計算社会科学のための Modal 入門

— サーバーレスクラウドによる大規模計算実験ガイド —

河合 勝彦
名古屋市立大学大学院 経済学研究科
kkawai@econ.nagoya-cu.ac.jp

2026 年 2 月 15 日

概要

計算社会科学（Computational Social Science）においては、エージェントベースモデル、モンテカルロシミュレーション、大規模ネットワーク分析、自然言語処理など、多大な計算資源を必要とする手法が広く用いられている。しかし、従来のクラスタ計算環境やクラウドインフラの構築には高度なシステム管理知識が求められ、研究者が計算実験に集中することが困難であった。

本チュートリアルでは、サーバーレスクラウドプラットフォーム **Modal**（https://modal.com）を紹介し、計算社会科学分野の研究者がインフラ管理の負担なく大規模計算実験を実行するための方法を体系的に解説する。Modal は Python コードにデコレータを付与するだけでクラウド上の計算資源（CPU, GPU）を利用可能にし、数百から数千の並列タスクを容易に実行できる。

本チュートリアルでは、Modal の基本概念から始め、囚人のジレンマ・トーナメント、意見ダイナミクスモデル、シェリングの分居モデル、投票者モデル、ネットワーク分析、テキスト感情分析、SIR 感染症拡散モデルなど、計算社会科学に特有の応用例を豊富に示す。すべての例題は PEP 723 に準拠したインラインメタデータを用い、`uv run` コマンドで即座に実行可能な自己完結型スクリプトとして提供される。

目次

1 はじめに 3
1.1 計算社会科学と計算資源の課題 . . . 3
1.2 Modal とは何か . . . 3
1.3 本チュートリアルの構成 . . . 4

2 環境構築 4
2.1 前提条件 . . . 4
2.2 uv のインストール . . . 4
2.3 Modal アカウントの設定 . . . 4
2.4 チュートリアルのディレクトリ構成 . . . 5
2.5 PEP 723 によるインラインスクリプトメタデータ . . . 5

3 Modal の基本概念 6
3.1 App（アプリケーション） . . . 6
3.2 Function（関数） . . . 6
3.3 ローカルエントリポイント . . . 7
3.4 コンテナイメージ . . . 7
3.5 実行の流れ . . . 8

4 Modal の主要機能 8
4.1 並列実行 . . . 8
4.2 GPU サポート . . . 10
4.3 ストレージ . . . 10
4.4 Web エンドポイント . . . 12
4.5 スケジューリング . . . 12
4.6 シークレット管理 . . . 13

5 計算社会科学への応用 13
5.1 例題 1: Modal の基本 — Hello World . . . 13
5.2 例題 2: モンテカルロ法による並列計算 . . . 15
5.3 例題 3: 囚人のジレンマ・トーナメント . . . 17
5.4 例題 4: 意見ダイナミクスモデル . . . 21
5.5 例題 5: シェリングの分居モデル . . . 25
5.6 例題 6: 投票者モデル . . . 29
5.7 例題 7: 大規模ネットワーク分析 . . . 34
5.8 例題 8: GPU を活用したテキスト感情分析 . . . 38
5.9 例題 9: Volume によるパラメータスイープと結果永続化 . . . 41
5.10 例題 10: Web ダッシュボード . . . 46

5.11 例題 11: 定期データ収集 49

6 発展的なトピック 52
6.1 ライフサイクル関数 52
6.2 コールドスタートの管理 52
6.3 タイムアウトとリトライ 52
6.4 Sandbox — 動的コード実行 53
6.5 デプロイメント 53

7 性能評価 54
7.1 ベンチマーク 1: モンテカルロ法 54
7.2 ベンチマーク 2: パラメータスイープ（SIR モデル） 54
7.3 ベンチマーク 3: ネットワーク分析 55
7.4 考察: Modal を使うべき場面 55

8 まとめ 56

付録 A Modal の代替サービス 57
A.1 サーバーレス GPU / クラウドコンピュート 57
A.2 大手クラウドのマネージドサービス 57
A.3 用途別の選択指針 57

1　はじめに

1.1　計算社会科学と計算資源の課題

計算社会科学は、社会現象の理解のために計算機シミュレーション、大規模データ分析、機械学習などの計算手法を活用する学際的分野である [1]。代表的な研究手法として以下が挙げられる：

- **エージェントベースモデル（ABM）**：個々のエージェントの行動ルールからマクロな社会現象の創発を分析する
- **モンテカルロシミュレーション**：確率的過程を多数回繰り返し、統計的性質を推定する
- **ネットワーク分析**：社会ネットワークの構造と、その上での拡散過程を分析する
- **自然言語処理（NLP）**：大規模テキストデータから社会的意見・感情を抽出する
- **パラメータスイープ**：モデルパラメータを系統的に変化させ、結果の感度分析・相転移の検出を行う

これらの手法はいずれも大量の計算資源を必要とする。例えば、エージェントベースモデルのパラメータスイープでは、パラメータの組合せ数が容易に数千〜数万に達する。各シミュレーションの実行時間が数分であっても、逐次実行では数日〜数週間を要することがある。

従来、このような大規模計算には HPC（High Performance Computing）クラスタやクラウドサービス（AWS, GCP, Azure）が利用されてきたが、これらの環境構築・管理には以下の困難が伴う：

1. ジョブスケジューラ（SLURM, PBS 等）の設定と運用
2. Docker コンテナや仮想環境の構築
3. ネットワーク、ストレージ、セキュリティの設計
4. 課金管理とリソースの最適化
5. 再現可能な計算環境の維持

1.2　Modal とは何か

Modal[*1] は、Python 開発者向けのサーバーレスクラウドプラットフォームである。Modal の核心的な特徴は、**Python コードにデコレータを付与するだけでクラウド上の計算資源を利用できる**点にある。

従来のクラウドサービスとは異なり、Modal では以下が**不要**である：

- Dockerfile の記述
- Kubernetes クラスタの管理
- クラウドコンソールでのインスタンス設定
- ジョブスケジューラの設定
- SSH 接続やリモートサーバーの管理

[*1] https://modal.com

その代わり、研究者は通常の Python コードを書き、Modal のデコレータ（`@app.function()`）を付けるだけで、そのコードがクラウド上のコンテナで実行される。

1.3　本チュートリアルの構成

本チュートリアルは以下の構成をとる。第 2 節で開発環境の構築方法を説明し、第 3 節で Modal の基本概念と API を解説する。第 4 節で並列実行、GPU 活用、ストレージ、Web エンドポイント、スケジューリングなどの主要機能を詳述する。第 5 節で計算社会科学の具体的応用例を示す。第 6 節で発展的なトピックを扱い、第 7 節で Modal の性能をベンチマークにより定量評価する。第 8 節でまとめを述べる。

2　環境構築

2.1　前提条件

本チュートリアルの実行には以下が必要である：

- Python 3.11 以上
- uv（Python パッケージマネージャ）
- Modal アカウント（無料枠あり）
- インターネット接続

2.2　uv のインストール

uv[*2] は Rust で実装された高速な Python パッケージマネージャである。

```
# macOS / Linux
curl -LsSf https://astral.sh/uv/install.sh | sh

# Windows (PowerShell)
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

2.3　Modal アカウントの設定

Modal のアカウントを作成し、認証トークンを設定する。

```
# Modal CLIのインストールと認証
uv run --with modal -m modal setup
```

ブラウザが開き、Modal のダッシュボードで認証が完了する。認証情報はユーザーのホームディレクトリ以下の `.modal/` ディレクトリに保存される。

[*2] https://docs.astral.sh/uv/

2.4　チュートリアルのディレクトリ構成

本チュートリアルの付属ファイルは以下のディレクトリ構成をとる。任意の作業ディレクトリに展開して使用できる。

```
modal-tutorial/
  examples/
    01_hello_modal.py           # 基本: Hello World
    02_parallel_monte_carlo.py  # 並列モンテカルロ法
    03_prisoners_dilemma.py     # 囚人のジレンマ
    04_opinion_dynamics.py      # 意見ダイナミクス
    05_schelling_segregation.py # シェリングの分居モデル
    06_voter_model.py           # 投票者モデル
    07_network_analysis.py      # ネットワーク分析
    08_text_analysis_gpu.py     # GPU テキスト分析
    09_parameter_sweep_volume.py # パラメータスイープ
    10_web_dashboard.py         # Web ダッシュボード
    11_scheduled_collection.py  # 定期データ収集
  tutorial.tex                  # 本文書のソース
  tutorial.pdf                  # コンパイル済みPDF
```

各スクリプトは PEP 723 インラインメタデータにより自己完結しているため、特定のプロジェクト設定やディレクトリ構造に依存しない。スクリプト単体を任意のディレクトリにコピーして uv run で実行することも可能である。

```
# ディレクトリのルートから実行する場合
cd modal-tutorial
uv run examples/01_hello_modal.py

# スクリプト単体をコピーして実行する場合
cp examples/02_parallel_monte_carlo.py /tmp/
uv run /tmp/02_parallel_monte_carlo.py
```

2.5　PEP 723 によるインラインスクリプトメタデータ

本チュートリアルのすべての例題は、**PEP 723**（Inline Script Metadata）[*3] に準拠している。PEP 723 は、Python スクリプトファイル内に依存関係をインラインで宣言する仕組みである。

```
# /// script
# requires-python = ">=3.11"
# dependencies = [
#   "modal>=0.73",
#   "numpy>=1.26",
# ]
# ///
```

[*3] https://peps.python.org/pep-0723/

このメタデータブロックは TOML 形式で記述され、各行は # で始まるコメントとして書かれる。uv run コマンドはこのメタデータを自動検出し、一時的な仮想環境を作成して依存パッケージをインストールした上でスクリプトを実行する。

```
# PEP 723 メタデータ付きスクリプトの実行
uv run examples/01_hello_modal.py
```

PEP 723 の利点

PEP 723 を使うことで、各スクリプトが自己完結的になる。pyproject.toml や仮想環境を共有する必要がなく、個々の例題を独立して実行できる。これは再現可能な計算実験にとって理想的である。

3 Modal の基本概念

Modal の基本的なプログラミングモデルは、以下の 3 つの概念から成り立つ。

3.1 App（アプリケーション）

modal.App は Modal プログラムの最上位の組織単位である。すべての関数やクラスは App に属する。

```
import modal

app = modal.App("my-app")
```

App 名は Modal ダッシュボードでの識別に用いられる。一意な名前を付けることが推奨される。

3.2 Function（関数）

@app.function() デコレータを付与された関数は、クラウド上のコンテナで実行される。

```
@app.function()
def square(x: int) -> int:
    return x ** 2
```

この関数は以下の 3 つの方法で呼び出せる：

.remote() クラウド上で実行し、結果を返す
.local() ローカルマシンで実行する（テスト用）
.map() 複数の入力に対して並列実行する

```
result = square.remote(42)      # クラウド実行: 1764
result = square.local(42)       # ローカル実行: 1764
results = list(square.map([1, 2, 3, 4]))  # 並列実行: [1, 4, 9, 16]
```

3.3 ローカルエントリポイント

`@app.local_entrypoint()` はローカルマシンで実行されるエントリポイント関数を定義する。この関数の中から `.remote()` や `.map()` を呼び出してクラウド上の関数を起動する。

```python
@app.local_entrypoint()
def main():
    # ここはローカルで実行される
    result = square.remote(42)  # ここでクラウド呼び出し
    print(result)
```

実行は `modal run` コマンド（または `uv run` 経由）で行う：

```
uv run examples/01_hello_modal.py
```

`modal run` と `uv run` の関係

PEP 723 メタデータに `modal` を依存関係として宣言しておけば、`uv run script.py` は内部で `modal run script.py` と同等の処理を行う。Modal のローカルエントリポイントが自動的に検出され実行される。ただし、`modal serve`（開発サーバー）や `modal deploy`（デプロイ）を使う場合は明示的に Modal CLI を呼び出す必要がある：`uv run -m modal serve script.py`。

3.4 コンテナイメージ

Modal はコンテナ内でコードを実行する。`modal.Image` オブジェクトでコンテナイメージを定義する。

```python
# Debian ベースの軽量イメージに pip パッケージを追加
image = (
    modal.Image.debian_slim(python_version="3.11")
    .pip_install("numpy>=1.26", "pandas>=2.1", "scipy>=1.11")
    .apt_install("libgomp1")
)

@app.function(image=image)
def compute():
    import numpy as np
    return np.random.rand(100).mean()
```

イメージの各操作（`.pip_install()`, `.apt_install()` 等）はレイヤーとしてキャッシュされる。2 回目以降の実行では変更のないレイヤーは再構築されず、起動が高速化される。

イメージ定義のベストプラクティス

- 変更頻度の低いパッケージを先にインストールする（キャッシュヒット率が向上する）
- `python_version` を明示的に指定して再現性を確保する
- 関数内で `import` する（コンテナ内にのみパッケージが存在するため）

3.5 実行の流れ

Modal プログラムの実行の全体像は以下の通りである。

1. ローカルマシンで `uv run script.py` を実行
2. uv が PEP 723 メタデータを読み取り、依存パッケージ（`modal` 等）を含む一時環境を作成
3. Modal CLI がスクリプトを解析し、`@app.local_entrypoint()` を特定
4. ローカルエントリポイント関数が実行される
5. `.remote()` / `.map()` 呼び出しにより Modal クラウド上にコンテナが起動
6. コンテナイメージが構築（初回）またはキャッシュから復元
7. 関数がクラウド上で実行され、結果がローカルに返される

4 Modal の主要機能

4.1 並列実行

計算社会科学において最も重要な Modal の機能は**並列実行**（parallel execution）である。`.map()` メソッドを使うことで、1 つの関数を多数の入力に対して自動的に並列実行できる。

```python
@app.function(image=image)
def simulate(params: dict) -> dict:
    # 1回のシミュレーションを実行
    result = run_simulation(params)
    return result

@app.local_entrypoint()
def main():
    param_list = [{"alpha": a, "seed": s}
                  for a in [0.1, 0.2, 0.3]
                  for s in range(100)]

    # 300回のシミュレーションを自動並列実行
    results = list(simulate.map(param_list))
```

`.map()` は結果をジェネレータとして返すため、完了したものから順次処理できる。Modal は入力の数に応じて自動的にコンテナを起動し、スケールアウトする。

4.1.1 .starmap() — 複数引数の並列実行

複数の引数を持つ関数には .starmap() を使う。

```python
@app.function()
def add(x: int, y: int) -> int:
    return x + y

@app.local_entrypoint()
def main():
    pairs = [(1, 2), (3, 4), (5, 6)]
    results = list(add.starmap(pairs))  # [3, 7, 11]
```

4.1.2 .spawn() — 非同期実行

.spawn() は関数を非同期で起動し、FunctionHandle を返す。結果は .get() で後から取得できる。

```python
@app.local_entrypoint()
def main():
    # 3つの分析を同時に起動
    h1 = analyze_degree.spawn(network)
    h2 = analyze_clustering.spawn(network)
    h3 = analyze_paths.spawn(network)

    # 結果を収集
    r1 = h1.get()
    r2 = h2.get()
    r3 = h3.get()
```

4.1.3 並行数の制御

```python
# 最大50コンテナまでに制限
@app.function(concurrency_limit=50)
def process(item):
    pass

# 1コンテナで複数入力を同時処理
@app.function(allow_concurrent_inputs=10)
def process(item):
    pass
```

計算社会科学での並列実行の活用

パラメータスイープでは、パラメータの組合せごとに独立したシミュレーションを実行することが多い。例えば、10 個のパラメータ値 × 100 回の乱数シードで合計 1,000 回のシミュレーションが必要な場合、.map() を使えばこれらを自動的に並列化できる。逐次実行では数時間

かかる計算が、数分で完了する。

4.2　GPU サポート

Modal では gpu パラメータで GPU を要求できる。

```python
@app.function(gpu="T4")      # NVIDIA T4
def inference():
    pass

@app.function(gpu="A10G")    # NVIDIA A10G
def train_medium():
    pass

@app.function(gpu="A100")    # NVIDIA A100 (40GB)
def train_large():
    pass

@app.function(gpu="H100")    # NVIDIA H100
def train_frontier():
    pass
```

複数 GPU やメモリ指定も可能である：

```python
import modal

# A100 80GB を2枚
@app.function(gpu=modal.gpu.A100(count=2, size="80GB"))
def distributed_training():
    pass
```

主要な GPU タイプを表 1 に示す。

表 1　Modal で利用可能な主要 GPU タイプ

GPU	VRAM	主な用途
T4	16 GB	推論、軽量な訓練
L4	24 GB	効率的な推論
A10G	24 GB	汎用（訓練・推論）
A100	40/80 GB	大規模訓練
H100	80 GB	最新世代の高速訓練

4.3　ストレージ

Modal は 3 種類のストレージプリミティブを提供する。

4.3.1 Volume — 高性能永続ストレージ

`modal.Volume` は高性能な永続ファイルシステムである。シミュレーション結果の保存に適している。

```python
vol = modal.Volume.from_name("my-results", create_if_missing=True)

@app.function(volumes={"/data": vol})
def save_result(data: dict):
    import json
    with open("/data/result.json", "w") as f:
        json.dump(data, f)
    vol.commit()  # 変更を永続化

@app.function(volumes={"/data": vol})
def load_result() -> dict:
    import json
    vol.reload()  # 最新データを取得
    with open("/data/result.json") as f:
        return json.load(f)
```

Volume の注意点

Volume への書き込み後は必ず `vol.commit()` を呼び出すこと。また、他のコンテナからの書き込みを反映するには `vol.reload()` を呼び出す必要がある。

4.3.2 NetworkFileSystem — 共有ファイルシステム

`modal.NetworkFileSystem` は POSIX 準拠の共有ストレージである。複数コンテナ間でのファイル共有が `commit()` / `reload()` なしで可能だが、Volume より低速である。

```python
nfs = modal.NetworkFileSystem.from_name("shared-fs", create_if_missing=True)

@app.function(network_file_systems={"/shared": nfs})
def process():
    with open("/shared/data.txt", "w") as f:
        f.write("共有データ")
```

4.3.3 CloudBucketMount — 外部クラウドストレージ

Amazon S3、Google Cloud Storage、Cloudflare R2 などの既存のクラウドストレージをマウントできる。

```python
bucket = modal.CloudBucketMount(
    bucket_name="my-s3-bucket",
    secret=modal.Secret.from_name("aws-credentials"),
    read_only=True,
)
```

```python

@app.function(volumes={"/s3": bucket})
def read_s3_data():
    with open("/s3/dataset.csv") as f:
        return f.read()
```

4.4 Web エンドポイント

Modal は関数を HTTP エンドポイントとして公開できる。シミュレーション結果のダッシュボードや API サーバーの構築に有用である。

4.4.1 シンプルな Web エンドポイント

```python
@app.function()
@modal.web_endpoint(method="GET")
def predict(x: float):
    result = compute(x)
    return {"input": x, "result": result}
```

4.4.2 FastAPI (ASGI) アプリケーション

より複雑な Web アプリケーションには FastAPI を使う：

```python
from fastapi import FastAPI

web_app = FastAPI()

@web_app.get("/simulate")
async def simulate(beta: float = 0.3, gamma: float = 0.1):
    result = run_sir(beta, gamma)
    return result

@app.function(image=my_image)
@modal.asgi_app()
def dashboard():
    return web_app
```

デプロイすると永続的な URL が付与される：

```
# 開発サーバー（ホットリロード付き）
uv run -m modal serve examples/10_web_dashboard.py

# 本番デプロイ
uv run -m modal deploy examples/10_web_dashboard.py
```

4.5 スケジューリング

`modal.Cron` または `modal.Period` で定期実行をスケジューリングできる。

```python
# cron式：毎日午前9時（UTC）に実行
@app.function(schedule=modal.Cron("0 9 * * *"))
def daily_collection():
    collect_data()

# 一定間隔：6時間ごとに実行
@app.function(schedule=modal.Period(hours=6))
def periodic_check():
    check_data()
```

スケジュール実行にはデプロイが必要である：

```
uv run -m modal deploy examples/11_scheduled_collection.py
```

4.6　シークレット管理

API キーやデータベース接続文字列などの機密情報は `modal.Secret` で安全に管理できる。

```
# CLI でシークレットを作成
modal secret create my-api-key API_KEY=sk-xxxxx
```

```python
@app.function(secrets=[modal.Secret.from_name("my-api-key")])
def call_api():
    import os
    api_key = os.environ["API_KEY"]
    # api_key を使って API を呼び出す
```

5　計算社会科学への応用

本節では、Modal を用いた計算社会科学の具体的な応用例を示す。すべての例題は `examples/` ディレクトリに PEP 723 準拠のスクリプトとして格納されている。

5.1　例題 1: Modal の基本 — Hello World

最初の例題として、Modal の基本的な使い方を示す。

例題 5.1 (Hello Modal)**.** クラウド上で関数を実行し、ローカルに結果を返す最小例。

Listing 1　Modal の基本例（`examples/01_hello_modal.py`）

```python
# /// script
# requires-python = ">=3.11"
# dependencies = [
#   "modal>=0.73",
# ]
# ///
```

```python
"""
例題1: Modal の基本 — Hello World
==================================
Modal の最も基本的な使い方を示す。
ローカルマシンからクラウド上の関数を呼び出す。

実行方法:
    uv run examples/01_hello_modal.py
"""

import modal

app = modal.App("hello-modal")


@app.function()
def greet(name: str) -> str:
    """クラウド上で実行される関数"""
    import platform

    hostname = platform.node()
    return f"こんにちは、{name}さん！ （実行ホスト: {hostname}）"


@app.local_entrypoint()
def main():
    """ローカルで実行されるエントリポイント"""
    # .remote() でクラウド上の関数を呼び出す
    result = greet.remote("計算社会科学者")
    print(result)

    # 複数の名前に対して並列実行
    names = ["Alice", "Bob", "Carol", "Dave"]
    for result in greet.map(names):
        print(result)
```

このスクリプトの実行方法：

```
uv run examples/01_hello_modal.py
```

実行結果を以下に示す：

```
$ uv run examples/01_hello_modal.py
こんにちは、計算社会科学者さん！ （実行ホスト: modal）
こんにちは、Aliceさん！ （実行ホスト: modal）
こんにちは、Bobさん！ （実行ホスト: modal）
こんにちは、Carolさん！ （実行ホスト: modal）
こんにちは、Daveさん！ （実行ホスト: modal）
```

ポイント：

- `@app.function()` でクラウド実行される関数を定義
- `.remote()` で単一呼び出し、`.map()` で並列呼び出し
- `@app.local_entrypoint()` がローカル側の制御フロー
- 実行ホストが `modal` となっており、クラウド上で関数が実行されたことが確認できる

5.2　例題 2: モンテカルロ法による並列計算

例題 5.2 (モンテカルロ法による π の推定)**.** 単位円への点の落下を利用したモンテカルロ法で π を推定する。N 個の点 (x_i, y_i) を $[-1, 1]^2$ 上に一様乱数で生成し、$x_i^2 + y_i^2 \le 1$ を満たす点の割合から π を推定する：

$$\hat{\pi} = 4 \cdot \frac{\#\{i : x_i^2 + y_i^2 \le 1\}}{N} \tag{1}$$

この例題では、10^8 個のサンプルを 100 バッチに分割し、各バッチを並列実行する。

Listing 2　モンテカルロ法（`examples/02_parallel_monte_carlo.py`）

```python
# /// script
# requires-python = ">=3.11"
# dependencies = [
#    "modal>=0.73",
#    "numpy>=1.26",
# ]
# ///
"""
例題2: モンテカルロ法による並列計算
====================================
モンテカルロ法で円周率を推定する。
Modal の .map() を使い、多数の試行を並列実行する。

実行方法:
    uv run examples/02_parallel_monte_carlo.py
"""

import modal

app = modal.App("monte-carlo-pi")

image = modal.Image.debian_slim(python_version="3.11").pip_install("numpy>=1.26")


@app.function(image=image)
def estimate_pi(args: tuple[int, int]) -> tuple[int, int]:
    """
    モンテカルロ法で円周率を推定する1バッチ。

    args: (seed, n_samples) のタプル
    returns: (ヒット数, サンプル数) のタプル
    """
```

```
    import numpy as np

    seed, n_samples = args
    rng = np.random.default_rng(seed)
    x = rng.uniform(-1, 1, n_samples)
    y = rng.uniform(-1, 1, n_samples)
    hits = int(np.sum(x**2 + y**2 <= 1.0))
    return (hits, n_samples)


@app.local_entrypoint()
def main():
    n_batches = 100  # 並列バッチ数
    samples_per_batch = 1_000_000  # バッチあたりのサンプル数

    args_list = [(seed, samples_per_batch) for seed in range(n_batches)]

    print(f"合計 {n_batches * samples_per_batch:,} サンプルを {n_batches} 並列バッチで
    実行...")

    total_hits = 0
    total_samples = 0
    for hits, n in estimate_pi.map(args_list):
        total_hits += hits
        total_samples += n

    pi_estimate = 4.0 * total_hits / total_samples
    print(f"推定値: π ≈ {pi_estimate:.8f}")
    print(f"誤差:   {abs(pi_estimate - 3.141592653589793):.2e}")
```

実行結果：

```
$ uv run examples/02_parallel_monte_carlo.py
合計 100,000,000 サンプルを 100 並列バッチで実行...
推定値: π ≈ 3.14165972
誤差:   6.71e-05
```

10^8 サンプルから得られた推定値は真値との誤差が 6.71×10^{-5} であり、十分な精度が確認できる。

計算量とスケーリング

逐次実行では 10^8 サンプルの処理に数十秒を要するが、100 並列で実行すれば各バッチ 10^6 サンプルとなり、全体の実行時間は大幅に短縮される。Modal の自動スケーリングにより、バッチ数を増やしても追加の設定は不要である。

5.3　例題 3: 囚人のジレンマ・トーナメント

例題 5.3 (繰り返し囚人のジレンマ)**.** Axelrod (1984) の古典的なトーナメントを再現する。2 人のプレイヤーが繰り返し囚人のジレンマゲームを行い、各戦略の総合成績を比較する。

利得行列は以下の通りである：

$$\begin{pmatrix} (R,R) & (S,T) \\ (T,S) & (P,P) \end{pmatrix} = \begin{pmatrix} (3,3) & (0,5) \\ (5,0) & (1,1) \end{pmatrix} \tag{2}$$

ここで $T > R > P > S$ かつ $2R > T + S$ が成立する。

本例題では 8 つの代表的戦略を実装し、全ペアの対戦を並列実行する：

AllC 常に協力する
AllD 常に裏切る
TFT しっぺ返し（Tit-for-Tat）— 初手は協力、以後は相手の前回の行動を模倣
GTFT 寛容なしっぺ返し — 裏切りに対して 10% の確率で許す
Pavlov Win-Stay, Lose-Shift — 前回の利得が高ければ同じ行動を繰り返す
Random 50% の確率で協力・裏切りをランダムに選択
Grudger 一度でも裏切られたら以後永久に裏切る
STFT 疑い深いしっぺ返し — 初手は裏切り

Listing 3　囚人のジレンマ・トーナメント（examples/03_prisoners_dilemma.py）

```python
# /// script
# requires-python = ">=3.11"
# dependencies = [
#   "modal>=0.73",
#   "numpy>=1.26",
# ]
# ///
"""
例題3: 囚人のジレンマ・トーナメント
====================================
Axelrod (1984) の繰り返し囚人のジレンマ・トーナメントを
Modal 上で並列実行する。

戦略ペアごとの対戦を .map() で並列化する。

実行方法:
    uv run examples/03_prisoners_dilemma.py
"""

import modal

app = modal.App("prisoners-dilemma")

image = modal.Image.debian_slim(python_version="3.11").pip_install("numpy>=1.26")
```

```

# --- 利得行列 ---
# (自分の行動, 相手の行動) -> 利得
# C = 協力, D = 裏切り
PAYOFF = {
    ("C", "C"): (3, 3),  # 相互協力
    ("C", "D"): (0, 5),  # 搾取される
    ("D", "C"): (5, 0),  # 搾取する
    ("D", "D"): (1, 1),  # 相互裏切り
}

N_ROUNDS = 200


def strategy_always_cooperate(my_history, opp_history):
    return "C"


def strategy_always_defect(my_history, opp_history):
    return "D"


def strategy_tit_for_tat(my_history, opp_history):
    if not opp_history:
        return "C"
    return opp_history[-1]


def strategy_generous_tft(my_history, opp_history):
    import random

    if not opp_history:
        return "C"
    if opp_history[-1] == "D":
        return "C" if random.random() < 0.1 else "D"
    return "C"


def strategy_pavlov(my_history, opp_history):
    if not my_history:
        return "C"
    if PAYOFF[(my_history[-1], opp_history[-1])][0] >= 3:
        return my_history[-1]
    return "D" if my_history[-1] == "C" else "C"


def strategy_random(my_history, opp_history):
    import random

```

```
    return random.choice(["C", "D"])


def strategy_grudger(my_history, opp_history):
    if "D" in opp_history:
        return "D"
    return "C"


def strategy_suspicious_tft(my_history, opp_history):
    if not opp_history:
        return "D"
    return opp_history[-1]


STRATEGIES = {
    "AllC": strategy_always_cooperate,
    "AllD": strategy_always_defect,
    "TFT": strategy_tit_for_tat,
    "GTFT": strategy_generous_tft,
    "Pavlov": strategy_pavlov,
    "Random": strategy_random,
    "Grudger": strategy_grudger,
    "STFT": strategy_suspicious_tft,
}


@app.function(image=image)
def play_match(matchup: dict) -> dict:
    """2つの戦略間の対戦を実行する"""
    import random

    random.seed(matchup["seed"])

    name_a = matchup["strategy_a"]
    name_b = matchup["strategy_b"]
    func_a = STRATEGIES[name_a]
    func_b = STRATEGIES[name_b]

    history_a, history_b = [], []
    score_a, score_b = 0, 0

    for _ in range(N_ROUNDS):
        action_a = func_a(history_a, history_b)
        action_b = func_b(history_b, history_a)
        pa, pb = PAYOFF[(action_a, action_b)]
        score_a += pa
        score_b += pb
        history_a.append(action_a)
```

```
        history_b.append(action_b)

    coop_rate_a = history_a.count("C") / N_ROUNDS
    coop_rate_b = history_b.count("C") / N_ROUNDS

    return {
        "strategy_a": name_a,
        "strategy_b": name_b,
        "score_a": score_a,
        "score_b": score_b,
        "coop_rate_a": coop_rate_a,
        "coop_rate_b": coop_rate_b,
    }


@app.local_entrypoint()
def main():
    strategy_names = list(STRATEGIES.keys())

    # 全ペアの対戦を生成
    matchups = []
    seed = 0
    for i, name_a in enumerate(strategy_names):
        for name_b in strategy_names[i:]:
            matchups.append({"strategy_a": name_a, "strategy_b": name_b, "seed": seed})
            seed += 1

    print(f"{len(strategy_names)} 戦略 × {len(matchups)} 対戦を並列実行...")

    # 合計スコアを集計
    total_scores: dict[str, int] = {name: 0 for name in strategy_names}
    match_count: dict[str, int] = {name: 0 for name in strategy_names}

    for result in play_match.map(matchups):
        total_scores[result["strategy_a"]] += result["score_a"]
        total_scores[result["strategy_b"]] += result["score_b"]
        match_count[result["strategy_a"]] += 1
        match_count[result["strategy_b"]] += 1

    # 結果の表示
    print(f"\n{'戦略':<12} {'合計得点':>10} {'対戦数':>6} {'平均得点':>10}")
    print("-" * 42)
    ranking = sorted(total_scores.items(), key=lambda x: x[1], reverse=True)
    for name, score in ranking:
        avg = score / match_count[name] if match_count[name] > 0 else 0
        print(f"{name:<12} {score:>10,} {match_count[name]:>6} {avg:>10.1f}")
```

Modal の `.map()` により、$\binom{8}{2} + 8 = 36$ 通りの対戦が自動的に並列化される。

実行結果：

```
$ uv run examples/03_prisoners_dilemma.py
8 戦略 × 36 対戦を並列実行...

戦略                    合計得点    対戦数      平均得点
------------------------------------------
GTFT                4,800      9      533.3
TFT                 4,744      9      527.1
Grudger             4,619      9      513.2
Pavlov              4,560      9      506.7
AllC                4,470      9      496.7
Random              3,773      9      419.2
AllD                3,480      9      386.7
STFT                3,387      9      376.3
```

寛容なしっぺ返し（GTFT）が最高成績を収め、次いで通常のしっぺ返し（TFT）が続いた。これは Axelrod (1984) の古典的結果と整合的であり、「協力的だが裏切りには対抗する」戦略が高い得点を得ることを示している。一方、常に裏切る AllD と疑い深い STFT は最下位付近に位置する。

5.4　例題 4: 意見ダイナミクスモデル

例題 5.4 (Deffuant-Weisbuch モデル)**.** 有界信頼モデル（Bounded Confidence Model）は、意見の差が閾値 ε 以内のエージェント同士のみが相互作用するモデルである。

各ステップで 2 人のエージェント i, j がランダムに選ばれ、以下の更新則に従う：

$$|o_i - o_j| < \varepsilon \implies \begin{cases} o_i \leftarrow o_i + \mu(o_j - o_i) \\ o_j \leftarrow o_j + \mu(o_i - o_j) \end{cases} \tag{3}$$

ここで $\varepsilon \in (0, 1]$ は信頼区間、$\mu \in (0, 0.5]$ は収束パラメータである。

ε が小さいほどクラスタ数が増え（意見の分極化）、ε が大きいほど合意形成が促進される。

この例題では ε と μ のパラメータスイープを Modal で並列実行し、意見分極化の相転移を観察する。

Listing 4　意見ダイナミクスモデル（`examples/04_opinion_dynamics.py`）

```python
# /// script
# requires-python = ">=3.11"
# dependencies = [
#   "modal>=0.73",
#   "numpy>=1.26",
# ]
# ///
"""
例題4: 意見ダイナミクスモデル（Deffuant-Weisbuchモデル）
============================================================
有界信頼モデル（Bounded Confidence Model）のパラメータスイープを
Modal で並列実行する。
```

```
13
14 閾値 ε（信頼区間）と収束パラメータ μ を変化させ、
15 合意形成パターンの相転移を観察する。
16
17 実行方法:
18     uv run examples/04_opinion_dynamics.py
19 """
20
21 import modal
22
23 app = modal.App("opinion-dynamics")
24
25 image = modal.Image.debian_slim(python_version="3.11").pip_install("numpy>=1.26")
26
27
28 @app.function(image=image, timeout=300)
29 def simulate_deffuant(params: dict) -> dict:
30     """
31     Deffuant-Weisbuch モデルの1回のシミュレーション。
32
33     params:
34         n_agents: エージェント数
35         epsilon: 信頼区間 (bounded confidence threshold)
36         mu: 収束パラメータ (0 < mu <= 0.5)
37         n_steps: ステップ数
38         seed: 乱数シード
39     """
40     import numpy as np
41
42     rng = np.random.default_rng(params["seed"])
43     n = params["n_agents"]
44     eps = params["epsilon"]
45     mu = params["mu"]
46     n_steps = params["n_steps"]
47
48     # 初期意見: [0, 1] の一様分布
49     opinions = rng.uniform(0, 1, n)
50
51     for _ in range(n_steps):
52         # ランダムにペアを選択
53         i, j = rng.integers(0, n, size=2)
54         if i == j:
55             continue
56         diff = abs(opinions[i] - opinions[j])
57         if diff < eps:
58             # 意見を近づける
59             opinions[i] += mu * (opinions[j] - opinions[i])
60             opinions[j] += mu * (opinions[i] - opinions[j])
61
```

```
    # クラスタ数を計算（近い意見をグループ化）
    sorted_ops = np.sort(opinions)
    gaps = np.diff(sorted_ops)
    n_clusters = int(np.sum(gaps > eps / 2)) + 1

    return {
        "epsilon": eps,
        "mu": mu,
        "seed": params["seed"],
        "n_clusters": n_clusters,
        "final_mean": float(np.mean(opinions)),
        "final_std": float(np.std(opinions)),
        "opinions_sample": sorted_ops[:: max(1, n // 20)].tolist(),
    }


@app.local_entrypoint()
def main():
    import json

    n_agents = 500
    n_steps = 500_000
    n_seeds = 10  # 各パラメータで10回繰り返し

    # パラメータグリッド
    epsilons = [0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.5]
    mus = [0.1, 0.3, 0.5]

    params_list = [
        {
            "n_agents": n_agents,
            "epsilon": eps,
            "mu": mu,
            "n_steps": n_steps,
            "seed": seed,
        }
        for eps in epsilons
        for mu in mus
        for seed in range(n_seeds)
    ]

    print(f"パラメータ組合せ: {len(epsilons)} × {len(mus)} × {n_seeds} = {len(
    params_list)} 回")
    print(f"エージェント数: {n_agents}, ステップ数: {n_steps:,}")
    print("並列実行中...")

    results = list(simulate_deffuant.map(params_list))

    # ε ごとの平均クラスタ数を集計
```

```
    from collections import defaultdict

    cluster_by_eps: dict[float, list[int]] = defaultdict(list)
    for r in results:
        cluster_by_eps[r["epsilon"]].append(r["n_clusters"])

    print(f"\n{'ε':>6}  {'平均クラスタ数':>14}  {'標準偏差':>8}")
    print("-" * 32)
    for eps in sorted(cluster_by_eps.keys()):
        vals = cluster_by_eps[eps]
        import numpy as np

        mean_c = np.mean(vals)
        std_c = np.std(vals)
        print(f"{eps:>6.2f}  {mean_c:>14.2f}  {std_c:>8.2f}")

    # 結果をJSON保存
    with open("opinion_dynamics_results.json", "w") as f:
        json.dump(results, f, indent=2, ensure_ascii=False)
    print(f"\n結果を opinion_dynamics_results.json に保存しました（{len(results)} 件）"
    )
```

実行結果：

```
$ uv run --with "modal>=0.73" --with "numpy>=1.26" \
    -m modal run examples/04_opinion_dynamics.py
パラメータ組合せ: 9 × 3 × 10 = 270 回
エージェント数: 500, ステップ数: 500,000
並列実行中...

     ε          平均クラスタ数       標準偏差
--------------------------------
  0.05           10.57       1.15
  0.10            5.33       0.75
  0.15            3.27       0.44
  0.20            2.73       0.51
  0.25            2.07       0.44
  0.30            1.70       0.78
  0.35            2.00       0.77
  0.40            1.57       0.72
  0.50            1.00       0.00
```

$\varepsilon = 0.05$ では平均 10.57 個のクラスタが形成され、意見の強い分極化が生じている。ε が増加するとクラスタ数は単調に減少し、$\varepsilon = 0.50$ では全エージェントが 1 つの合意に収束する。$\varepsilon \approx 0.25$ 付近で急激にクラスタ数が変化する相転移が観察され、これは理論的に知られている臨界値と整合する。

結果の解釈

$\varepsilon < 0.25$ 付近で急激にクラスタ数が増加する相転移が観察される。$\varepsilon > 0.5$ では通常 1 つのクラスタに合意する。この臨界値は理論的にも知られている結果と一致する。

5.5　例題 5: シェリングの分居モデル

例題 5.5 (Schelling の分居モデル)**.** Schelling (1971) の分居モデルは、個人レベルの軽い同質性選好がマクロレベルでの強い空間的分居（segregation）を生み出すことを示した古典的なエージェントベースモデルである。

$N \times N$ のグリッド上に 2 種類のエージェントが配置される。各エージェントは 8 近傍のうち同種の割合が閾値 θ を下回ると「不満」となり、空きセルにランダムに移動する。

θ が比較的低い値（例: $\theta = 0.3$）であっても、マクロレベルでは高度な分居が創発することが、このモデルの重要な知見である。

Listing 5　シェリングの分居モデル（examples/05_schelling_segregation.py）

```python
# /// script
# requires-python = ">=3.11"
# dependencies = [
#    "modal>=0.73",
#    "numpy>=1.26",
# ]
# ///
"""
例題5: シェリングの分居モデル
==============================
Schelling (1971) の分居モデル (Segregation Model) を
Modal で並列パラメータスイープする。

同質性選好の閾値を変化させ、マクロな分居パターンの創発を観察する。

実行方法:
    uv run examples/05_schelling_segregation.py
"""

import modal

app = modal.App("schelling-segregation")

image = modal.Image.debian_slim(python_version="3.11").pip_install("numpy>=1.26")


@app.function(image=image, timeout=300)
def simulate_schelling(params: dict) -> dict:
    """
    シェリングの分居モデルを実行する。
```

```
31
32      params:
33          grid_size: グリッドの一辺のサイズ
34          density: 初期の居住密度 (0-1)
35          threshold: 同種エージェントの最低割合閾値
36          n_steps: 最大ステップ数
37          seed: 乱数シード
38      """
39      import numpy as np
40
41      rng = np.random.default_rng(params["seed"])
42      size = params["grid_size"]
43      density = params["density"]
44      threshold = params["threshold"]
45      n_steps = params["n_steps"]
46
47      # グリッド初期化: 0=空, 1=タイプA, 2=タイプB
48      grid = np.zeros((size, size), dtype=int)
49      n_cells = size * size
50      n_occupied = int(n_cells * density)
51      n_a = n_occupied // 2
52      n_b = n_occupied - n_a
53
54      positions = rng.permutation(n_cells)
55      for idx in positions[:n_a]:
56          grid[idx // size, idx % size] = 1
57      for idx in positions[n_a : n_a + n_b]:
58          grid[idx // size, idx % size] = 2
59
60      def get_neighbors(r, c):
61          neighbors = []
62          for dr in [-1, 0, 1]:
63              for dc in [-1, 0, 1]:
64                  if dr == 0 and dc == 0:
65                      continue
66                  nr, nc = (r + dr) % size, (c + dc) % size
67                  neighbors.append(grid[nr, nc])
68          return neighbors
69
70      def is_happy(r, c):
71          agent_type = grid[r, c]
72          if agent_type == 0:
73              return True
74          neighbors = get_neighbors(r, c)
75          same = sum(1 for n in neighbors if n == agent_type)
76          total = sum(1 for n in neighbors if n != 0)
77          if total == 0:
78              return True
79          return same / total >= threshold
```

```
80
81      # シミュレーション実行
82      n_moved_total = 0
83      for step in range(n_steps):
84          unhappy = [(r, c) for r in range(size) for c in range(size) if grid[r, c] !=
         0 and not is_happy(r, c)]
85
86          if not unhappy:
87              break
88
89          empty = [(r, c) for r in range(size) for c in range(size) if grid[r, c] ==
        0]
90
91          if not empty:
92              break
93
94          rng.shuffle(unhappy)
95          rng.shuffle(empty)
96
97          n_moved = min(len(unhappy), len(empty))
98          for k in range(n_moved):
99              ur, uc = unhappy[k]
100             er, ec = empty[k]
101             grid[er, ec] = grid[ur, uc]
102             grid[ur, uc] = 0
103         n_moved_total += n_moved
104
105     # 分居指数 (Segregation Index) の計算
106     segregation_scores = []
107     for r in range(size):
108         for c in range(size):
109             if grid[r, c] == 0:
110                 continue
111             neighbors = get_neighbors(r, c)
112             same = sum(1 for n in neighbors if n == grid[r, c])
113             total = sum(1 for n in neighbors if n != 0)
114             if total > 0:
115                 segregation_scores.append(same / total)
116
117     seg_index = float(np.mean(segregation_scores)) if segregation_scores else 0.0
118     happy_ratio = sum(1 for r in range(size) for c in range(size) if grid[r, c] != 0
         and is_happy(r, c)) / max(
119         1, n_occupied
120     )
121
122     return {
123         "threshold": threshold,
124         "density": density,
125         "seed": params["seed"],
```

```
        "steps_used": step + 1,
        "total_moves": n_moved_total,
        "segregation_index": seg_index,
        "happy_ratio": happy_ratio,
    }


@app.local_entrypoint()
def main():
    grid_size = 50
    density = 0.9
    n_steps = 500
    n_seeds = 10

    thresholds = [0.1, 0.2, 0.3, 0.375, 0.4, 0.45, 0.5, 0.6, 0.7, 0.8]

    params_list = [
        {
            "grid_size": grid_size,
            "density": density,
            "threshold": t,
            "n_steps": n_steps,
            "seed": seed,
        }
        for t in thresholds
        for seed in range(n_seeds)
    ]

    print(f"グリッド: {grid_size}×{grid_size}, 密度: {density}")
    print(f"閾値 {len(thresholds)} 種 × {n_seeds} 試行 = {len(params_list)} シミュレーション")
    print("並列実行中...")

    results = list(simulate_schelling.map(params_list))

    # 閾値ごとの分居指数を集計
    from collections import defaultdict

    import numpy as np

    seg_by_threshold: dict[float, list[float]] = defaultdict(list)
    for r in results:
        seg_by_threshold[r["threshold"]].append(r["segregation_index"])

    print(f"\n{'閾値':>6}  {'分居指数(平均)':>14}  {'標準偏差':>8}  {'満足率(平均)':>12}")
    print("-" * 46)
    for t in sorted(seg_by_threshold.keys()):
        vals = seg_by_threshold[t]
```

```
        happy_vals = [r["happy_ratio"] for r in results if r["threshold"] == t]
        print(f"{t:>6.2f}  {np.mean(vals):>14.4f}  {np.std(vals):>8.4f}  {np.mean(
    happy_vals):>12.4f}")
```

実行結果：

```
$ uv run --with "modal>=0.73" --with "numpy>=1.26" \
    -m modal run examples/05_schelling_segregation.py
グリッド: 50×50, 密度: 0.9
閾値 10 種 × 10 試行 = 100 シミュレーション
並列実行中...

    閾値          分居指数(平均)        標準偏差          満足率(平均)
------------------------------------------------
  0.10          0.5085     0.0041         1.0000
  0.20          0.5692     0.0095         1.0000
  0.30          0.7518     0.0110         1.0000
  0.38          0.7737     0.0082         1.0000
  0.40          0.8295     0.0057         1.0000
  0.50          0.8741     0.0081         1.0000
  0.60          0.9748     0.0020         1.0000
  0.70          0.9954     0.0006         1.0000
  0.80          0.5569     0.0095         0.1584
```

注目すべきは、閾値 $\theta = 0.30$（各エージェントが近傍の 30% 以上が同種であることを求めるという比較的穏やかな選好）でも、分居指数が 0.75 に達する点である。$\theta = 0.70$ では分居指数が 0.9954 とほぼ完全な分居状態となる。一方 $\theta = 0.80$ では、要求が厳しすぎて満足できるセルが見つからず、満足率が 15.8% に急落し分居指数も低下する。これは Schelling (1971) の古典的知見—「軽い同質性選好がマクロレベルの強い分居を生む」—を定量的に確認するものである。

5.6　例題 6: 投票者モデル

例題 5.6 (ネットワーク上の投票者モデル)**.** 投票者モデル（Voter Model）は、各ノードがランダムに選んだ隣人の意見をコピーするというシンプルな社会的影響モデルである。

本例題では 3 種類のネットワーク構造を比較する：

1. **格子グラフ**：規則的な空間構造
2. **Erdős-Rényi ランダムグラフ**：均質なランダム構造
3. **Barabási-Albert スケールフリーグラフ**：次数分布がべき乗則に従う不均質な構造

ネットワーク構造が合意形成時間に与える影響を調べる。

Listing 6　投票者モデル（`examples/06_voter_model.py`）

```
# /// script
# requires-python = ">=3.11"
# dependencies = [
#   "modal>=0.73",
```

```
#   "numpy>=1.26",
# ]
# ///
"""
例題6: 投票者モデル (Voter Model)
==================================
ネットワーク上の投票者モデルを Modal で並列実行する。
ネットワーク構造（格子・ランダム・スケールフリー）が
合意形成時間に与える影響を調べる。

実行方法:
    uv run examples/06_voter_model.py
"""

import modal

app = modal.App("voter-model")

image = modal.Image.debian_slim(python_version="3.11").pip_install("numpy>=1.26")


def generate_lattice_edges(n: int, k: int = 4) -> list[tuple[int, int]]:
    """周期境界条件つき1次元格子グラフ（各ノードk近傍）"""
    edges = []
    for i in range(n):
        for j in range(1, k // 2 + 1):
            edges.append((i, (i + j) % n))
    return edges


def generate_random_edges(n: int, p: float, rng) -> list[tuple[int, int]]:
    """Erdős-Rényi ランダムグラフ"""
    edges = []
    for i in range(n):
        for j in range(i + 1, n):
            if rng.random() < p:
                edges.append((i, j))
    return edges


def generate_ba_edges(n: int, m: int, rng) -> list[tuple[int, int]]:
    """Barabási-Albert スケールフリーグラフ"""
    edges = []
    degree = [0] * n
    # 完全グラフで初期化
    for i in range(m + 1):
        for j in range(i + 1, m + 1):
            edges.append((i, j))
            degree[i] += 1
```

```
                degree[j] += 1
    # 優先的接続
    for new_node in range(m + 1, n):
        total_degree = sum(degree[:new_node])
        if total_degree == 0:
            targets = list(range(m))
        else:
            probs = [degree[i] / total_degree for i in range(new_node)]
            targets = list(rng.choice(new_node, size=m, replace=False, p=probs))
        for t in targets:
            edges.append((new_node, t))
            degree[new_node] += 1
            degree[t] += 1
    return edges


@app.function(image=image, timeout=600)
def simulate_voter(params: dict) -> dict:
    """
    投票者モデルの1回のシミュレーション。

    ルール：各ステップでランダムにノードを選び、
    ランダムな隣人の意見をコピーする。
    """
    import numpy as np

    rng = np.random.default_rng(params["seed"])
    n = params["n_agents"]
    network_type = params["network_type"]
    max_steps = params["max_steps"]

    # ネットワーク生成
    if network_type == "lattice":
        edges = generate_lattice_edges(n, k=4)
    elif network_type == "random":
        edges = generate_random_edges(n, p=4.0 / n, rng=rng)
    elif network_type == "scale_free":
        edges = generate_ba_edges(n, m=2, rng=rng)
    else:
        raise ValueError(f"Unknown network type: {network_type}")

    # 隣接リスト構築
    adj: dict[int, list[int]] = {i: [] for i in range(n)}
    for u, v in edges:
        adj[u].append(v)
        adj[v].append(u)

    # 初期意見：0 or 1（半々）
    opinions = np.zeros(n, dtype=int)
```

```
    opinions[rng.choice(n, size=n // 2, replace=False)] = 1

    # シミュレーション
    consensus_time = max_steps
    for step in range(1, max_steps + 1):
        node = rng.integers(0, n)
        neighbors = adj[node]
        if neighbors:
            neighbor = neighbors[rng.integers(0, len(neighbors))]
            opinions[node] = opinions[neighbor]

        # 合意チェック（一定間隔で）
        if step % n == 0:
            if np.all(opinions == opinions[0]):
                consensus_time = step
                break

    unique_opinions = len(set(opinions.tolist()))
    return {
        "network_type": network_type,
        "n_agents": n,
        "seed": params["seed"],
        "consensus_time": consensus_time,
        "converged": unique_opinions == 1,
        "final_fraction_1": float(np.mean(opinions)),
    }


@app.local_entrypoint()
def main():
    import numpy as np

    n_agents = 200
    max_steps = 2_000_000
    n_seeds = 20

    params_list = [
        {
            "n_agents": n_agents,
            "network_type": net_type,
            "max_steps": max_steps,
            "seed": seed,
        }
        for net_type in ["lattice", "random", "scale_free"]
        for seed in range(n_seeds)
    ]

    print(f"エージェント数: {n_agents}")
    print(f"ネットワーク3種 × {n_seeds} 試行 = {len(params_list)} シミュレーション")
```

```
    print("並列実行中...")

    results = list(simulate_voter.map(params_list))

    # ネットワーク種別ごとの集計
    from collections import defaultdict

    by_type: dict[str, list[dict]] = defaultdict(list)
    for r in results:
        by_type[r["network_type"]].append(r)

    print(f"\n{'ネットワーク':<12} {'合意率':>8} {'平均合意時間':>14} {'標準偏差':>10}")
    print("-" * 48)
    for net_type in ["lattice", "random", "scale_free"]:
        runs = by_type[net_type]
        converged = [r for r in runs if r["converged"]]
        conv_rate = len(converged) / len(runs)
        if converged:
            times = [r["consensus_time"] for r in converged]
            mean_t = np.mean(times)
            std_t = np.std(times)
        else:
            mean_t, std_t = float("nan"), float("nan")
        print(f"{net_type:<12} {conv_rate:>8.1%} {mean_t:>14,.0f} {std_t:>10,.0f}")
```

実行結果：

```
$ uv run --with "modal>=0.73" --with "numpy>=1.26" \
    -m modal run examples/06_voter_model.py
エージェント数: 200
ネットワーク3種 × 20 試行 = 60 シミュレーション
並列実行中...

ネットワーク              合意率          平均合意時間        標準偏差
--------------------------------------------------
lattice         100.0%         291,610     292,155
random           10.0%          40,700      16,300
scale_free      100.0%          15,660       7,398
```

スケールフリーネットワークは平均 15,660 ステップで合意に達し、格子グラフ（291,610 ステップ）の約 19 分の 1 である。ランダムグラフでは制限ステップ内での合意率が 10% にとどまっているが、合意した場合の時間は 40,700 ステップと中間的な値を示す。

ネットワーク構造と合意形成

スケールフリーネットワークではハブ（高次数ノード）の存在により合意形成が促進される傾向がある。一方、格子グラフでは空間的な局所性により合意形成に長い時間を要する。

5.7　例題 7: 大規模ネットワーク分析

例題 5.7 (ネットワーク指標の並列計算)**.** 大規模ネットワークの複数の構造的指標を `.spawn()` で同時に計算する。

計算する指標：

次数分布 各ノードの接続数の統計量

クラスタ係数 ノード v のクラスタ係数は、v の隣人同士が接続している割合：

$$C_v = \frac{2|\{(u,w) : u,w \in N(v), (u,w) \in E\}|}{k_v(k_v-1)} \tag{4}$$

平均経路長 任意の 2 ノード間の最短経路長の平均

Listing 7　ネットワーク分析（`examples/07_network_analysis.py`）

```python
# /// script
# requires-python = ">=3.11"
# dependencies = [
#    "modal>=0.73",
#    "numpy>=1.26",
# ]
# ///
"""
例題7: 大規模ネットワーク分析
==============================
複数のネットワーク指標（次数分布・クラスタ係数・平均経路長）を
Modal で並列計算する。

各指標の計算を独立した関数として定義し、
1つのネットワークに対して複数の分析を同時実行する。

実行方法:
    uv run examples/07_network_analysis.py
"""

import modal

app = modal.App("network-analysis")

image = modal.Image.debian_slim(python_version="3.11").pip_install("numpy>=1.26")


def generate_ba_network(n: int, m: int, seed: int) -> dict:
    """Barabási-Albert ネットワークを生成し隣接リストで返す"""
    import numpy as np

    rng = np.random.default_rng(seed)
    adj: dict[int, set[int]] = {i: set() for i in range(n)}
```

```
    # 初期完全グラフ
    for i in range(m + 1):
        for j in range(i + 1, m + 1):
            adj[i].add(j)
            adj[j].add(i)

    degree = [len(adj[i]) for i in range(n)]

    for new_node in range(m + 1, n):
        total = sum(degree[:new_node])
        if total == 0:
            continue
        probs = [degree[i] / total for i in range(new_node)]
        targets = rng.choice(new_node, size=m, replace=False, p=probs)
        for t in targets:
            adj[new_node].add(int(t))
            adj[int(t)].add(new_node)
            degree[new_node] += 1
            degree[int(t)] += 1

    # シリアライズ
    return {str(k): list(v) for k, v in adj.items()}


@app.function(image=image, timeout=300)
def compute_degree_distribution(network: dict) -> dict:
    """次数分布を計算する"""
    import numpy as np

    degrees = [len(neighbors) for neighbors in network.values()]
    deg_array = np.array(degrees)

    return {
        "metric": "degree_distribution",
        "mean_degree": float(np.mean(deg_array)),
        "std_degree": float(np.std(deg_array)),
        "max_degree": int(np.max(deg_array)),
        "min_degree": int(np.min(deg_array)),
    }


@app.function(image=image, timeout=300)
def compute_clustering_coefficient(network: dict) -> dict:
    """クラスタ係数を計算する"""
    n = len(network)
    coefficients = []

    for node_str, neighbors in network.items():
```

```
        k = len(neighbors)
        if k < 2:
            coefficients.append(0.0)
            continue
        # 隣人間のリンク数
        neighbor_set = set(neighbors)
        links = 0
        for i, ni in enumerate(neighbors):
            for nj in neighbors[i + 1 :]:
                if nj in set(network.get(str(ni), [])):
                    links += 1
        max_links = k * (k - 1) / 2
        coefficients.append(links / max_links)

    import numpy as np

    return {
        "metric": "clustering_coefficient",
        "mean_cc": float(np.mean(coefficients)),
        "std_cc": float(np.std(coefficients)),
    }


@app.function(image=image, timeout=600)
def compute_avg_path_length(network: dict) -> dict:
    """BFSで平均経路長を計算する（サンプリング）"""
    import numpy as np
    from collections import deque

    n = len(network)
    rng = np.random.default_rng(42)

    # 大規模ネットワークではサンプリング
    sample_size = min(100, n)
    sample_nodes = rng.choice(n, size=sample_size, replace=False)

    all_distances = []
    for source in sample_nodes:
        visited = {source: 0}
        queue = deque([source])
        while queue:
            node = queue.popleft()
            for neighbor in network.get(str(node), []):
                if neighbor not in visited:
                    visited[neighbor] = visited[node] + 1
                    queue.append(neighbor)
        all_distances.extend(visited.values())

    distances = [d for d in all_distances if d > 0]
```

```
    return {
        "metric": "average_path_length",
        "avg_path_length": float(np.mean(distances)) if distances else float("inf"),
        "max_path_length": int(np.max(distances)) if distances else 0,
        "n_sampled": sample_size,
    }


@app.local_entrypoint()
def main():
    n_nodes = 2000
    m = 3
    seed = 42

    print(f"Barabási-Albert ネットワーク生成中 (n={n_nodes}, m={m})...")
    network = generate_ba_network(n_nodes, m, seed)
    print(f"ノード数: {len(network)}, エッジ数: {sum(len(v) for v in network.values())
     // 2}")

    # 3つの分析を並列実行
    print("3つのネットワーク指標を並列計算中...")

    # spawn で非同期実行
    handle_degree = compute_degree_distribution.spawn(network)
    handle_cc = compute_clustering_coefficient.spawn(network)
    handle_path = compute_avg_path_length.spawn(network)

    # 結果を収集
    result_degree = handle_degree.get()
    result_cc = handle_cc.get()
    result_path = handle_path.get()

    print("\n=== ネットワーク分析結果 ===")
    print(f"平均次数:          {result_degree['mean_degree']:.2f} (±{result_degree['
    std_degree']:.2f})")
    print(f"最大次数:          {result_degree['max_degree']}")
    print(f"平均クラスタ係数: {result_cc['mean_cc']:.4f} (±{result_cc['std_cc']:.4f})"
    )
    print(f"平均経路長:        {result_path['avg_path_length']:.4f}")
    print(f"最大経路長:        {result_path['max_path_length']}")
```

この例題では `.spawn()` を使い、3 つの独立した分析関数を同時に起動する。`.map()` が同一関数の複数入力に対する並列化であるのに対し、`.spawn()` は**異なる関数**を並列実行するのに適している。

実行結果：

```
$ uv run --with "modal>=0.73" --with "numpy>=1.26" \
    --with "networkx>=3.2" \
    -m modal run examples/07_network_analysis.py
```

```
Barabási-Albert ネットワーク生成中 (n=2000, m=3)...
ノード数: 2000, エッジ数: 5994
3つのネットワーク指標を並列計算中...

=== ネットワーク分析結果 ===
平均次数:          5.99 (±7.80)
最大次数:          129
平均クラスタ係数: 0.0202 (±0.0715)
平均経路長:        3.7300
最大経路長:        6
```

Barabási-Albert モデルの特徴が明確に現れている。次数分布は高い標準偏差（7.80）を示し、最大次数 129 のハブが存在する不均質な構造を反映している。平均経路長 3.73 はスモールワールド性を示し、クラスタ係数 0.0202 は BA モデルに典型的な低い値である。

5.8　例題 8: GPU を活用したテキスト感情分析

例題 5.8 (Transformers による感情分析)**.** Modal の GPU サポートを活用し、BERT ベースの多言語感情分析モデルで大量のテキストデータを処理する。

`@app.cls()` と `@modal.enter()` を組み合わせ、モデルのロードをコンテナ起動時に 1 回だけ行い、推論リクエストを効率的に処理する。

Listing 8　GPU テキスト分析（`examples/08_text_analysis_gpu.py`）

```python
# /// script
# requires-python = ">=3.11"
# dependencies = [
#    "modal>=0.73",
# ]
# ///
"""
例題8: GPU を活用したテキスト感情分析
=====================================
Modal の GPU サポートを使い、Transformers モデルによる
大量テキストの感情分析を高速実行する。

コンテナのライフサイクル関数 (@modal.enter) でモデルを
事前ロードし、推論リクエストを効率的に処理する。

実行方法:
    uv run examples/08_text_analysis_gpu.py
"""

import modal

app = modal.App("text-analysis-gpu")

image = (
```

```
    modal.Image.debian_slim(python_version="3.11")
    .pip_install(
        "torch>=2.1",
        "transformers>=4.36",
        "sentencepiece>=0.1.99",
    )
)


@app.cls(image=image, gpu="T4", scaledown_window=120)
class SentimentAnalyzer:
    """GPU上で感情分析を実行するクラス"""

    @modal.enter()
    def load_model(self):
        """コンテナ起動時にモデルをロード（1回だけ実行）"""
        from transformers import pipeline

        self.classifier = pipeline(
            "sentiment-analysis",
            model="nlptown/bert-base-multilingual-uncased-sentiment",
            device=0,  # GPU使用
        )
        print("モデルロード完了")

    @modal.method()
    def analyze(self, texts: list[str]) -> list[dict]:
        """テキストのバッチを感情分析する"""
        results = self.classifier(texts, batch_size=32, truncation=True, max_length
    =512)
        return [{"text": t[:50], "label": r["label"], "score": r["score"]} for t, r
    in zip(texts, results)]


@app.local_entrypoint()
def main():
    # サンプルテキスト（社会調査の自由回答を模擬）
    sample_texts = [
        "この政策は非常に良いと思います。市民の生活が改善されるでしょう。",
        "公共交通機関の遅延が多すぎて困っています。改善してほしい。",
        "新しい公園ができて子供たちが喜んでいます。素晴らしい取り組みです。",
        "税金の使い方に疑問を感じます。もっと透明性が必要です。",
        "地域のボランティア活動が活発になり、コミュニティが強くなりました。",
        "道路の整備が遅れていて、通勤が大変です。",
        "図書館のサービスが充実していて、とても満足しています。",
        "ゴミの分別ルールが複雑すぎて分かりにくいです。",
        "防災訓練が定期的に行われていて安心感があります。",
        "医療費の負担が重く、将来が不安です。",
        "The new community center has brought people together in wonderful ways.",
```

```
        "Public safety has improved significantly in our neighborhood.",
        "The education system needs fundamental reform to meet modern challenges.",
        "Environmental pollution continues to be a major concern for residents.",
        "Local businesses are thriving thanks to supportive municipal policies.",
    ]

    # テキストを大量に複製して大規模処理をシミュレート
    texts_large = sample_texts * 20  # 300件

    # バッチに分割
    batch_size = 32
    batches = [texts_large[i : i + batch_size] for i in range(0, len(texts_large),
    batch_size)]

    print(f"合計 {len(texts_large)} テキストを {len(batches)} バッチで処理...")

    analyzer = SentimentAnalyzer()
    all_results = []
    for batch_result in analyzer.analyze.map(batches):
        all_results.extend(batch_result)

    # 結果の集計
    from collections import Counter

    label_counts = Counter(r["label"] for r in all_results)
    print(f"\n=== 感情分析結果 (n={len(all_results)}) ===")
    for label, count in sorted(label_counts.items()):
        pct = count / len(all_results) * 100
        print(f"  {label}: {count:>4} ({pct:.1f}%)")

    # サンプル出力
    print("\n--- サンプル結果 ---")
    for r in all_results[:5]:
        print(f"  [{r['label']}] (確信度: {r['score']:.3f}) {r['text']}...")
```

実行結果：

```
$ uv run --with "modal>=0.73" \
    -m modal run examples/08_text_analysis_gpu.py
合計 300 テキストを 10 バッチで処理...

=== 感情分析結果 (n=300) ===
  3 stars:  100 (33.3%)
  4 stars:  100 (33.3%)
  5 stars:  100 (33.3%)

--- サンプル結果 ---
  [4 stars] (確信度: 0.431) この政策は非常に良いと思います。市民の生活が...
  [3 stars] (確信度: 0.382) 公共交通機関の遅延が多すぎて困っています。改善...
  [5 stars] (確信度: 0.613) 新しい公園ができて子供たちが喜んでいます。素晴...
```

```
  [5 stars]（確信度: 0.511）税金の使い方に疑問を感じます。もっと透明性が...
  [5 stars]（確信度: 0.445）地域のボランティア活動が活発になり、コミュニ...
```

300 件のテキストが 10 バッチに分割され、GPU 上の BERT モデルで処理された。サンプルテキストは 15 種類を 20 回複製したため、各星数の分布が均等（各 33.3%）となっている。多言語モデルにより日本語・英語の両方のテキストが分析されている。

クラスベースの関数定義

`@app.cls()` と `@modal.enter()` の組合せは、重い初期化（モデルロード、DB 接続等）を 1 回だけ行い、以後の呼び出しではその結果を再利用するパターンである。機械学習モデルの推論パイプラインに特に有効である。

5.9　例題 9: Volume によるパラメータスイープと結果永続化

例題 5.9 (SIR モデルのパラメータスイープ)**.** SIR（Susceptible-Infected-Recovered）モデルは感染症の拡散を記述する基本的な区画モデルである。確率的 SIR モデルでは、各ステップで：

$$P(\mathrm{S} \to \mathrm{I}) = \beta \cdot \frac{I}{N} \tag{5}$$

$$P(\mathrm{I} \to \mathrm{R}) = \gamma \tag{6}$$

ここで β は感染率、γ は回復率、I は現在の感染者数、N は総人口である。

本例題では β をスイープし、各シミュレーション結果を Modal Volume に永続保存する。

Listing 9　SIR モデル・パラメータスイープ（`examples/09_parameter_sweep_volume.py`）

```python
# /// script
# requires-python = ">=3.11"
# dependencies = [
#    "modal>=0.73",
#    "numpy>=1.26",
# ]
# ///
"""
例題9: Volume を使ったパラメータスイープと結果永続化
=====================================================
Modal Volume に大規模パラメータスイープの結果を保存し、
後から集計・分析する。

SIRモデル（感染症拡散モデル）のパラメータスイープを行い、
各シミュレーション結果を Volume に保存する。

実行方法:
    uv run examples/09_parameter_sweep_volume.py
"""

import modal
```

```python

app = modal.App("parameter-sweep-sir")

image = modal.Image.debian_slim(python_version="3.11").pip_install("numpy>=1.26")

vol = modal.Volume.from_name("sir-results", create_if_missing=True)


@app.function(image=image, volumes={"/results": vol}, timeout=300)
def simulate_sir(params: dict) -> dict:
    """
    SIR (Susceptible-Infected-Recovered) モデルの確率的シミュレーション。

    params:
        n_agents: 人口
        beta: 感染率
        gamma: 回復率
        n_steps: ステップ数
        seed: 乱数シード
        run_id: 実行ID
    """
    import json

    import numpy as np

    rng = np.random.default_rng(params["seed"])
    n = params["n_agents"]
    beta = params["beta"]
    gamma = params["gamma"]
    n_steps = params["n_steps"]

    # 状態: 0=S, 1=I, 2=R
    state = np.zeros(n, dtype=int)
    # 初期感染者: 1%
    n_initial = max(1, n // 100)
    state[rng.choice(n, size=n_initial, replace=False)] = 1

    trajectory = []
    for step in range(n_steps):
        s_count = int(np.sum(state == 0))
        i_count = int(np.sum(state == 1))
        r_count = int(np.sum(state == 2))
        trajectory.append({"step": step, "S": s_count, "I": i_count, "R": r_count})

        if i_count == 0:
            break

        # 感染: S -> I (確率 beta * I/N)
        susceptible = np.where(state == 0)[0]
```

```
        if len(susceptible) > 0:
            infection_prob = beta * i_count / n
            new_infected = susceptible[rng.random(len(susceptible)) < infection_prob
    ]
            state[new_infected] = 1

        # 回復: I -> R (確率 gamma)
        infected = np.where(state == 1)[0]
        if len(infected) > 0:
            recovered = infected[rng.random(len(infected)) < gamma]
            state[recovered] = 2

    # 最終状態
    final_s = int(np.sum(state == 0))
    final_i = int(np.sum(state == 1))
    final_r = int(np.sum(state == 2))

    result = {
        "params": params,
        "final_S": final_s,
        "final_I": final_i,
        "final_R": final_r,
        "attack_rate": final_r / n,
        "peak_I": max(t["I"] for t in trajectory),
        "epidemic_duration": len(trajectory),
    }

    # Volume に保存
    run_id = params["run_id"]
    filepath = f"/results/run_{run_id:06d}.json"
    with open(filepath, "w") as f:
        json.dump(result, f, ensure_ascii=False)
    vol.commit()

    return result


@app.function(image=image, volumes={"/results": vol}, timeout=120)
def aggregate_results() -> dict:
    """Volume に保存された全結果を集計する"""
    import glob
    import json

    import numpy as np

    vol.reload()

    files = sorted(glob.glob("/results/run_*.json"))
    results = []
```

```
    for filepath in files:
        with open(filepath) as f:
            results.append(json.load(f))

    if not results:
        return {"error": "結果ファイルが見つかりません"}

    # beta ごとの集計
    from collections import defaultdict

    by_beta: dict[float, list[dict]] = defaultdict(list)
    for r in results:
        by_beta[r["params"]["beta"]].append(r)

    summary = {}
    for beta in sorted(by_beta.keys()):
        runs = by_beta[beta]
        attack_rates = [r["attack_rate"] for r in runs]
        peak_Is = [r["peak_I"] for r in runs]
        summary[str(beta)] = {
            "mean_attack_rate": float(np.mean(attack_rates)),
            "std_attack_rate": float(np.std(attack_rates)),
            "mean_peak_I": float(np.mean(peak_Is)),
            "n_runs": len(runs),
        }

    return summary


@app.local_entrypoint()
def main():
    n_agents = 10_000
    gamma = 0.1  # 回復率固定
    n_steps = 300
    n_seeds = 20

    # β（感染率）のスイープ
    betas = [0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5]

    params_list = []
    run_id = 0
    for beta in betas:
        for seed in range(n_seeds):
            params_list.append(
                {
                    "n_agents": n_agents,
                    "beta": beta,
                    "gamma": gamma,
                    "n_steps": n_steps,
```

```
                    "seed": seed,
                    "run_id": run_id,
                }
            )
            run_id += 1

    print(f"SIRモデル: N={n_agents:,}, γ={gamma}")
    print(f"β: {len(betas)} 種 × {n_seeds} 試行 = {len(params_list)} シミュレーション")
    print("並列実行中...")

    # 全シミュレーション実行
    results = list(simulate_sir.map(params_list))

    # Volume から集計
    print("\nVolume から結果を集計中...")
    summary = aggregate_results.remote()

    print(f"\n{'β':>6}  {'感染率(平均)':>12}  {'標準偏差':>8}  {'ピーク感染者':>12}")
    print("-" * 44)
    for beta_str in sorted(summary.keys(), key=float):
        s = summary[beta_str]
        print(
            f"{float(beta_str):>6.2f}  "
            f"{s['mean_attack_rate']:>12.4f}  "
            f"{s['std_attack_rate']:>8.4f}  "
            f"{s['mean_peak_I']:>12.0f}"
        )
```

実行結果：

```
$ uv run --with "modal>=0.73" --with "numpy>=1.26" \
    -m modal run examples/09_parameter_sweep_volume.py
SIRモデル: N=10,000, γ=0.1
β: 8 種 × 20 試行 = 160 シミュレーション
並列実行中...

     β        感染率(平均)       標準偏差          ピーク感染者
--------------------------------------------
  0.05        0.0191     0.0021          100
  0.10        0.0784     0.0213          103
  0.15        0.4972     0.0244          450
  0.20        0.7430     0.0100         1162
  0.25        0.8612     0.0076         1849
  0.30        0.9229     0.0041         2478
  0.40        0.9747     0.0018         3526
  0.50        0.9917     0.0009         4262
```

$\beta = 0.10$ から $\beta = 0.15$ の間で感染率が 0.078 から 0.497 へ急激に上昇しており、エピデミック閾値（$\beta/\gamma = 1$、すなわち $\beta \approx 0.1$）付近での相転移が明確に確認できる。$\beta = 0.50$ ではほぼ全人口

（99.2%）が感染する大流行となる。この結果は Kermack-McKendrick の SIR 理論と整合的である。

Volume の活用パターン

シミュレーション結果を Volume に保存しておけば、後から別のスクリプトで集計・可視化を行える。この「実行と分析の分離」は大規模計算実験で有用なパターンである。

5.10　例題 10: Web ダッシュボード

例題 5.10 (シミュレーション結果の Web 公開)**.** Modal の ASGI サポートを使い、FastAPI でシミュレーション結果のダッシュボードを構築する。パラメータを URL クエリで指定し、リアルタイムにシミュレーションを実行・表示する。

Listing 10　Web ダッシュボード（`examples/10_web_dashboard.py`）

```python
# /// script
# requires-python = ">=3.11"
# dependencies = [
#   "modal>=0.73",
#   "fastapi>=0.104",
# ]
# ///
"""
例題10: Web ダッシュボードによるシミュレーション結果の公開
============================================================
Modal の Web エンドポイント機能を使い、
シミュレーション結果を REST API として公開する。

FastAPI (ASGI) を使ったダッシュボードを構築する。

実行方法（開発サーバー）:
    uv run -m modal serve examples/10_web_dashboard.py

デプロイ:
    uv run -m modal deploy examples/10_web_dashboard.py
"""

import modal

app = modal.App("simulation-dashboard")

image = modal.Image.debian_slim(python_version="3.11").pip_install(
    "fastapi>=0.104",
    "numpy>=1.26",
)


def run_quick_simulation(n_agents: int, beta: float, gamma: float, steps: int, seed:
```

```
     int) -> dict:
    """SIRモデルの簡易シミュレーション"""
    import numpy as np

    rng = np.random.default_rng(seed)
    state = np.zeros(n_agents, dtype=int)
    state[rng.choice(n_agents, size=max(1, n_agents // 100), replace=False)] = 1

    trajectory = []
    for step in range(steps):
        s = int(np.sum(state == 0))
        i = int(np.sum(state == 1))
        r = int(np.sum(state == 2))
        trajectory.append({"step": step, "S": s, "I": i, "R": r})

        if i == 0:
            break

        susceptible = np.where(state == 0)[0]
        if len(susceptible) > 0:
            prob = beta * i / n_agents
            state[susceptible[rng.random(len(susceptible)) < prob]] = 1

        infected = np.where(state == 1)[0]
        if len(infected) > 0:
            state[infected[rng.random(len(infected)) < gamma]] = 2

    return {
        "params": {"n_agents": n_agents, "beta": beta, "gamma": gamma, "seed": seed
    },
        "trajectory": trajectory,
        "final_attack_rate": int(np.sum(state == 2)) / n_agents,
    }


@app.function(image=image)
@modal.asgi_app()
def dashboard():
    """FastAPI ダッシュボード"""
    from fastapi import FastAPI, Query
    from fastapi.responses import HTMLResponse

    web_app = FastAPI(title="SIRモデル・シミュレーション・ダッシュボード")

    @web_app.get("/", response_class=HTMLResponse)
    async def index():
        return """
        <html>
        <head><title>SIRモデル・ダッシュボード</title></head>
```

```
        <body>
            <h1>SIRモデル・シミュレーション・ダッシュボード</h1>
            <p>APIエンドポイント:</p>
            <ul>
                <li><a href="/simulate?beta=0.3&gamma=0.1">/simulate</a> -
                    シミュレーション実行</li>
                <li><a href="/sweep?beta_min=0.1&beta_max=0.5&n_points=5">/sweep</a>
 -
                    パラメータスイープ</li>
                <li><a href="/docs">/docs</a> - API ドキュメント</li>
            </ul>
        </body>
        </html>
        """

    @web_app.get("/simulate")
    async def simulate(
        n_agents: int = Query(1000, description="人口"),
        beta: float = Query(0.3, description="感染率"),
        gamma: float = Query(0.1, description="回復率"),
        steps: int = Query(200, description="ステップ数"),
        seed: int = Query(42, description="乱数シード"),
    ):
        result = run_quick_simulation(n_agents, beta, gamma, steps, seed)
        return result

    @web_app.get("/sweep")
    async def sweep(
        n_agents: int = Query(1000),
        beta_min: float = Query(0.1),
        beta_max: float = Query(0.5),
        n_points: int = Query(10),
        gamma: float = Query(0.1),
    ):
        import numpy as np

        betas = np.linspace(beta_min, beta_max, n_points).tolist()
        results = []
        for beta in betas:
            r = run_quick_simulation(n_agents, beta, gamma, 200, 42)
            results.append(
                {
                    "beta": beta,
                    "final_attack_rate": r["final_attack_rate"],
                }
            )
        return {"sweep_results": results}

    return web_app
```

デプロイすると永続的な URL が発行され、ブラウザからシミュレーション結果にアクセスできる。これにより、研究成果のインタラクティブな公開が可能になる。

```
$ uv run -m modal deploy examples/10_web_dashboard.py
✓ Created objects.
├──   Created function SIRDashboard.*.wsgi
└──   Created web function dashboard =>
      https://katzkawai--sir-dashboard-dashboard.modal.run
✓ App deployed!
```

デプロイ後、表示された URL にブラウザでアクセスすると、SIR モデルのパラメータ（β, γ, N）をクエリパラメータで指定してリアルタイムにシミュレーションを実行できる。

5.11　例題 11: 定期データ収集

例題 5.11 (スケジュール実行による定期収集)**.** Modal の Cron スケジューリングを使い、公開 API から定期的にデータを収集するパイプラインを構築する。本例題では Wikipedia API から最近の編集情報を 6 時間ごとに収集する。

Listing 11　定期データ収集（examples/11_scheduled_collection.py）

```python
# /// script
# requires-python = ">=3.11"
# dependencies = [
#   "modal>=0.73",
# ]
# ///
"""
例題11: スケジュール実行による定期データ収集
==========================================
Modal の Cron スケジューリング機能を使い、
定期的なデータ収集パイプラインを構築する。

デプロイ後、指定したスケジュールで自動実行される。

デプロイ方法:
    uv run -m modal deploy examples/11_scheduled_collection.py
"""

import modal

app = modal.App("scheduled-collector")

image = modal.Image.debian_slim(python_version="3.11").pip_install(
    "requests>=2.31",
)

vol = modal.Volume.from_name("collected-data", create_if_missing=True)

```

```python

@app.function(
    image=image,
    volumes={"/data": vol},
    schedule=modal.Cron("0 */6 * * *"),  # 6時間ごと
    timeout=300,
)
def collect_public_data():
    """
    公開APIからデータを定期収集する。

    例: Wikipedia の最近の更新を取得。
    """
    import json
    from datetime import datetime, timezone

    import requests

    timestamp = datetime.now(timezone.utc).strftime("%Y%m%d_%H%M%S")

    # Wikipedia API: 最近の変更
    url = "https://en.wikipedia.org/w/api.php"
    params = {
        "action": "query",
        "list": "recentchanges",
        "rcnamespace": "0",  # メイン名前空間のみ
        "rclimit": "50",
        "rcprop": "title|timestamp|user|sizes|comment",
        "format": "json",
    }

    headers = {"User-Agent": "ModalTutorialBot/1.0 (educational project)"}
    response = requests.get(url, params=params, headers=headers, timeout=30)
    response.raise_for_status()

    try:
        data = response.json()
    except requests.exceptions.JSONDecodeError:
        # フォールバック: APIがJSON以外を返した場合
        data = {"query": {"recentchanges": []}}
        print(f"警告: APIレスポンスのパースに失敗（status={response.status_code}）")

    changes = data.get("query", {}).get("recentchanges", [])

    # 保存
    filepath = f"/data/wikipedia_changes_{timestamp}.json"
    with open(filepath, "w") as f:
        json.dump(
            {
```

```
                "collected_at": timestamp,
                "n_changes": len(changes),
                "changes": changes,
            },
            f,
            ensure_ascii=False,
            indent=2,
        )
    vol.commit()

    print(f"[{timestamp}] {len(changes)} 件の変更を収集・保存しました: {filepath}")
    return {"timestamp": timestamp, "n_changes": len(changes)}


@app.function(image=image, volumes={"/data": vol})
def list_collected_files() -> list[str]:
    """収集済みファイルの一覧を返す"""
    import glob

    vol.reload()
    files = sorted(glob.glob("/data/wikipedia_changes_*.json"))
    return files


@app.local_entrypoint()
def main():
    # 手動で1回実行してテスト
    print("データ収集テスト実行...")
    result = collect_public_data.remote()
    print(f"結果: {result}")

    files = list_collected_files.remote()
    print(f"\n保存済みファイル数: {len(files)}")
    for f in files[-5:]:
        print(f"  {f}")
```

手動で 1 回実行してテストした結果：

```
$ uv run --with "modal>=0.73" \
    -m modal run examples/11_scheduled_collection.py
データ収集テスト実行...
[20260214_094024] 50 件の変更を収集・保存しました:
  /data/wikipedia_changes_20260214_094024.json
結果: {'timestamp': '20260214_094024', 'n_changes': 50}

保存済みファイル数: 2
  /data/wikipedia_changes_20260214_092738.json
  /data/wikipedia_changes_20260214_094024.json
```

Wikipedia API から 50 件の最近の編集情報が取得され、Modal Volume に JSON ファイルとし

て保存された。`modal deploy` でデプロイすれば、以後 6 時間ごとに自動的にデータ収集が実行される。

6　発展的なトピック

6.1　ライフサイクル関数

Modal クラス（`@app.cls()`）では、コンテナのライフサイクルに応じた処理を定義できる。

`@modal.enter()` コンテナ起動時に 1 回だけ実行される。モデルのロード、データベース接続、大規模データの読み込みなど、高コストな初期化処理に使う。

`@modal.method()` リクエストごとに実行されるメソッド。

`@modal.exit()` コンテナ終了時に実行される。リソースの解放やログの最終出力に使う。

```
@app.cls(gpu="A10G")
class MyModel:
    @modal.enter()
    def setup(self):
        import torch
        self.model = torch.load("/data/model.pt")

    @modal.method()
    def predict(self, x):
        return self.model(x)

    @modal.exit()
    def cleanup(self):
        print("コンテナ終了")
```

6.2　コールドスタートの管理

Modal のコンテナは使用されていない間にシャットダウンされる。新しいリクエストが来ると新たにコンテナが起動するが、この「コールドスタート」には時間がかかる場合がある。

`container_idle_timeout` パラメータでコンテナのアイドル保持時間を設定できる：

```
@app.function(container_idle_timeout=120)  # 120秒間保持
def process(x):
    return x * 2
```

6.3　タイムアウトとリトライ

長時間のシミュレーションにはタイムアウトの設定が重要である。また、一時的なエラーに対するリトライも設定できる。

```
@app.function(
```

```
    timeout=3600,    # 最大1時間
    retries=3,       # 最大3回リトライ
)
def long_simulation(params):
    pass
```

6.4　Sandbox — 動的コード実行

`modal.Sandbox` は対話的な長寿命コンテナを提供する。ユーザー投稿コードの実行やインタラクティブな実験に有用である。

```
@app.function()
def run_user_code(code: str):
    sandbox = modal.Sandbox.create(
        "python", "-c", code,
        image=modal.Image.debian_slim().pip_install("numpy"),
        app=app,
        timeout=60,
    )
    sandbox.wait()
    return sandbox.stdout.read()
```

6.5　デプロイメント

Modal アプリケーションのデプロイ方法を表 2 にまとめる。

表 2　Modal の実行・デプロイ方法

コマンド	用途	特徴
`modal run`	一時実行	実行後にコンテナ破棄
`modal serve`	開発サーバー	ホットリロード付き
`modal deploy`	本番デプロイ	永続実行（Web/Cron）

`uv run` と組み合わせる場合：

```
# 一時実行
uv run examples/01_hello_modal.py

# 開発サーバー
uv run -m modal serve examples/10_web_dashboard.py

# 本番デプロイ
uv run -m modal deploy examples/11_scheduled_collection.py
```

7　性能評価

本節では、Modal によるクラウド並列実行の性能を定量的に評価する。3 種類のベンチマークを通じて、ローカル逐次実行、ローカル並列実行（`multiprocessing`）、Modal クラウド並列実行の実行時間を比較し、Modal が有効に機能する条件を明らかにする。

すべてのベンチマークスクリプトは `benchmarks/` ディレクトリに格納されている。実行方法は以下の通りである：

```
# 各ベンチマークの実行
uv run --with "modal>=0.73" --with "numpy>=1.26" \
    -m modal run benchmarks/bench_monte_carlo.py

uv run --with "modal>=0.73" --with "numpy>=1.26" \
    -m modal run benchmarks/bench_parameter_sweep.py

uv run --with "modal>=0.73" --with "numpy>=1.26" \
    -m modal run benchmarks/bench_network.py
```

7.1　ベンチマーク 1: モンテカルロ法

モンテカルロ法による π 推定（例題 2 と同一の計算）を用い、バッチ数を 10・50・200 と変化させて実行時間を比較した。各バッチは 10^6 サンプルを処理する。

表 3　モンテカルロ法ベンチマーク結果

規模	総サンプル数	逐次 [秒]	並列 [秒]	Modal [秒]	Modal 倍率
小規模（10 バッチ）	10^7	0.33	1.14	4.06	0.08×
中規模（50 バッチ）	5×10^7	1.61	1.15	1.67	0.96×
大規模（200 バッチ）	2×10^8	6.15	2.68	2.68	2.29×

表 3 に結果を示す。小規模（10 バッチ）ではコンテナ起動とネットワーク通信のオーバーヘッド（約 2〜4 秒）が支配的であり、Modal は逐次実行の約 12 倍遅い。しかし、バッチ数の増加に伴いオーバーヘッドが相対的に小さくなり、200 バッチでは逐次実行の 2.29 倍、ローカル並列（8 プロセス）と同等の性能を達成した。

7.2　ベンチマーク 2: パラメータスイープ（SIR モデル）

確率的 SIR モデルのパラメータスイープを、タスク数 36・120・500 で比較した。大規模ケースでは人口 $N = 50{,}000$、ステップ数 300 の計算量の大きいシミュレーションを用いた。

表 4 に示す通り、**500 タスク・$N = 50{,}000$ の大規模ケースでは Modal が逐次実行の約 7 倍、ローカル並列の約 3.2 倍高速**であった。これは、計算社会科学で典型的な「多数の独立したシミュレーションをパラメータを変えて実行する」パターンにおいて、Modal が最も効果を発揮することを示している。

表 4　SIR モデル・パラメータスイープ ベンチマーク結果

規模	タスク数	N	逐次 [秒]	並列 [秒]	Modal [秒]	Modal 倍率
小規模	36	10,000	0.52	1.27	4.61	0.11×
中規模	120	10,000	1.84	1.41	2.77	0.66×
大規模	500	50,000	30.03	13.99	4.31	**6.97**×

Modal は各タスクを独立したコンテナで並列実行するため、ローカルマシンの CPU コア数（この場合 8 コア）の制約を超えてスケールアウトできる。500 タスクの場合、Modal は数十〜数百のコンテナを同時に起動し、大部分のタスクを並列処理した。

7.3　ベンチマーク 3: ネットワーク分析

Barabási-Albert ネットワークに対する 4 つの構造的指標（次数分布・クラスタ係数・平均経路長・媒介中心性）を `.spawn()` で同時計算した結果を比較した。

表 5　ネットワーク分析ベンチマーク結果

規模	ノード数	逐次 [秒]	並列 [秒]	Modal [秒]	Modal 倍率
小規模	2,000	0.75	1.01	6.65	0.11×
中規模	5,000	2.37	2.73	5.18	0.46×
大規模	10,000	6.37	4.87	8.06	0.79×

表 5 に示す通り、ネットワーク分析では Modal がローカル実行より遅い結果となった。これは `.spawn()` のタスク数がわずか 4 つであり、コンテナ起動オーバーヘッドに対して計算量が不十分であるためである。同一の分析をネットワーク数百個に対して `.map()` で実行する場合は、パラメータスイープと同様に Modal が有利になると予想される。

7.4　考察: Modal を使うべき場面

以上のベンチマーク結果から、Modal の性能特性は以下のように整理できる。

1. **固定オーバーヘッド**：Modal はコンテナの起動、コードの転送、ネットワーク往復に約 2〜4 秒のオーバーヘッドを要する。これは計算量の大小によらずほぼ一定である。
2. **損益分岐点**：逐次実行で合計約 10 秒以上を要する場合に Modal が有利になり始める。個々のタスクが軽量でも、タスク数が多ければ合計計算時間が大きくなり、Modal の並列化効果が活きる。
3. **スケーリング特性**：Modal はローカルマシンの CPU コア数に制約されない。タスク数が数十〜数百を超える場合、ローカル並列（通常 4〜16 コア）に対して大きな優位性を持つ。本ベンチマークでは 500 タスクで約 7 倍の高速化を達成した。
4. **推奨される用途**：`.map()` による大規模パラメータスイープ（数百〜数千タスク）が Modal の最も効果的な利用パターンである。少数の重い計算を `.spawn()` で並列化する場合は、個々の

タスクが数十秒以上を要する場合にのみ有効である。

表 6 に、タスク特性に応じた実行方法の選択指針を示す。

表 6　タスク特性に応じた実行方法の選択指針

条件	推奨方法	理由
タスク数少・計算軽量	ローカル逐次	オーバーヘッドなし
タスク数少・計算重い	ローカル並列	コア数分の高速化
タスク数多・計算軽量	ローカル並列	オーバーヘッド回避
タスク数多・計算重い	**Modal**	**大幅なスケールアウト**
GPU が必要	**Modal**	**オンデマンド GPU**

ベンチマーク結果の再現

本ベンチマークの結果は実行環境（ローカルマシンの CPU 性能、ネットワーク帯域、Modal のコンテナ割り当て状況）に依存する。読者自身の環境で `benchmarks/` ディレクトリ内のスクリプトを実行し、実際の性能を確認されたい。

8　まとめ

本チュートリアルでは、サーバーレスクラウドプラットフォーム Modal を用いた計算社会科学の計算実験手法を包括的に解説した。主要な知見を以下にまとめる。

1. **インフラ管理の排除**：Modal は Python デコレータだけでクラウド実行を実現し、Docker・Kubernetes・ジョブスケジューラの知識を不要にする。これにより研究者はモデルの設計と分析に集中できる。
2. **容易な並列化**：`.map()` メソッドにより、パラメータスイープや多数回のモンテカルロシミュレーションを 1 行で並列化できる。逐次実行では数時間〜数日を要する計算が数分で完了する可能性がある。
3. **定量的な性能評価**：ベンチマーク（第 7 節）により、500 タスクの大規模パラメータスイープでは逐次実行の約 7 倍の高速化を確認した。Modal の有効性はタスク数と個別計算量に依存し、タスク数が多く計算が重いほど効果が大きい。
4. **オンデマンド GPU**：デコレータに `gpu="T4"` と記述するだけで GPU を利用でき、自然言語処理モデルや大規模数値計算を高速化できる。
5. **結果の永続化と共有**：Volume によるストレージと Web エンドポイント機能により、シミュレーション結果の保存・分析・公開を一貫したプラットフォーム上で実現できる。
6. **再現性の確保**：PEP 723 インラインメタデータと Modal のイメージ定義により、計算環境を完全に宣言的に記述でき、実験の再現性が向上する。

本チュートリアルで示した例題は、囚人のジレンマ（ゲーム理論）、意見ダイナミクス（社会物理学）、分居モデル（空間的社会現象）、投票者モデル（社会的影響）、ネットワーク分析（複雑ネット

ワーク科学)、感情分析（計算言語学)、SIR モデル（数理疫学）と、計算社会科学の広範な領域をカバーしている。これらの例題を出発点として、読者の研究分野に応じたモデルの開発と大規模実験が可能となることを期待する。

参考文献

[1] Lazer, D., Pentland, A., Adamic, L., et al. (2009). Computational Social Science. *Science*, 323(5915), 721–723.

[2] Axelrod, R. (1984). *The Evolution of Cooperation*. Basic Books.

[3] Deffuant, G., Neau, D., Amblard, F., & Weisbuch, G. (2000). Mixing beliefs among interacting agents. *Advances in Complex Systems*, 3(01n04), 87–98.

[4] Schelling, T. C. (1971). Dynamic models of segregation. *Journal of Mathematical Sociology*, 1(2), 143–186.

[5] Barabási, A.-L. & Albert, R. (1999). Emergence of Scaling in Random Networks. *Science*, 286(5439), 509–512.

[6] Kermack, W. O. & McKendrick, A. G. (1927). A contribution to the mathematical theory of epidemics. *Proceedings of the Royal Society A*, 115(772), 700–721.

[7] Castellano, C., Fortunato, S., & Loreto, V. (2009). Statistical physics of social dynamics. *Reviews of Modern Physics*, 81(2), 591–646.

付録 A　Modal の代替サービス

Modal は計算社会科学の大規模計算に有用なプラットフォームであるが、研究の要件やコスト制約に応じて他のサービスが適する場合もある。本付録では、Modal と同様のサーバーレスクラウドコンピュート機能を提供する主要な代替サービスを概観する。

A.1　サーバーレス GPU / クラウドコンピュート

表 7 に、Modal の主な代替サービスとその特徴を示す。

A.2　大手クラウドのマネージドサービス

大手クラウドプロバイダも機械学習・計算ワークロード向けのマネージドサービスを提供している(表 8)。これらはエンタープライズ規模の運用や既存クラウドインフラとの統合が必要な場合に適している。

A.3　用途別の選択指針

表 9 に、研究の要件に応じたプラットフォーム選択の指針を示す。

表 7　Modal の代替サービス一覧

サービス	特徴
RunPod[*4]	秒単位課金の GPU コンテナ。サーバーレスとポッド（専有インスタンス）の両形態に対応。Modal より低コストな傾向があり、コスト重視の場合に有力。
Anyscale[*5]	分散計算フレームワーク Ray の開発元によるマネージドプラットフォーム。大規模バッチ推論や分散訓練に強く、Ray のエコシステム（Ray Tune, Ray Data 等）との統合が魅力。
Replicate[*6]	事前訓練済みモデルのサーバーレス推論に特化。豊富なモデルライブラリを API 経由で即座に利用可能。自前のモデルを Cog コンテナとしてデプロイすることもできる。
Beam Cloud[*7]	Python SDK によるコンテナ高速起動とホットリロード対応。Modal に近い開発者体験を提供し、デコレータベースの API でクラウド実行を実現する。
Cerebrium[*8]	12 種以上の GPU タイプをサポートし、高速オートスケーリングと低レイテンシを実現。リアルタイム推論とバッチ処理の両方に対応。
Coiled[*9]	Dask ベースの分散計算プラットフォーム。pandas や scikit-learn のワークロードをスケールアウトでき、データサイエンスのパイプラインに最適。

表 8　大手クラウドの ML プラットフォーム

サービス	特徴
AWS SageMaker	Amazon Web Services の統合 ML プラットフォーム。訓練・推論・モニタリングの包括的機能を提供。AWS エコシステム（S3, Lambda 等）との統合が強み。
Google Vertex AI	Google Cloud Platform 上の ML プラットフォーム。AutoML、カスタム訓練、モデルレジストリを統合。BigQuery や Cloud Storage とのシームレスな連携。
Azure Machine Learning	Microsoft Azure の ML サービス。Azure DevOps との CI/CD 統合や、VS Code からのリモート開発環境に対応。

表 9　用途別プラットフォーム選択の指針

主な要件	推奨サービス	理由
開発の容易さ	Modal, Beam Cloud	デコレータベースの API
低コスト GPU	RunPod	秒単位課金、低価格帯
分散並列計算	Anyscale (Ray)	Ray エコシステム
データ分析の並列化	Coiled (Dask)	pandas 互換の分散処理
推論 API のみ	Replicate	事前デプロイ済みモデル
GPU 種類の豊富さ	Cerebrium	12 種以上の GPU 対応
既存クラウド統合	SageMaker 等	エコシステム連携

計算社会科学における選択

計算社会科学のパラメータスイープのような用途（CPU 中心・大量タスクの独立並列実行）では、Modal のデコレータベース API（`.map()` による 1 行並列化）に匹敵する開発体験を提供するサービスは少ない。Anyscale (Ray) や Coiled (Dask) がプログラミングモデルとして最も近い選択肢であるが、いずれも Modal とは異なる抽象化を採用している。研究グループの技術スタックや予算に応じて検討されたい。