削減 | 目標接種率到
達日")
print("-" * 60)

max_infected_baseline = np.max(results[:, 1, 0])
total_infected_baseline = S0 - results[-1, 0, 0] - results[-1, 3, 0]

for i, strategy in enumerate(strategies):
    max_infected = np.max(results[:, 1, i+1])
    total_infected = S0 - results[-1, 0, i+1] - results[-1, 3, i+1]

    peak_reduction = (max_infected_baseline - max_infected) /
max_infected_baseline
    total_reduction = (total_infected_baseline - total_infected) /
total_infected_baseline

    # 目標接種率に達する日数
    vac_idx = np.where(results[:, 3, i+1] / population >= coverage)[0]
    target_day = vac_idx[0] if len(vac_idx) > 0 else np.inf

    day_str = f"{target_day}日" if target_day != np.inf else "達成せず"

    print(f"{strategy['name']:<20} | {peak_reduction:>18.2%} |
{total_reduction:>18.2%} | {day_str:>15}")

print("-" * 60)

return results

# ワクチン接種戦略のシミュレーション
```

```

params = {
    'beta': 0.3,          # 感染率
    'gamma': 0.1,         # 回復率
    'N': 100000,          # 人口
    'I0': 100,            # 初期感染者数
    't_max': 200          # シミュレーション期間
}

# 異なる戦略の定義
strategies = [
    {
        'name': '早期緩速接種',
        'start_day': 10,
        'daily_rate': 0.005,  # 1日あたり人口の0.5%
        'duration': None     # 終了まで継続
    },
    {
        'name': '中期急速接種',
        'start_day': 50,
        'daily_rate': 0.015,  # 1日あたり人口の1.5%
        'duration': None     # 終了まで継続
    },
    {
        'name': '2段階接種',
        'start_day': 20,
        'daily_rate': 0.003,  # 最初は0.3%/日
        'duration': 60,
        'second_stage': {
            'start_day': 80,
            'daily_rate': 0.01,  # その後1%/日
            'duration': None
        }
    }
]

# 2段階接種戦略の特別処理
# 実際の実装では、sirv_modelを拡張して2段階接種を直接サポートする必要がある

# ワクチン接種戦略のシミュレーション実行
vaccination_results = simulate_vaccination_strategies(params, strategies[:2],
coverage=0.7)

```

## 年齢層別のワクチン優先順位

```

def simulate_age_prioritized_vaccination(age_data, contact_matrix, params,
                                            priority_strategies, coverage=0.7):
    """

```

年齢層別のワクチン優先順位戦略をシミュレーションする関数

**Parameters:**

-----

```
age_data : pandas.DataFrame  
    年齢層別の人ロデータ  
contact_matrix : numpy.ndarray  
    年齢層間の接觸行列  
params : dict  
    シミュレーションパラメータ  
priority_strategies : list of dict  
    優先順位戦略のリスト  
coverage : float  
    目標全体接種率
```

**Returns:**

-----  
dict

各戦略のシミュレーション結果

```
"""\nimport copy
```

```
# パラメータの展開
```

```
beta = params['beta']
```

```
gamma = params['gamma']
```

```
I0_ratio = params.get('I0_ratio', 0.001) # 初期感染者率 (デフォルト0.1%)
```

```
t_max = params['t_max']
```

```
# 年齢層の数
```

```
n_groups = len(age_data)
```

```
# 年齢層別初期状態の設定
```

```
N_vec = age_data['population'].values
```

```
total_population = np.sum(N_vec)
```

```
# 各年齢層で人口比例した初期感染者を設定
```

```
I0_vec = I0_ratio * N_vec
```

```
S0_vec = N_vec - I0_vec
```

```
R0_vec = np.zeros(n_groups)
```

```
V0_vec = np.zeros(n_groups) # 初期ワクチン接種者
```

```
# 時間配列
```

```
t = np.arange(t_max)
```

```
# 結果の格納用辞書
```

```
results = {}
```

```
# ベースライン (ワクチンなし) の定義
```

```
baseline = {'name': 'ワクチンなし'}
```

```
strategies_with_baseline = [baseline] + priority_strategies
```

```
# 年齢構造化SIRVモデルの定義
```

```
def age_structured_sirv_model(y, t, beta, gamma, contact_matrix, N_vec,  
vac_rates=None):  
    """年齢構造化SIRVモデル"""  
    # 状態変数の展開  
    y_reshaped = y.reshape(4, n_groups)  
    S = y_reshaped[0, :]
```

```
I = y_reshaped[1, :]
R = y_reshaped[2, :]
V = y_reshaped[3, :]

# 各年齢層の力 ( 感染力 )
# force_of_infection[i] = 年齢層iの感受性個体が感染する力
force_of_infection = np.zeros(n_groups)

for i in range(n_groups):
    # 各年齢層からの感染力の合計
    force_i = 0
    for j in range(n_groups):
        # 年齢層j→年齢層iへの感染力 = 接触率 * 年齢層jの感染者割合
        force_i += beta * contact_matrix[i, j] * I[j] / N_vec[j]
    force_of_infection[i] = force_i

# 感染による変化
infection = S * force_of_infection

# ワクチン接種による変化
vaccination = np.zeros(n_groups)
if vac_rates is not None:
    # 指定された接種率に従ってワクチン接種
    for i in range(n_groups):
        # 未感染者へのワクチン接種
        vaccination[i] = min(vac_rates[i] * S[i], S[i])

# 微分方程式
dSdt = -infection - vaccination
dIdt = infection - gamma * I
dRdt = gamma * I
dVdt = vaccination

# 結果を結合
result = np.vstack([dSdt, dIdt, dRdt, dVdt]).flatten()
return result

# 各戦略のシミュレーション
for strategy_idx, strategy in enumerate(strategies_with_baseline):
    # 初期状態
    y0 = np.vstack([S0_vec, I0_vec, R0_vec, V0_vec]).flatten()

    # この戦略のワクチン接種率設定
    if strategy_idx == 0: # ベースライン
        result = spi.odeint(age_structured_sirv_model, y0, t,
                            args=(beta, gamma, contact_matrix, N_vec, None))
    else:
        # 年齢層ごとの1日あたりワクチン接種率の時系列
        vac_rates_time = np.zeros((t_max, n_groups))

        # 優先順位に基づく接種率の設定
        priority_sequence = strategy['priority_sequence']
        start_day = strategy.get('start_day', 0)
        daily_rate = strategy.get('daily_rate', 0.005) # デフォルト0.5%/日
```

```
# 各時点での接種率を計算
current_target_group = 0 # 現在のターゲット年齢層インデックス
cumulative_vac = np.zeros(n_groups) # 累積接種者数

for day in range(start_day, t_max):
    if current_target_group >= len(priority_sequence):
        break # すべての年齢層の目標達成

    # 現在のターゲット年齢層
    target_idx = priority_sequence[current_target_group]

    # この年齢層の目標接種者数
    target_vac = coverage * N_vec[target_idx]

    # まだ目標に達していなければ接種を継続
    if cumulative_vac[target_idx] < target_vac:
        # この日の接種率
        vac_rates_time[day, target_idx] = daily_rate

        # 累積接種者数の更新（簡略化のため、すべての接種が成功すると仮定）
        cumulative_vac[target_idx] += daily_rate * N_vec[target_idx]

    # 目標達成判定
    if cumulative_vac[target_idx] >= target_vac:
        current_target_group += 1 # 次の優先グループへ

# 各時点でのODEを積分
result = np.zeros((t_max, 4 * n_groups))
result[0, :] = y0

for day in range(1, t_max):
    prev_state = result[day-1, :]

    # 1日分の積分
    day_result = spi.odeint(age_structured_sirv_model, prev_state, [0,
1],
                           args=(beta, gamma, contact_matrix, N_vec,
vac_rates_time[day-1, :]))

    # 最終状態を保存
    result[day, :] = day_result[-1, :]

# 結果の保存
results[strategy['name']] = result.reshape(t_max, 4, n_groups)

# 結果の可視化
plt.figure(figsize=(15, 12))

# 1. 総感染者数の比較
plt.subplot(2, 1, 1)
for strategy in strategies_with_baseline:
    strategy_name = strategy['name']
    # 全年齢層の感染者合計
```

```
total_infected = np.sum(results[strategy_name][:, 1, :], axis=1)
plt.plot(t, total_infected, label=strategy_name, linewidth=2)

plt.title('年齢層別ワクチン優先順位戦略の効果（総感染者数）', fontsize=15)
plt.xlabel('シミュレーション日数')
plt.ylabel('総感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# 2. 年齢層別感染者数
plt.subplot(2, 1, 2)
# 最後の戦略の年齢層別感染者数を表示
last_strategy = strategies_with_baseline[-1]
last_result = results[last_strategy['name']]

for i in range(n_groups):
    plt.plot(t, last_result[:, 1, i], label=f'年齢層 {age_data["age_group"].iloc[i]}', linewidth=2)

plt.title(f'戦略「{last_strategy["name"]}」での年齢層別感染者数', fontsize=15)
plt.xlabel('シミュレーション日数')
plt.ylabel('感染者数')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

plt.tight_layout()
plt.show()

# 戰略の効果比較
print("各ワクチン優先順位戦略の効果比較:")
print("-" * 80)
print("戦略 | 総感染者ピーク削減 | 累積感染者削減 | 高齢者感染削減 |")
目標接種率到達日)
print("-" * 80)

baseline_result = results['ワクチンなし']
total_infected_baseline = np.sum(baseline_result[:, 1, :], axis=1)
max_infected_baseline = np.max(total_infected_baseline)
cumulative_infected_baseline = np.sum(S0_vec) - np.sum(baseline_result[-1, 0, :])

# 高齢者（最後の2つの年齢層）の感染者
elderly_idx = list(range(n_groups - 2, n_groups))
elderly_infected_baseline = np.sum(S0_vec[elderly_idx]) -
np.sum(baseline_result[-1, 0, elderly_idx])

for strategy in priority_strategies:
    strategy_name = strategy['name']
    result = results[strategy_name]

    # 総感染者数とピーク
    total_infected = np.sum(result[:, 1, :], axis=1)
    max_infected = np.max(total_infected)
```

```
# 累積感染者数
cumulative_infected = np.sum(S0_vec) - np.sum(result[-1, 0, :]) -
np.sum(result[-1, 3, :])

# 高齢者の累積感染者
elderly_infected = np.sum(S0_vec[elderly_idx]) - np.sum(result[-1, 0,
elderly_idx]) - np.sum(result[-1, 3, elderly_idx])

# 効果の計算
peak_reduction = (max_infected_baseline - max_infected) /
max_infected_baseline
cumulative_reduction = (cumulative_infected_baseline -
cumulative_infected) / cumulative_infected_baseline
elderly_reduction = (elderly_infected_baseline - elderly_infected) /
elderly_infected_baseline

# 目標接種率到達日 (簡易計算)
total_vaccinated = np.sum(result[:, 3, :], axis=1)
target_idx = np.where(total_vaccinated >= coverage * total_population)[0]
target_day = target_idx[0] if len(target_idx) > 0 else np.inf

day_str = f"{target_day}日" if target_day != np.inf else "達成せず"

print(f"{strategy_name}<19} | {peak_reduction:>17.2%} |
{cumulative_reduction:>14.2%} | {elderly_reduction:>14.2%} | {day_str:>16}")

print("-" * 80)

return results

# 年齢層別のワクチン優先順位シミュレーション
# 年齢データ
age_data = prepare_age_structured_data(use_real=True)

# 接触行列
contact_matrix = generate_sample_contact_matrix(n_age_groups=9)

# シミュレーションパラメータ
params = {
    'beta': 0.03,      # 基本感染率
    'gamma': 0.1,       # 回復率
    'I0_ratio': 0.001,  # 初期感染者割合
    't_max': 200        # シミュレーション期間
}

# 年齢インデックス (0-8) :
# 0: 0-9歳, 1: 10-19歳, ..., 8: 80歳以上

# ワクチン優先順位戦略
priority_strategies = [
{
    'name': '高齢者優先',
    'priority_sequence': [8, 7, 6, 5, 4, 3, 2, 1, 0],  # 80+歳から順に接種
    'start_day': 20,
```

```

        'daily_rate': 0.01
    },
    {
        'name': '若年層優先',
        'priority_sequence': [1, 2, 3, 4, 5, 0, 6, 7, 8], # 10-19歳から順に接種 ( 0-
9歳は後回し)
        'start_day': 20,
        'daily_rate': 0.01
    },
    {
        'name': '接触多層優先',
        # 接触数の多い順に優先 ( 例 )
        'priority_sequence': [1, 2, 3, 4, 0, 5, 6, 7, 8],
        'start_day': 20,
        'daily_rate': 0.01
    }
]
]

# 年齢層別のワクチン優先順位シミュレーション実行
age_vac_results = simulate_age_prioritized_vaccination(
    age_data, contact_matrix, params, priority_strategies, coverage=0.7)

```

⌚ **初心者向けヒント**：介入効果のシミュレーションでは、開始タイミングが重要です。早すぎると社会・経済コストが高くなり、遅すぎると効果が低減します。異なる介入の組み合わせ（ミックス戦略）も検討する価値があります。

## 14. 実データを用いた検証

理論的な数理モデルを実データに適用することで、モデルの妥当性を検証し、実世界の感染症ダイナミクスをより深く理解できます。

### 実データとモデル予測の比較

```

def validate_model_with_real_data(real_data, model_func, params, population_size,
t_span):
    """実データとモデル予測を比較する関数"""

    # モデルによる予測
    t = np.linspace(t_span[0], t_span[1], t_span[1] - t_span[0] + 1)
    model_result = model_func(params, population_size, t)

    # 実データとモデル予測の比較グラフ
    plt.figure(figsize=(14, 8))

    plt.subplot(2, 1, 1)
    plt.plot(real_data['date'], real_data['new_cases_smooth'], 'o-', color='blue',
alpha=0.7, label='実データ (平滑化)')
    plt.plot(t, model_result['new_cases'], '-', color='red', label='モデル予測')
    plt.title('新規感染者数: 実データ vs モデル予測', fontsize=15)
    plt.ylabel('新規感染者数')

```

```
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)

plt.subplot(2, 1, 2)
plt.plot(real_data['date'], real_data['cumulative_cases'], 'o-', color='blue',
alpha=0.7, label='実データ')
plt.plot(t, model_result['total_cases'], '-', color='red', label='モデル予測')
plt.title('累積感染者数: 実データ vs モデル予測', fontsize=15)
plt.xlabel('日付 / 経過日数')
plt.ylabel('累積感染者数')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()

# 予測精度の計算
# モデル予測と実データの長さを揃える
min_length = min(len(real_data), len(model_result['new_cases']))

rmse = np.sqrt(mean_squared_error(
    real_data['new_cases_smooth'][:min_length],
    model_result['new_cases'][:min_length]
))

r2 = r2_score(
    real_data['new_cases_smooth'][:min_length],
    model_result['new_cases'][:min_length]
)

metrics = {
    'RMSE': rmse,
    'R2
```

```
# 微分方程式を解く
solution = spi.solve_ivp(
    lambda t, y: sir_model(y, t, beta, gamma),
    [0, max(t)],
    [S0, I0, R0],
    t_eval=t,
    method='RK45'
)

S = solution.y[0]
I = solution.y[1]
R = solution.y[2]

# 新規感染者数の計算 (S(t-1) - S(t))
new_cases = -np.diff(S, prepend=S0)

# 累積感染者数の計算
total_cases = N - S

result = {
    'S': S,
    'I': I,
    'R': R,
    'new_cases': new_cases,
    'total_cases': total_cases
}

return result

# 最適化されたパラメータを使って実データとモデル予測を比較
# 例: β=0.3, γ=0.1 (R₀=3) のSIRモデル
opt_params = [0.3, 0.1]
population_size = 100000 # 10万人規模の人口を想定
t_span = [0, len(processed_data) - 1] # 時間範囲

# モデル検証の実行
validation_metrics = validate_model_with_real_data(
    processed_data,
    sir_model_prediction,
    opt_params,
    population_size,
    t_span
)

print("モデル検証の結果:")
for metric, value in validation_metrics.items():
    print(f"{metric}: {value:.4f}")
```

## 複数のモデルの比較

```
def compare_multiple_models(real_data, model_funcs, param_sets, model_names, population_size, t_span):
    """複数のモデルを実データと比較する関数"""

    # 時間軸
    t = np.linspace(t_span[0], t_span[1], t_span[1] - t_span[0] + 1)

    # 各モデルの予測結果を取得
    model_results = []
    for model_func, params in zip(model_funcs, param_sets):
        result = model_func(params, population_size, t)
        model_results.append(result)

    # 新規感染者数の可視化
    plt.figure(figsize=(14, 10))

    plt.subplot(2, 1, 1)
    plt.plot(real_data['date'], real_data['new_cases_smooth'], 'o-', color='black', alpha=0.6, linewidth=1, markersize=4, label='実データ（平滑化）')

    colors = ['red', 'blue', 'green', 'purple', 'orange']
    for i, (result, name) in enumerate(zip(model_results, model_names)):
        plt.plot(t, result['new_cases'], '--', color=colors[i % len(colors)], linewidth=2, label=f'モデル: {name}')

    plt.title('新規感染者数: 実データ vs 複数モデル', fontsize=15)
    plt.ylabel('新規感染者数')
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.7)

    # 累積感染者数の可視化
    plt.subplot(2, 1, 2)
    plt.plot(real_data['date'], real_data['cumulative_cases'], 'o-', color='black', alpha=0.6, linewidth=1, markersize=4, label='実データ')

    for i, (result, name) in enumerate(zip(model_results, model_names)):
        plt.plot(t, result['total_cases'], '--', color=colors[i % len(colors)], linewidth=2, label=f'モデル: {name}')

    plt.title('累積感染者数: 実データ vs 複数モデル', fontsize=15)
    plt.xlabel('日付 / 経過日数')
    plt.ylabel('累積感染者数')
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.7)

    plt.tight_layout()
    plt.show()

    # 各モデルの予測精度を計算
    comparison_metrics = []

    for i, (result, name) in enumerate(zip(model_results, model_names)):
        # モデル予測と実データの長さを揃える
```

```
min_length = min(len(real_data), len(result['new_cases']))

rmse = np.sqrt(mean_squared_error(
    real_data['new_cases_smooth'][:min_length],
    result['new_cases'][:min_length]
))

r2 = r2_score(
    real_data['new_cases_smooth'][:min_length],
    result['new_cases'][:min_length]
))

comparison_metrics.append({
    'Model': name,
    'RMSE': rmse,
    'R2
```

```
new_cases = sigma * E

# 累積感染者数の計算
total_cases = N - S

result = {
    'S': S,
    'E': E,
    'I': I,
    'R': R,
    'new_cases': new_cases,
    'total_cases': total_cases
}

return result

# 時変パラメータを持つSIRモデル
def time_varying_sir_prediction(params, N, t):
    """時間変化するパラメータを持つSIRモデルによる予測を行う関数"""
    beta_0, beta_1, t_change, gamma = params

    # 初期条件
    I0 = 1 # 初期感染者数
    R0 = 0 # 初期回復者数
    S0 = N - I0 - R0 # 初期感受性者数

    def time_varying_sir(y, t, beta_0, beta_1, t_change, gamma):
        S, I, R = y

        # 時間に依存するβの値
        if t < t_change:
            beta = beta_0
        else:
            beta = beta_1

        dSdt = -beta * S * I / N
        dIdt = beta * S * I / N - gamma * I
        dRdt = gamma * I
        return [dSdt, dIdt, dRdt]

    # 微分方程式を解く
    solution = spi.solve_ivp(
        lambda t, y: time_varying_sir(y, t, beta_0, beta_1, t_change, gamma),
        [0, max(t)],
        [S0, I0, R0],
        t_eval=t,
        method='RK45'
    )

    S = solution.y[0]
    I = solution.y[1]
    R = solution.y[2]

    # 新規感染者数の計算
```

```
new_cases = -np.diff(S, prepend=S0)

# 累積感染者数の計算
total_cases = N - S

result = {
    'S': S,
    'I': I,
    'R': R,
    'new_cases': new_cases,
    'total_cases': total_cases
}

return result

# 複数のモデルを比較
model_funcs = [
    sir_model_prediction,
    seir_model_prediction,
    time_varying_sir_prediction
]

param_sets = [
    [0.3, 0.1], # SIR: beta, gamma
    [0.3, 0.2, 0.1], # SEIR: beta, sigma, gamma
    [0.4, 0.2, 30, 0.1] # Time-varying SIR: beta_0, beta_1, t_change, gamma
]

model_names = [
    'SIR',
    'SEIR',
    'Time-varying SIR'
]

# モデル比較の実行
comparison_results = compare_multiple_models(
    processed_data,
    model_funcs,
    param_sets,
    model_names,
    population_size,
    t_span
)

print("モデル比較結果:")
print(comparison_results)
```

## クロスバリデーションによるモデル評価

```
def cross_validate_epidemic_model(data, model_func, param_ranges, n_splits=5,
population_size=100000):
```

```
"""時系列クロスバリデーションでモデルを評価する関数"""

# データを訓練用とテスト用に分割
n_samples = len(data)
test_size = n_samples // n_splits

cv_results = []

for fold in range(n_splits - 1): # 最後の分割は予測に使用できないため除外
    # 訓練/テスト区間の決定
    train_end = (fold + 1) * test_size
    test_start = train_end
    test_end = test_start + test_size

    # 訓練データとテストデータの抽出
    train_data = data.iloc[:train_end].copy()
    test_data = data.iloc[test_start:test_end].copy()

    print(f"Fold {fold+1}: 訓練 [0:{train_end}], テスト [{test_start}:{test_end}]")

    # 訓練データを使ってパラメータの最適化 ( グリッドサーチの簡易版 )
    best_params = None
    best_rmse = float('inf')

    # パラメータグリッドの生成 ( 簡略化のため2パラメータのみ )
    beta_values = np.linspace(param_ranges['beta'][0], param_ranges['beta'][1], 5)
    gamma_values = np.linspace(param_ranges['gamma'][0], param_ranges['gamma'][1], 5)

    for beta in beta_values:
        for gamma in gamma_values:
            params = [beta, gamma]

            # モデルによる予測
            t_train = np.arange(len(train_data))
            model_result = model_func(params, population_size, t_train)

            # RMSE計算
            rmse = np.sqrt(mean_squared_error(
                train_data['new_cases_smooth'],
                model_result['new_cases'][:len(train_data)]
            ))

            if rmse < best_rmse:
                best_rmse = rmse
                best_params = params

    print(f" 最適パラメータ: beta={best_params[0]:.3f}, gamma={best_params[1]:.3f}")

    # テストデータでの予測
    t_test = np.arange(test_start, test_end)
```

```
test_pred = model_func(best_params, population_size, t_test)

# テストデータでの性能評価
test_rmse = np.sqrt(mean_squared_error(
    test_data['new_cases_smooth'],
    test_pred['new_cases'][:len(test_data)])
))

test_r2 = r2_score(
    test_data['new_cases_smooth'],
    test_pred['new_cases'][:len(test_data)])
)

cv_results.append({
    'Fold': fold + 1,
    'Train RMSE': best_rmse,
    'Test RMSE': test_rmse,
    'Test R222'].mean(),
```

```

        'Mean Beta': cv_df['Beta'].mean(),
        'Mean Gamma': cv_df['Gamma'].mean()
    }

    return cv_df, cv_summary

# クロスバリデーションの実行
param_ranges = {
    'beta': [0.1, 0.5],
    'gamma': [0.05, 0.2]
}

cv_results, cv_summary = cross_validate_epidemic_model(
    processed_data,
    sir_model_prediction,
    param_ranges,
    n_splits=4,
    population_size=100000
)

print("\nクロスバリデーション結果:")
print(cv_results)

print("\n平均結果:")
for metric, value in cv_summary.items():
    print(f"{metric}: {value:.4f}")

```

**💡 初心者向けヒント :** モデルの予測精度は RMSE（平均二乗誤差の平方根）と  $R^2$  スコアで評価できます。RMSE は値が小さいほど、 $R^2$  は 1 に近いほど予測精度が高いことを示します。また、時系列データの場合は通常のランダム分割ではなく、時間順にデータを分割して評価することが重要です。

## 15. 実践的なミニプロジェクト

理論を学ぶだけでなく、実際のプロジェクトに取り組むことで理解が深まります。ここでは、実践的なミニプロジェクトを通じて、これまで学んだ知識を応用していきましょう。

### プロジェクト 1: 環境音の分類器

環境音（都市騒音、自然音、機械音など）を自動分類するシステムを開発します。

```

def environmental_sound_classifier():
    """環境音分類器のミニプロジェクト"""

    # サンプル音声ファイルのダウンロード
    !wget -q -O urban.wav "https://freesound.org/data/previews/573/573577_5834212-1q.mp3"
    !wget -q -O nature.wav "https://freesound.org/data/previews/573/573947_12723337-1q.mp3"
    !wget -q -O machine.wav "https://freesound.org/data/previews/558/558862_12295155-1q.mp3"

```

```
# ファイルの読み込みと特徴量抽出の関数
def extract_features(file_path):
    y, sr = librosa.load(file_path, sr=None)

    # 波形表示
    plt.figure(figsize=(10, 4))
    librosa.display.waveform(y, sr=sr)
    plt.title(f'波形: {file_path}')
    plt.show()
    display(Audio(y, rate=sr))

    # 特徴量抽出
    features = {}

    # 時間領域特徴量
    features['rms'] = np.mean(librosa.feature.rms(y=y)[0])
    features['zcr'] = np.mean(librosa.feature.zero_crossing_rate(y)[0])

    # スペクトル特徴量
    features['spectral_centroid'] =
        np.mean(librosa.feature.spectral_centroid(y=y, sr=sr)[0])
    features['spectral_bandwidth'] =
        np.mean(librosa.feature.spectral_bandwidth(y=y, sr=sr)[0])
    features['spectral_rolloff'] =
        np.mean(librosa.feature.spectral_rolloff(y=y, sr=sr)[0])
    features['spectral_flatness'] =
        np.mean(librosa.feature.spectral_flatness(y=y)[0])

    # MFCC
    mfcc = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)
    for i in range(13):
        features[f'mfcc_{i+1}'] = np.mean(mfcc[i])

    # メルスペクトログラム表示
    plt.figure(figsize=(10, 4))
    mel_spec = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128)
    mel_spec_db = librosa.power_to_db(mel_spec, ref=np.max)
    librosa.display.specshow(mel_spec_db, sr=sr, x_axis='time', y_axis='mel')
    plt.colorbar(format='%.2f dB')
    plt.title(f'メルスペクトログラム: {file_path}')
    plt.tight_layout()
    plt.show()

    return features

# 各音声ファイルから特徴量を抽出
print("都市騒音の分析:")
urban_features = extract_features('urban.wav')

print("\n自然音の分析:")
nature_features = extract_features('nature.wav')

print("\n機械音の分析:")
```

```
machine_features = extract_features('machine.wav')

# 特徴量の比較
features_df = pd.DataFrame({
    '都市騒音': urban_features,
    '自然音': nature_features,
    '機械音': machine_features
})

# 特徴量の可視化
plt.figure(figsize=(14, 10))

# 基本特徴量の比較
basic_features = ['rms', 'zcr', 'spectral_centroid', 'spectral_bandwidth',
'spectral_rolloff', 'spectral_flatness']
plt.subplot(2, 1, 1)
basic_df = features_df.loc[basic_features]
basic_df = basic_df / basic_df.max() # 正規化
basic_df.T.plot(kind='bar', ax=plt.gca())
plt.title('基本音響特徴量の比較（正規化済み）')
plt.ylabel('正規化値')
plt.xticks(rotation=0)

# MFCC特徴量の比較
plt.subplot(2, 1, 2)
mfcc_features = [f'mfcc_{i+1}' for i in range(13)]
mfcc_df = features_df.loc[mfcc_features]
sns.heatmap(mfcc_df, cmap='viridis', annot=True, fmt='.2f')
plt.title('MFCC特徴量の比較')

plt.tight_layout()
plt.show()

# 特徴量に基づく簡易的な分類器
print("\n簡易的な環境音分類器のデモンストレーション:")
print("1. 新しい音声が与えられたとき、既知の音源との特徴量の類似度を計算")
print("2. 最も類似度が高いカテゴリに分類")

# 新しい音声の分析（デモとして都市騒音を少し変化させたもの）
new_sound_path = 'urban.wav' # デモのために同じファイルを使用
y_new, sr_new = librosa.load(new_sound_path, sr=None)
# 少しノイズを加えて「新しい音声」とする
y_new = y_new + np.random.normal(0, 0.01, len(y_new))

# 新しい音声の特徴量抽出
new_features = {}
# 時間領域特徴量
new_features['rms'] = np.mean(librosa.feature.rms(y=y_new)[0])
new_features['zcr'] = np.mean(librosa.feature.zero_crossing_rate(y_new)[0])
# スペクトル特徴量
new_features['spectral_centroid'] =
np.mean(librosa.feature.spectral_centroid(y=y_new, sr=sr_new)[0])
new_features['spectral_bandwidth'] =
np.mean(librosa.feature.spectral_bandwidth(y=y_new, sr=sr_new)[0])
```

```

new_features[ 'spectral_rolloff' ] =
np.mean(librosa.feature.spectral_rolloff(y=y_new, sr=sr_new)[0])
new_features[ 'spectral_flatness' ] =
np.mean(librosa.feature.spectral_flatness(y=y_new)[0])
# MFCC
mfcc = librosa.feature.mfcc(y=y_new, sr=sr_new, n_mfcc=13)
for i in range(13):
    new_features[f'mfcc_{i+1}'] = np.mean(mfcc[i])

# 各カテゴリとの距離（類似度）を計算
from scipy.spatial.distance import euclidean

distances = {}
for category, features in [ ('都市騒音', urban_features), ('自然音',
nature_features), ('機械音', machine_features) ]:
    # 特微量ベクトルを抽出
    v1 = np.array([new_features[f] for f in new_features.keys()])
    v2 = np.array([features[f] for f in features.keys()])
    # 正規化
    v1 = v1 / np.linalg.norm(v1)
    v2 = v2 / np.linalg.norm(v2)
    # ユークリッド距離を計算
    dist = euclidean(v1, v2)
    distances[category] = dist

# 結果を表示
print("\n新しい音声と各カテゴリとの距離（値が小さいほど類似）:")
for category, dist in distances.items():
    print(f"{category}: {dist:.4f}")

# 最も近いカテゴリを分類結果とする
closest_category = min(distances, key=distances.get)
print(f"\n分類結果: この音声は「{closest_category}」に最も近いと判断されました。")

return "環境音分類器のデモが完了しました。"

# 環境音分類器のプロジェクトを実行
environmental_sound_classifier()

```

## プロジェクト 2: リアルタイム騒音モニタリングシステム

マイクから入力された音声のリアルタイム分析を行い、騒音レベルをモニタリングするシステムを開発します。

```

def realtime_noise_monitoring():
    """リアルタイム騒音モニタリングシステムのミニプロジェクト"""

    # ※注意: Colaboratoryでは実際のマイク入力はできないため、
    # ファイルから読み込んだデータでシミュレーションします

    print("リアルタイム騒音モニタリングシステムのシミュレーション")

```

```
print("※実際のリアルタイム処理はローカル環境で実行する必要があります")  
  
# サンプル音声ファイルのダウンロード  
!wget -q -O ambient_noise.wav  
"https://freesound.org/data/previews/517/517742_1923344-1q.mp3"  
  
# 音声データの読み込み  
y, sr = librosa.load('ambient_noise.wav', sr=None)  
  
# フレームサイズとホップ長の設定  
frame_length = 1024  
hop_length = 512 # 50%オーバーラップ  
  
# シミュレーション用のバッファ（実際のリアルタイム処理では、マイク入力のバッファを使用）  
buffer_duration = 30 # 秒  
buffer_size = int(buffer_duration * sr)  
buffer = y[:buffer_size]  
  
# 時間軸の作成  
time = np.linspace(0, buffer_duration, buffer_size)  
  
# プロットの初期化  
plt.figure(figsize=(14, 10))  
  
# 1. 波形表示エリア  
plt.subplot(3, 1, 1)  
waveform_line, = plt.plot(time, buffer)  
plt.title('リアルタイム音声波形')  
plt.xlabel('時間 (秒)')  
plt.ylabel('振幅')  
plt.xlim(0, buffer_duration)  
plt.ylim(-1, 1)  
  
# 2. RMSレベル（音圧レベル）表示エリア  
plt.subplot(3, 1, 2)  
n_frames = (buffer_size - frame_length) // hop_length + 1  
frame_times = np.linspace(0, buffer_duration, n_frames)  
rms = librosa.feature.rms(y=buffer, frame_length=frame_length,  
hop_length=hop_length)[0]  
rms_db = 20 * np.log10(rms + 1e-8) # デシベルに変換（クリッピング防止のため小さな値を加算）  
rms_line, = plt.plot(frame_times, rms_db)  
plt.title('リアルタイムRMSレベル (dB)')  
plt.xlabel('時間 (秒)')  
plt.ylabel('RMS (dB)')  
plt.xlim(0, buffer_duration)  
plt.ylim(-60, 0)  
  
# 3. スペクトログラム表示エリア  
plt.subplot(3, 1, 3)  
spec = librosa.stft(buffer, n_fft=frame_length, hop_length=hop_length)  
spec_db = librosa.amplitude_to_db(np.abs(spec), ref=np.max)  
spec_img = plt.imshow(spec_db, aspect='auto', origin='lower',  
extent=[0, buffer_duration, 0, sr/2], cmap='viridis')
```

```
plt.colorbar(format='%+2.0f dB')
plt.title('リアルタイムスペクトログラム')
plt.xlabel('時間 (秒)')
plt.ylabel('周波数 (Hz)')
plt.ylim(0, 8000) # 8kHzまで表示

plt.tight_layout()
plt.show()

# リアルタイム処理のシミュレーション（フレームごとの処理）
print("\nリアルタイム処理のシミュレーション:")
print("フレームごとの音響特徴量を計算し、騒音レベルを評価")

# 分析用の特徴量を計算
n_frames = (len(buffer) - frame_length) // hop_length + 1

# 各フレームの特徴量を格納する配列
frame_features = {
    'time': frame_times,
    'rms': rms,
    'rms_db': rms_db,
    'zcr': np.zeros(n_frames),
    'spectral_centroid': np.zeros(n_frames),
    'spectral_bandwidth': np.zeros(n_frames),
    'spectral_rolloff': np.zeros(n_frames),
    'spectral_flatness': np.zeros(n_frames)
}

# フレームごとの特徴量計算
for i in range(n_frames):
    frame_start = i * hop_length
    frame_end = frame_start + frame_length
    if frame_end <= len(buffer):
        frame = buffer[frame_start:frame_end]

        # ゼロ交差率
        frame_features['zcr'][i] =
np.mean(librosa.feature.zero_crossing_rate(frame)[0])

        # スペクトル特徴量（フレームごとに直接計算）
        frame_spec = np.abs(np.fft.rfft(frame * np.hanning(len(frame))))
        freqs = np.fft.rfftfreq(len(frame), 1/sr)

        # スペクトル重心
        if np.sum(frame_spec) > 0:
            frame_features['spectral_centroid'][i] = np.sum(freqs *
frame_spec) / np.sum(frame_spec)

        # スペクトル帯域幅（シンプルな実装）
        if np.sum(frame_spec) > 0 and frame_features['spectral_centroid'][i] >
0:
            frame_features['spectral_bandwidth'][i] = np.sqrt(
                np.sum(((freqs - frame_features['spectral_centroid'][i]) ** 2) *
frame_spec) / np.sum(frame_spec))
```

```
)  
  
    # スペクトルロールオフ (85%)  
    cumsum = np.cumsum(frame_spec)  
    rolloff_point = 0.85 * cumsum[-1]  
    rolloff_idx = np.where(cumsum >= rolloff_point)[0][0]  
    frame_features['spectral_rolloff'][i] = freqs[rolloff_idx]  
  
    # スペクトルフラットネス (簡易計算 )  
    if np.product(frame_spec + 1e-10) > 0 and np.mean(frame_spec) > 0:  
        frame_features['spectral_flatness'][i] =  
        np.exp(np.mean(np.log(frame_spec + 1e-10))) / np.mean(frame_spec)  
  
    # 特徴量の可視化  
    plt.figure(figsize=(14, 12))  
  
    # RMSレベル (音圧レベル)  
    plt.subplot(5, 1, 1)  
    plt.plot(frame_features['time'], frame_features['rms_db'])  
    plt.title('音圧レベル (dB)')  
    plt.xlabel('時間 (秒)')  
    plt.ylabel('RMS (dB)')  
  
    # ゼロ交差率  
    plt.subplot(5, 1, 2)  
    plt.plot(frame_features['time'], frame_features['zcr'])  
    plt.title('ゼロ交差率')  
    plt.xlabel('時間 (秒)')  
    plt.ylabel('ZCR')  
  
    # スペクトル重心  
    plt.subplot(5, 1, 3)  
    plt.plot(frame_features['time'], frame_features['spectral_centroid'])  
    plt.title('スペクトル重心 (Hz)')  
    plt.xlabel('時間 (秒)')  
    plt.ylabel('周波数 (Hz)')  
  
    # スペクトル帯域幅  
    plt.subplot(5, 1, 4)  
    plt.plot(frame_features['time'], frame_features['spectral_bandwidth'])  
    plt.title('スペクトル帯域幅 (Hz)')  
    plt.xlabel('時間 (秒)')  
    plt.ylabel('帯域幅 (Hz)')  
  
    # スペクトルフラットネス  
    plt.subplot(5, 1, 5)  
    plt.plot(frame_features['time'], frame_features['spectral_flatness'])  
    plt.title('スペクトルフラットネス')  
    plt.xlabel('時間 (秒)')  
    plt.ylabel('フラットネス')  
  
plt.tight_layout()  
plt.show()
```

```
# 騒音イベント検出のシミュレーション
print("\n騒音イベント検出:")

# RMSレベルに基づく閾値検出
threshold_db = -30 # 閾値 (デシベル)
noise_events = []
in_event = False
event_start = 0

for i, db_level in enumerate(frame_features['rms_db']):
    if not in_event and db_level > threshold_db:
        # イベント開始
        in_event = True
        event_start = frame_features['time'][i]
    elif in_event and db_level <= threshold_db:
        # イベント終了
        in_event = False
        event_end = frame_features['time'][i]
        # 持続時間が0.2秒以上のイベントのみ記録
        if event_end - event_start >= 0.2:
            noise_events.append((event_start, event_end))

    # 最後のイベントが終了していない場合
if in_event:
    noise_events.append((event_start, frame_features['time'][-1]))

# 検出されたイベントの表示
plt.figure(figsize=(14, 6))
plt.plot(frame_features['time'], frame_features['rms_db'])
plt.axhline(y=threshold_db, color='r', linestyle='--', label=f'閾値: {threshold_db} dB')

for start, end in noise_events:
    plt.axvspan(start, end, color='yellow', alpha=0.3)
    plt.text(start, -15, f'{start:.1f}s', fontsize=8, ha='left')

plt.title('騒音イベント検出')
plt.xlabel('時間 (秒)')
plt.ylabel('RMS (dB)')
plt.legend()
plt.tight_layout()
plt.show()

print(f"\n検出された騒音イベント数: {len(noise_events)}")
for i, (start, end) in enumerate(noise_events):
    duration = end - start
    print(f"イベント {i+1}: {start:.2f}秒 - {end:.2f}秒 (持続時間: {duration:.2f}秒)")

return "リアルタイム騒音モニタリングのシミュレーションが完了しました。"

# リアルタイム騒音モニタリングのプロジェクトを実行
realtime_noise_monitoring()
```

## プロジェクト 3: 音声強調と騒音抑制パイプライン

複数の騒音除去手法を組み合わせた、高度な音声強調パイプラインを開発します。

```
def audio_enhancement_pipeline():
    """音声強調と騒音抑制パイプラインのミニプロジェクト"""

    # サンプル音声ファイルのダウンロード
    !wget -q -O noisy_speech.wav
    "https://freesound.org/data/previews/573/573579_5834212-1q.mp3"

    # 音声の読み込み
    y, sr = librosa.load('noisy_speech.wav', sr=None)

    # 元の音声を再生
    plt.figure(figsize=(14, 4))
    librosa.display.waveform(y, sr=sr)
    plt.title('ノイズの多い音声 (元の音声)')
    plt.show()

    print("元の音声:")
    display(Audio(y, rate=sr))

    # 音声強調パイプラインの定義
    def enhance_audio(y, sr, pipeline_config):
        """音声強調パイプラインを適用する関数"""

        enhanced = y.copy()
        steps_output = {}
        steps_output['原音声'] = y

        # 各ステップを適用
        for step_name, step_config in pipeline_config.items():
            print(f"適用中: {step_name}")

            if step_config['type'] == 'highpass_filter':
                # ハイパスフィルタ (低周波ノイズの除去)
                cutoff = step_config['cutoff']
                order = step_config.get('order', 4)
                b, a = signal.butter(order, cutoff/(sr/2), 'high')
                enhanced = signal.filtfilt(b, a, enhanced)

            elif step_config['type'] == 'lowpass_filter':
                # ローパスフィルタ (高周波ノイズの除去)
                cutoff = step_config['cutoff']
                order = step_config.get('order', 4)
                b, a = signal.butter(order, cutoff/(sr/2), 'low')
                enhanced = signal.filtfilt(b, a, enhanced)

            elif step_config['type'] == 'spectral_subtraction':
                # スペクトル減算 (背景ノイズの除去)
                frame_length = step_config.get('frame_length', 2048)
                hop_length = step_config.get('hop_length', 512)
```

```
alpha = step_config.get('alpha', 2.0)

# STFTの計算
D = librosa.stft(enhanced, n_fft=frame_length,
hop_length=hop_length)

# ノイズプロファイルの推定（最初の数フレームを使用）
noise_frames = step_config.get('noise_frames', 10)
noise_profile = np.mean(np.abs(D[:, :noise_frames])) ** 2, axis=1)

# スペクトル減算の適用
mag = np.abs(D)
phase = np.angle(D)

# 各フレームにスペクトル減算を適用
mag_filtered = np.zeros_like(mag)
for i in range(mag.shape[1]):
    # ノイズプロファイルとのパワー差を計算
    power_diff = mag[:, i] ** 2 - alpha * noise_profile
    # 負の値をゼロにする
    power_diff = np.maximum(power_diff, 0)
    # パワーから振幅に戻す
    mag_filtered[:, i] = np.sqrt(power_diff)

# 位相を保持して逆変換
D_filtered = mag_filtered * np.exp(1j * phase)
enhanced = librosa.istft(D_filtered, hop_length=hop_length)

elif step_config['type'] == 'wiener_filter':
    # ウィナーフィルタ（定常ノイズの除去）
    frame_length = step_config.get('frame_length', 2048)
    hop_length = step_config.get('hop_length', 512)

    # STFTの計算
    D = librosa.stft(enhanced, n_fft=frame_length,
hop_length=hop_length)

    # ノイズパワーの推定（最初の数フレームを使用）
    noise_frames = step_config.get('noise_frames', 10)
    noise_power = np.mean(np.abs(D[:, :noise_frames])) ** 2, axis=1)

    # 各フレームにウィナーフィルタを適用
    mag = np.abs(D)
    phase = np.angle(D)

    # フィルタリング
    mag_filtered = np.zeros_like(mag)
    for i in range(mag.shape[1]):
        # 信号パワー
        signal_power = mag[:, i] ** 2
        # SNRの推定
        snr = np.maximum(signal_power / (noise_power + 1e-10) - 1, 0)
        # ウィナーフィルタ係数
        wiener_gain = snr / (snr + 1)
```

```
# フィルタ適用
mag_filtered[:, i] = mag[:, i] * wiener_gain

# 位相を保持して逆変換
D_filtered = mag_filtered * np.exp(1j * phase)
enhanced = librosa.istft(D_filtered, hop_length=hop_length)

elif step_config['type'] == 'normalization':
    # 音量の正規化
    target_rms = step_config.get('target_rms', 0.1)
    current_rms = np.sqrt(np.mean(enhanced ** 2))
    gain = target_rms / (current_rms + 1e-10)
    enhanced = enhanced * gain
    # クリッピング防止
    enhanced = np.clip(enhanced, -1.0, 1.0)

    # このステップの出力を保存
    steps_output[step_name] = enhanced.copy()

return enhanced, steps_output

# パイプラインの設定
pipeline_config = {
    '1. ハイパスフィルタ': {
        'type': 'highpass_filter',
        'cutoff': 80, # 80Hz以下をカット
        'order': 4
    },
    '2. ローパスフィルタ': {
        'type': 'lowpass_filter',
        'cutoff': 8000, # 8kHz以上をカット
        'order': 4
    },
    '3. スペクトル減算': {
        'type': 'spectral_subtraction',
        'frame_length': 2048,
        'hop_length': 512,
        'alpha': 2.0, # オーバーサブトラクションファクター
        'noise_frames': 15
    },
    '4. ウィナーフィルタ': {
        'type': 'wiener_filter',
        'frame_length': 2048,
        'hop_length': 512,
        'noise_frames': 15
    },
    '5. 音量正規化': {
        'type': 'normalization',
        'target_rms': 0.1
    }
}

# パイプラインを適用
enhanced, steps_output = enhance_audio(y, sr, pipeline_config)
```

```
# 結果の可視化
plt.figure(figsize=(14, 10))

# 波形の比較
plt.subplot(2, 1, 1)
plt.plot(np.linspace(0, len(y)/sr, len(y)), y, alpha=0.7, label='元の音声')
plt.plot(np.linspace(0, len(enhanced)/sr, len(enhanced)), enhanced, alpha=0.7,
label='強調後の音声')
plt.title('波形の比較')
plt.xlabel('時間 (秒)')
plt.ylabel('振幅')
plt.legend()

# スペクトログラムの比較
plt.subplot(2, 2, 3)
D_original = librosa.stft(y, n_fft=2048, hop_length=512)
D_db_original = librosa.amplitude_to_db(np.abs(D_original), ref=np.max)
librosa.display.specshow(D_db_original, sr=sr, x_axis='time', y_axis='hz',
cmap='viridis')
plt.colorbar(format='%+2.0f dB')
plt.title('元の音声のスペクトログラム')

plt.subplot(2, 2, 4)
D_enhanced = librosa.stft(enhanced, n_fft=2048, hop_length=512)
D_db_enhanced = librosa.amplitude_to_db(np.abs(D_enhanced), ref=np.max)
librosa.display.specshow(D_db_enhanced, sr=sr, x_axis='time', y_axis='hz',
cmap='viridis')
plt.colorbar(format='%+2.0f dB')
plt.title('強調後の音声のスペクトログラム')

plt.tight_layout()
plt.show()

# 各ステップの結果を再生
print("\n各ステップの出力結果:")
for step_name, audio in steps_output.items():
    print(f"\n{step_name}:")
    display(Audio(audio, rate=sr))

# 音質評価
print("\n音質の客観的評価:")

# SNR (信号対雑音比) の推定
def estimate_snr(clean, noisy):
    noise = noisy - clean
    signal_power = np.sum(clean ** 2)
    noise_power = np.sum(noise ** 2)
    if noise_power > 0:
        snr = 10 * np.log10(signal_power / noise_power)
    else:
        snr = float('inf')
    return snr
```

```
# この例では元の音声がすでにノイズを含むため、SNR評価は参考値
# 実際には清浄な参照音声が必要
# ここでは強調前後の差をノイズとみなして簡易評価
enhanced_truncated = enhanced[:len(y)] # 長さを揃える
estimated_snr = estimate_snr(enhanced_truncated, y)
print("推定SNR改善度: {estimated_snr:.2f} dB")

# スペクトル距離の計算
def spectral_distance(spec1, spec2):
    # 対数スペクトル距離
    log_spec1 = np.log(np.abs(spec1) + 1e-10)
    log_spec2 = np.log(np.abs(spec2) + 1e-10)
    return np.mean(np.abs(log_spec1 - log_spec2))

dist = spectral_distance(D_original, D_enhanced)
print("対数スペクトル距離: {dist:.4f}")

return "音声強調パイプラインが完了しました。"

# 音声強調パイプラインのプロジェクトを実行
audio_enhancement_pipeline()
```

## 16. よくある質問と回答

騒音解析と除去に関するよくある質問とその回答をまとめました。

### 基本的な質問

#### Q: 騒音解析と音声解析の違いは何ですか？

A: 騒音解析は主に不要な音（ノイズ）の特性を理解し、できれば除去することを目的としています。一方、音声解析は人間の発話など、特定の「意味を持つ」音声信号の分析に重点を置いています。両者は使用する技術に多くの共通点がありますが、目的と最適化する対象が異なります。

#### Q: サンプリングレートとビット深度の選択はどのように行うべきですか？

A: サンプリングレートは分析対象の最高周波数の少なくとも 2 倍（ナイキスト周波数）以上にする必要があります。人間の可聴域（20Hz～20kHz）をカバーするには 44.1kHz や 48kHz が一般的です。ビット深度は信号のダイナミックレンジを決定し、16 ビットで約 96dB のダイナミックレンジが得られます。高品質な音響分析には 24 ビットが推奨されることもあります。

#### Q: どのようなノイズの種類がありますか？

A: 主なノイズの種類は以下のとおりです：

- **ホワイトノイズ**: すべての周波数で均等なパワーを持つノイズ
- **ピンクノイズ**: 周波数が高くなるほどパワーが減少する ( $1/f$  特性)
- **ブラウンノイズ**: 周波数の 2 乗に反比例してパワーが減少する ( $1/f^2$  特性)
- **インパルスノイズ**: 急激な短時間の妨害（ポップノイズなど）
- **定常ノイズ**: 時間とともに統計的特性が変化しないノイズ（エアコンの稼働音など）

- **非定常ノイズ**: 時間とともに特性が変化するノイズ（交通騒音など）

技術的な質問

**Q: FFT サイズとホップ長はどのように選べばよいですか？**

A: FFT サイズは周波数分解能と時間分解能のトレードオフに影響します。大きな FFT サイズ（例：4096 点）は周波数分解能が高くなりますが、時間分解能は低下します。一般的には音声分析では 1024～2048 点がよく使われます。ホップ長は連続するフレーム間の重複を決定し、通常は FFT サイズの 1/4～1/2（75%～50% の重複）が適切です。

**Q: メルスペクトログラムとは何ですか？通常のスペクトログラムと何が違いますか？**

A: メルスペクトログラムは、人間の聴覚特性に基づいた非線形周波数スケール（メルスケール）を使用したスペクトログラムです。低周波では高い分解能を持ち、高周波になるほど分解能が低下するという人間の聴覚特性を模倣しています。これにより、機械学習モデルの入力として使う場合などに人間の知覚により近い表現が得られます。

**Q: スペクトル減算法とウィナーフィルタの違いは何ですか？**

A: どちらも騒音除去のための手法ですが、アプローチが異なります：

- **スペクトル減算法**: 信号のパワースペクトルからノイズのパワースペクトルを単純に引き算します。実装が簡単ですが、「ミュージカルノイズ」と呼ばれる人工的なアーティファクトが発生することがあります。
- **ウィナーフィルタ**: 信号とノイズの両方の統計的特性を考慮し、平均二乗誤差を最小化するフィルタです。より高品質な結果が得られますが、信号とノイズの特性の正確な推定が必要です。

**Q: 複数のマイクを使用する騒音除去（アレイ処理）の利点は何ですか？**

A: 複数マイクを使用すると、音源の空間的な情報を活用できます。主な利点：

1. **ビームフォーミング**: 特定の方向からの音を強調し、他の方向からの騒音を抑制できます
2. **空間フィルタリング**: 音源分離や騒音抑制の性能が向上します
3. **頑健性の向上**: 単一マイクの故障や位置による影響を軽減できます

実装に関する質問

**Q: Python 以外で騒音解析をするのに適した言語やツールはありますか？**

A: いくつかの選択肢があります：

- **MATLAB**: 信号処理に特化した多機能な環境で、Audio Toolbox など専用のツールボックスがあります
- **C/C++**: リアルタイム処理や組み込みシステムに適しており、FFTW、Eigen、OpenCV などのライブラリが利用できます
- **Audacity**: GUI ベースの音声編集ソフトウェアで、基本的な騒音除去機能が組み込まれています
- **Praat**: 音声学研究用のソフトウェアで、詳細な音響分析機能を備えています

**Q: リアルタイム処理を実装する際の注意点は何ですか？**

A: リアルタイム処理では以下の点に注意が必要です：

1. **レイテンシ:** 処理遅延を最小限に抑えるため、フレームサイズとバッファサイズの最適化が重要です
2. **計算効率:** 効率的なアルゴリズムを選択し、必要に応じてダウンサンプリングを検討します
3. **適応性:** 音響環境の変化に対応するため、適応型アルゴリズムが望ましいです
4. **メモリ使用:** メモリの使用量と割り当て・解放の頻度を最小限に抑えます
5. **コールバック処理:** 音声ストリームの処理は通常コールバック関数で実装します

Q: 騒音除去の性能をどのように評価すればよいですか？

A: 一般的な評価指標には以下があります：

- **SNR (Signal-to-Noise Ratio):** 信号対雑音比の改善度
- **PESQ (Perceptual Evaluation of Speech Quality):** 音声品質の知覚評価
- **STOI (Short-Time Objective Intelligibility):** 短時間客観的明瞭度
- **SDR (Signal-to-Distortion Ratio):** 信号対歪み比
- **主観評価:** リスニングテストによる MOS (平均オピニオンスコア)

実際の応用では、特定のタスク（音声認識など）のパフォーマンス改善度も重要な評価指標となります。

## 17. 次のステップ

騒音解析と除去の基本を学習した後、さらに知識を深めるための次のステップをご紹介します。

### 上級レベルの技術

#### 1. 深層学習ベースの騒音除去

- U-Net やオートエンコーダを用いた音声強調
- 敵対的生成ネットワーク (GAN) による高品質な音声再構成
- 深層強化学習を用いた適応的ノイズ除去

#### 2. ブラインド信号分離

- 独立成分分析 (ICA)
- 非負値行列因子分解 (NMF)
- ディープクラスタリング

#### 3. 多チャンネル処理技術

- マイクロホンアレイを用いたビームフォーミング
- 音源定位と追跡
- 適応型マルチチャンネルノイズキャンセレーション

### 応用プロジェクトのアイデア

#### 1. 個人用リアルタイムノイズキャンセリングアプリ

- スマートフォンのマイクを使用して周囲の騒音を分析
- ヘッドフォンで再生する音声にリアルタイムノイズキャンセリングを適用
- 使用環境に応じて自動的にフィルタパラメータを最適化

#### 2. 産業機械の異常検知システム

- 機械の通常稼働時の音響特性をモデル化
- リアルタイムで音響特徴を監視し、異常を検出
- 異常の種類を分類し、メンテナンスの必要性を予測

### 3. 環境騒音マッピングシステム

- 複数の測定点からの騒音データを収集
- 地理情報システム（GIS）と統合して騒音マップを作成
- 都市計画や騒音低減策の評価に活用

### 4. スマートホーム音響モニタリング

- 家庭内の異常音（ガラスの破損、火災警報など）を検出
- プライバシーを保護しながら重要な音響イベントのみを通知
- 赤ちゃんの泣き声分析や高齢者の転倒検知などの特殊機能

## おすすめの学習リソース

### 書籍

- "Digital Signal Processing: Principles, Algorithms and Applications" by Proakis & Manolakis
- "Fundamentals of Speech Recognition" by Rabiner & Juang
- "Speech Enhancement: Theory and Practice" by Loizou
- "Deep Learning for Audio Signal Processing" by Purwins et al.

### オンラインコース

- Coursera: "Audio Signal Processing for Music Applications"
- Udemy: "Digital Signal Processing (DSP) From Ground Up™ with MATLAB"
- edX: "Fundamentals of Digital Signal Processing"

### ツールとライブラリ

- **TensorFlow Audio**: 深層学習ベースの音声処理
- **PyTorch Audio**: 音声処理のための深層学習ツール
- **Asteroid**: 音源分離のための PyTorch ベースライブラリ
- **ODAS**: オープンソースの音源定位・追跡ライブラリ
- **Kaldi**: 音声認識のためのオープンソースツールキット

### 学術会議と論文

- IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)
- INTERSPEECH
- IEEE/ACM Transactions on Audio, Speech, and Language Processing
- arXiv.org の "Audio and Speech Processing" セクション

### コミュニティとフォーラム

- Stack Overflow の [python] + [audio] タグ
- Reddit の r/DSP (Digital Signal Processing)

- GitHub の音声処理プロジェクト
  - 音響学会の地域支部やミートアップ
- 

## まとめ

騒音解析と除去の世界は、基本的な信号処理の概念から最先端の深層学習手法まで、幅広い技術と応用を含む魅力的な分野です。この教材では、初心者が最短で中級レベルまで習得できるように、以下の内容を提供しました：

1. 騒音解析の基本概念と音の物理的特性
2. 時間領域と周波数領域での音声信号解析手法
3. デジタルフィルタの設計と実装
4. スペクトル減算法やウィナーフィルタなどの古典的な騒音除去技術
5. 実践的なミニプロジェクトによる応用例

この知識を基盤として、あなた自身の興味や必要に応じて特定の領域をさらに深く探求し、実際のアプリケーションやシステムの開発に活用してください。音響と信号処理の分野は常に進化しており、新しい手法や技術が登場し続けています。継続的な学習と実践を通じて、より高度な騒音解析と除去のスキルを身につけていきましょう。

最後に、この教材が皆さんの学習の一助となり、音響技術の可能性を広げるきっかけになることを願っています。質問やフィードバックがあれば、いつでもお気軽にお寄せください。

Happy coding and happy noise analysis!

---

## 付録: 便利なワンライナーとチートシート

### 基本操作

```
# 音声ファイルの読み込み
y, sr = librosa.load('audio.wav', sr=None) # 元のサンプリングレートを保持

# 音声の再生
display(Audio(y, rate=sr))

# 音声ファイルの保存
sf.write('output.wav', y, sr)

# リサンプリング
y_resampled = librosa.resample(y, orig_sr=sr, target_sr=16000)
```

### 時間領域処理

```
# RMS計算
rms = librosa.feature.rms(y=y)[0]
```

```
# ゼロ交差率
zcr = librosa.feature.zero_crossing_rate(y)[0]

# 自己相関
autocorr = librosa.autocorrelate(y)

# 無音区間の除去
y_trimmed, _ = librosa.effects.trim(y, top_db=20)
```

## 周波数解析

```
# FFT計算
X = np.fft.rfft(y)
freqs = np.fft.rfftfreq(len(y), 1/sr)

# スペクトログラム
D = librosa.stft(y, n_fft=2048, hop_length=512)
D_db = librosa.amplitude_to_db(np.abs(D), ref=np.max)

# メルスペクトログラム
mel_spec = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128)
mel_spec_db = librosa.power_to_db(mel_spec, ref=np.max)

# MFCC
mfcc = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)
```

## フィルタリング

```
# ローパスフィルタ
b, a = signal.butter(4, 1000/(sr/2), 'low')
y_lowpass = signal.filtfilt(b, a, y)

# ハイパスフィルタ
b, a = signal.butter(4, 500/(sr/2), 'high')
y_highpass = signal.filtfilt(b, a, y)

# バンドパスフィルタ
b, a = signal.butter(4, [500/(sr/2), 3000/(sr/2)], 'band')
y_bandpass = signal.filtfilt(b, a, y)

# メディアンフィルタ（ノイズ除去）
y_median = signal.medfilt(y, kernel_size=5)
```

## 可視化

```
# 波形プロット
plt.figure(figsize=(14, 5))
librosa.display.waveform(y, sr=sr)
plt.title('音声波形')
plt.show()

# スペクトログラムプロット
plt.figure(figsize=(14, 5))
librosa.display.specshow(D_db, sr=sr, x_axis='time', y_axis='hz')
plt.colorbar(format='%+2.0f dB')
plt.title('スペクトログラム')
plt.show()

# MFCCプロット
plt.figure(figsize=(14, 5))
librosa.display.specshow(mfcc, x_axis='time')
plt.colorbar()
plt.title('MFCC')
plt.show()
```

## ノイズ除去

```
# スペクトル減算法 ( 簡易版 )
def spectral_subtraction(y, sr, frame_length=2048, hop_length=512,
noise_frames=10):
    # STFT
    D = librosa.stft(y, n_fft=frame_length, hop_length=hop_length)

    # ノイズプロファイル推定
    noise_profile = np.mean(np.abs(D[:, :noise_frames])**2, axis=1)

    # スペクトル減算
    mag = np.abs(D)
    phase = np.angle(D)
    mag_filtered = np.maximum(mag**2 - 2*noise_profile.reshape(-1, 1), 0)**0.5

    # 逆STFT
    D_filtered = mag_filtered * np.exp(1j * phase)
    y_filtered = librosa.istft(D_filtered, hop_length=hop_length)

    return y_filtered
```