GIS DAY IN 東京 2024
Eコース資料

SCIKIT-MOBILITYを用いた行動パターンの分析入門

東京都立大学 都市環境学部 地理学教室
中山大地

目次

1 scikit-mobility を用いた行動パターンの分析入門

- 東京都立大学 都市環境学部 地理環境学科 中山大地

- GIS Day in 東京 2024 E コース
- 2024 年 12 月 14 日 東京都立大学 南大沢キャンパス
- このテキストの URL https://github.com/bokutachi256/gisday2024
- Google Colaboratory の URL https://colab.research.google.com/

1.1 使い方

1.1.1 動作環境について

1. Google ドライブに `gisday2024` フォルダを作成する（フォルダ名は「すべて小文字」にすること）.
2. 共有フォルダ（https://drive.google.com/drive/folders/1OKj-JS8O2NR2qPlvLV4oiTNdrMEV8Jpn

?usp=sharing）から以下のデータをダウンロードする．

- `gps_logdata.zip`: GPX 型式の GPS ログデータを圧縮したもの
- `r2ka.fgb`: FlatGeoBuf 型式の熊本県と宮崎県の小地域ポリゴン（e-Stat より入手）
- `gisday2024_agent_vars.fgb`: FlatGeoBuf 型式のマルチエージェントシミュレーションの計算結果（熊本県阿蘇市内牧地区）

3. Google Drive に作成した `gisday2024` フォルダに上記 3 点のファイルをアップロードする．

■1.1.1.1 **使用しているライブラリ** 使用している主なライブラリの一覧．

- gpxpy: GPS のログデータ（GPX 型式）を扱うためのライブラリ
- scikit-mobility: モビリティデータを扱うライブラリ
- movingpandas: 移動データを扱うライブラリ
- folium: 地図（WEB マップ）を扱うためのライブラリ
- datetime: 時刻を扱うためのライブラリ
- zoneinfo: タイムゾーンを扱うためのライブラリ
- pandas: データフレームを扱うためのライブラリ
- geopandas: 地理情報を扱うためのライブラリ
- pyproj: PROJ（地図投影と座標系を扱うためのライブラリ）への Python インターフェイス
- glob: ファイルパス名解決のためのライブラリ
- shapely: ジオメトリを扱うためのライブラリ
- numpy: 行列を扱うためのライブラリ

1.2 実行の準備

1.2.1 ライブラリのインストールとインポート

■1.2.1.1 **Google Colab へのライブラリインストール** Google Colaboratory は基本的なライブラリはインストール済みですが，標準ではインストールされていないライブラリをインストールする必要があります．インストールするとランタイムの再起動を求められる事があるので，指示に従ってランタイムを再起動してください．再起動後は「ライブラリのインポート」以降を実行してください．

```
import sys

# Google Colaboratory を使用している場合は以下の必須ライブラリをインストールする
# gpxpy: GPX フォーマットを扱うライブラリ
# scikit-mobility: モビリティデータを扱うライブラリ
# movingpandas: 移動データを扱うライブラリ
# folium: 地図を表示するライブラリ
```

```
if 'google.colab' in sys.modules:
  %pip install gpxpy
  %pip install scikit-mobility
  %pip install movingpandas
  %pip install folium
  %pip install mapclassify
  %pip install -U geopandas
  %pip install stonesoup
```

```
# 必要なライブラリをインポート
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo

import sys
import gpxpy
import gpxpy.gpx
import folium
import pandas as pd
import geopandas as gpd
import movingpandas as mpd
import skmob
from skmob import TrajDataFrame
from skmob.preprocessing import detection, clustering, compression
from skmob.preprocessing import filtering
from skmob.tessellation import tilers

from shapely.geometry import Point, LineString
from pyproj import CRS
import glob
import zipfile

import numpy as np
```

■1.2.1.2　ライブラリのインポート

1.2.2 データフォルダのマウント

Google Colaboratory から Google Drive にアップロードしたデータにアクセスするため，データフォルダのマウントを行います．途中で Google Colaboratory から Google Drive へのアクセス権付与を求められますので，すべての権限を与えてください．

```
if 'google.colab' in sys.modules:
  # google.coalb: GoogleDrive 内のファイルにアクセスするために必要
  from google.colab import drive
  drive.mount('/content/drive')
  # GoogleDrive 内の "マイドライブ/gisday2024/" フォルダにアクセスできるような設定を行う．
  # フォルダ名の最後に `/` を必ず追加すること．
  base_dir = "/content/drive/MyDrive/gisday2024/"
else:
  base_dir = "./data/"

%matplotlib inline
```

2 GPX データの分析

熊本県阿蘇地方近辺の 31 日分の GPS ログデータ（学外実習時に取得したものです）を用いてデータの分析を行います．ここでは GPS ログデータから立ち寄り地（採水ポイントや現地調査のポイント）を抽出し，立ち寄り地間のフローを作成します．

手順は以下になります．

1. GPX データの読み込みと TrajDataFrame への変換
2. フィルタを用いた TrajDataFrame のクリーニングおよび縮約
3. 立ち寄り地点の抽出
4. 立ち寄り地点間の OD 作成
5. OD からフローの作成

2.1 GPS から TrajDataFrame を作る

2.1.1 GPX データの読み込み

zip 型式に圧縮されている GPX ファイルを読み込み，DataFrame 型式に変換した後に TrajDataFrame を作成します．TrajDataFrame は scikit-mobikity の基本的なデータ形式で，TrajDataFrame に対して様々なフィルタをかけたりデータの縮約をします．また，立ち寄り地の検出も TrajDataFrame に対して行います．

GPX データの読み込みは gpxpy ライブラリを使用します．GPX のトラックを構成するセグメントからポイントを抽出し，緯度経度の座標や時刻を取得します．取得した座標などの情報から pandas の DataFrame を作成し，それを TrajDataFrame に変換します．

```
# タイムゾーンの設定
dt_tz = ZoneInfo('Asia/Tokyo')
# 日時文字列形式
dt_fmt = '%Y-%m-%d %H:%M:%S'
# 配列の初期化
datetime = []
lat = []
lon = []
alt = []
id = []
uid = 0
df = pd.DataFrame()

# ZIP 圧縮された GPX ファイルの読み込み
inzipfile =  'gps_logdata.zip'
with zipfile.ZipFile(base_dir + inzipfile) as zf:
    infiles = zf.namelist()
    for infile in infiles:
        gpx_file = zf.open(infile, 'r')
        # GPX ファイルのパース
        gpx = gpxpy.parse(gpx_file)
        # GPX データの読み込み
        for track in gpx.tracks:
            for segment in track.segments:
                # ポイントデータリストの読み込み
                points = segment.points
                # ポイントデータの読み込み
                for i in range(len(points)):
                    # データ抽出
                    datetime.append(points[i].time.astimezone(dt_tz).strftime(dt_fmt))
                    lat.append(points[i].latitude)
                    lon.append(points[i].longitude)
                    alt.append(points[i].elevation)
                    id.append(uid)
        uid += 1

df['lat'] = lat
df['lon'] = lon
df['alt'] = alt
```

```
df['datetime'] = datetime
df['id'] = id
```

2.1.2 TrajDataFrame の作成

GPX データから作成した DataFrame を TrajDataFrame に変換します．TrajDataFrame には個人を識別する user_id がありますが，今回は 31 日分の GPS データのそれぞれに異なる user_id を付与しています．

```
# gpx を TrajDataFrame に変換
tdf = skmob.TrajDataFrame(df, latitude='lat', longitude = 'lon', datetime='datetime',
                          user_id='id', crs={'init': 'epsg:4326'})
```

GPX データから TrajDataFrame ができましたので，データ数の確認をしてみます．

```
print(f'オリジナル: {len(tdf)}')
```

```
オリジナル: 128802
```

全部で 128,802 個のポイントがありました．

2.2 TrajDataFrame のクリーニングと各種前処理

GPS の位置情報にはノイズが含まれていることがあるため，それらを取り除く必要があります．そのためにデータのクリーニングを行います．まずは確認のため TrajDataFrame をプロットしてみます．

TrajDataFrame のプロットは plot_trajectory メソッドを使います．

```
# TrajDataFrame のプロット
tdf.plot_trajectory(zoom=12, weight=3, opacity=0.9, max_users=100, max_points=None,
                    start_end_markers=True)
```

```
<folium.folium.Map at 0x34a15dbe0>
```

各 Trajectory の始点と終点にマーカーが表示されています．クリックすると日付時刻と座標がポップアップします．

2.2.1 速度の計算

GPS データのノイズとして異常に速い速度が記録される事があります．確認のため経路ごとの速度を求めて地図表示してみましょう．

scikit-mobility には経路の速度を計算する方法がないため，TrajDataFrame を MovingPands の TrajectoryCollection に変換して速度を求めます．まずは TrajDatFrame を GeoPandas の GeoDataFrame に変換し，そこから TrajctoryCollection に変換します．その後，TrajectoryCollection に対して速度を求めます．

```
# TrajDataFrame を MovingPandas の TrajectoryCollection に変換する

# TrajDataFrame を GeoDataFrame に変換する
gdf = tdf.to_geodataframe()
# GeoDataFrame を TrajectoryCollection に変換する
collection = mpd.TrajectoryCollection(gdf, 'uid', t='datetime')

# TrajectoryCollection に対して速度を計算する
collection.add_speed(overwrite=True, units=("km", "h"))
```

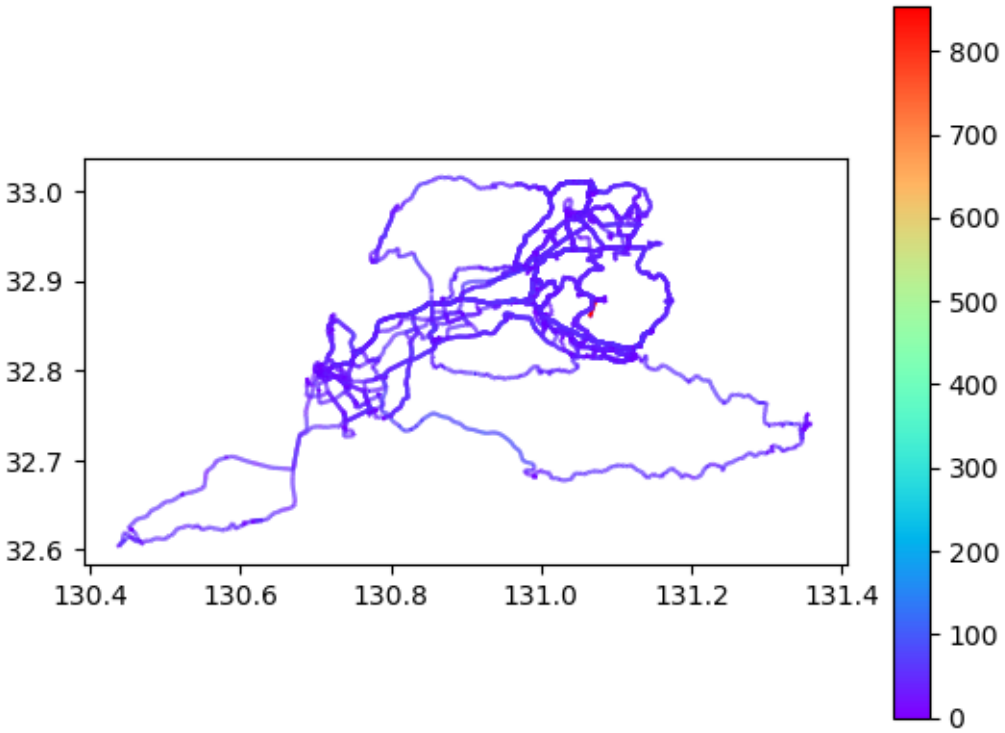
TrajectoryCollection に対して add_speed メソッドを使って速度を求めます．求まった速度は speed に格納されます．

速度が求まったら表示してみましょう．速度を求めた TrajectoryCollection は GeoPandas の GeoDataFrame オブジェクトなので，plot メソッドや explore メソッドが使えます．

plot メソッドを使って speed で色分けして表示します．

```
# すべての Trajectory の speed をプロットする
# やけにスピードが速いものがある
collection.plot(column="speed", cmap='rainbow', legend=True)
```

```
<Axes: >
```



どうやら時速 800km を越えるセグメントがあるようです．これは明らかにノイズなのでフィルタリングする必要があります．

2.2.2 データのクリーニング

主なフィルタリングには以下の 2 種類があります．

- 速度によるフィルタリング
- 短小ループの削除

またデータを縮約するための経路圧縮も必要になります．

まずは速度フィルタリングを行います．ここでは時速 100km 以上のセグメントを削除します．速度フィルタリングは `filtering.filter` を使います．

対象となる TrajDataFrame は `tdf` ですので，`filtering.filter` の第一引数に `tdf` を指定します．`max_speed_km` で指定する値以上のセグメントを削除します．フィルタリング後のデータは `stdf` とします．

```
#速度によるフィルタリング
# TrajDataFrame のうち，速度が 100km/h を超えるデータを除外
stdf = filtering.filter(tdf, max_speed_kmh=100)
```

速度フィルタリングが終わったらデータ数の確認をしてみましょう．

```
print(f'オリジナル: {len(tdf)}')
print(f'速度フィルタリング後: {len(stdf)}')
```

オリジナル: 128802
速度フィルタリング後: 128727

オリジナルデータに比べてフィルタリング後のデータは小さくなっていることがわかります．

次に速度フィルタを通した TrajDataFrame を表示して確認してみます．`plot_trajectory` メソッドを使います．

```
stdf.plot_trajectory(zoom=12, weight=3, opacity=0.9, max_users=100, max_points=None)
```

```
<folium.folium.Map at 0x35fb91a90>
```

■2.2.2.1 速度をフィルタリングした TrajectoryDataFrame に対して速度を計算する プロットではフィルタリングの効果がわからないので，速度フィルタリング済みのデータに対して再度速度を計算し，それをプロットして確認します．

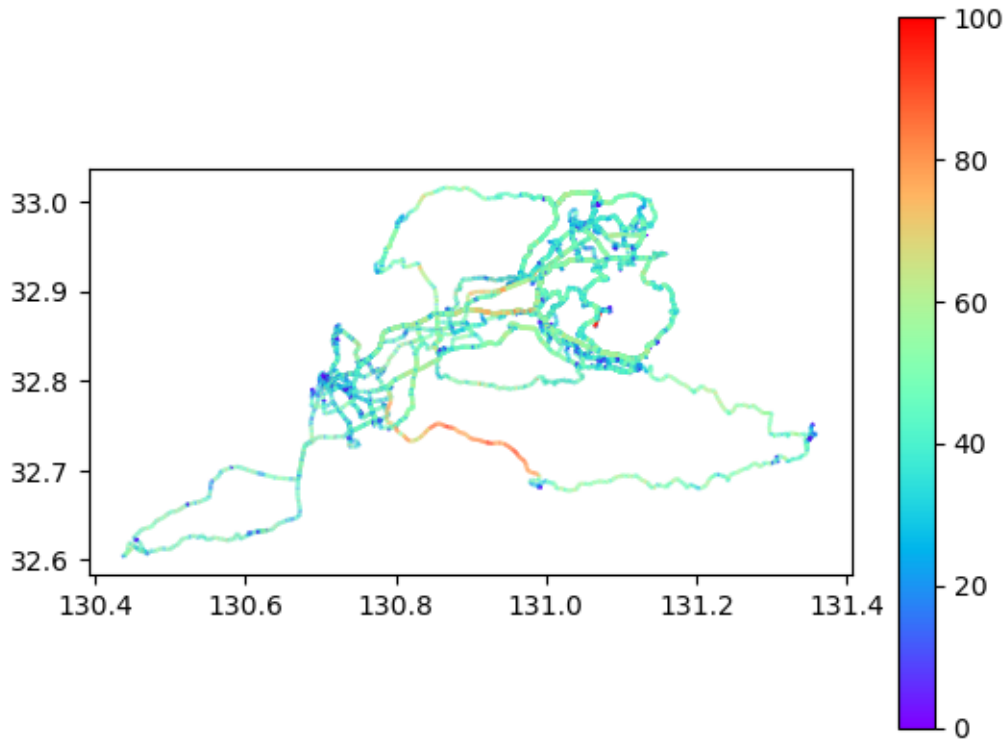
速度計算のためには一旦 MovingPandas の TrajectoryCollection に変換する必要があります．

```
# TrajDataFrame を GeoDataFrame 経由で MovingPandas の TrajectoryCollection に変換する
gdf = stdf.to_geodataframe()
collection = mpd.TrajectoryCollection(gdf, 'uid', t='datetime')
# TrajectoryCollection の速度を計算する
collection.add_speed(overwrite=True, units=("km", "h"))
# TrajectoryCollection を日付でソートする
collection.trajectories.sort(key=lambda x: x.get_start_time())
```

速度のプロットを表示してみましょう.

```
# すべての Trajectory の speed をプロットする
collection.plot(column="speed", cmap='rainbow', legend=True)
```

```
<Axes: >
```



最高速度が時速 100km になっているので，速度フィルタリングはうまくいっているようです.
次に explore メソッドを使ってインタラクティブマップに速度を日付ごとに表示してみます.

```
mapcenter = [32.92, 131.1]
```

```
m = folium.Map(
    location=mapcenter,
    zoom_start=11
)

# 地理院タイル（淡色地図）を追加
# '全国最新写真（シームレス）'も面白いかも
# https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg
tile_layer = folium.TileLayer(
    tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
    attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
        地理院タイル</a>',
    name='地理院地図'
)
tile_layer.add_to(m)

# カスタムCSSを追加してタイルレイヤーをモノクロ化
m.get_root().html.add_child(folium.Element("""
<style>
    .leaflet-tile {
        filter: grayscale(100%);
    }
</style>
"""))

# レイヤーコントロールに日付を表示するための日付フォーマット
dt_fmt = '%Y-%m-%d'
# TrajectoryCollectionの各Trajectoryをプロット
for trajectory in collection.trajectories:
    # Trajectoryの開始時刻を取得
    start_time = trajectory.get_start_time().strftime(dt_fmt)
    # Trajectoryを表示
    trajectory.explore(
        m=m,
        column="speed",
        cmap='rainbow',
        vmin=0, vmax=100,
        legend=False,
        style_kwds={'weight': 5, 'opacity': 0.7},
```

```
        name=f'{start_time}'
    )

# レイヤーコントロールを追加
folium.LayerControl().add_to(m)
display(m)
```

■2.2.2.2　**インタラクティブマップに経路の速度を表示する**

```
<folium.folium.Map at 0x36f4c91d0>
```

explore メソッドを使っているので，右上のレイヤーコントロールメニューで表示のオンオフを変えることができます．また，マウスオーバーすることによりセグメントの情報をポップアップすることもできます．

南側の速度が速いセグメントは，部分開通している九州中央自動車道を通行しているときのものです．

■2.2.2.3　**ループの削除**　GPS データのノイズには短小ループが作成される事があります．短小ループの削除は速度によるフィルタリングと同時に行います．これも実行してみます．

```
lstdf = filtering.filter(tdf, max_speed_kmh=100, include_loops=True)
```

これでフィルタリングによるデータクリーニングは完了です．クリーニング前後のデータ数を見て，効果を見てみましょう．

```
print(f'オリジナル: {len(tdf)}')
print(f'速度フィルタリング後: {len(stdf)}')
print(f'ループを含む速度フィルタリング後: {len(lstdf)}')
```

オリジナル：128802
速度フィルタリング後：128727
ループを含む速度フィルタリング後：128665

データ数が少なくなっているのでクリーニングはできていますが，思ったよりもデータは減っていないようです．マップ表示での確認もしてみます．

```
lstdf.plot_trajectory(zoom=12, weight=3, opacity=0.9, max_users=100, max_points=None)
```

```
<folium.folium.Map at 0x374225940>
```

■2.2.2.4　**データの圧縮**　このままではデータ量が多くて扱いが不便なので，compression.comress を使って経路データの圧縮を行います．spatial_radius_km パラメーターで削減する範囲をしています．ここでは 50m としました．

```
clstdf = compression.compress(lstdf, spatial_radius_km=0.05)
```

経路圧縮の効果を見てみましょう.

```
print(f' オリジナル: {len(tdf)}')
print(f' 速度フィルタリング後: {len(stdf)}')
print(f' ループを含む速度フィルタリング後: {len(lstdf)}')
print(f' 圧縮後: {len(clstdf)}')
```

50m 範囲での経路圧縮をすることにより，元データの 3 分の 1 ほどのサイズになりました.

オリジナルデータと圧縮データを地図表示して効果を見てみます．すべての Trajetory を表示するのは難しいので，最初の Trajectory（uid が 0 の Trajectory ）のみを表示します.

オリジナルデータは赤で，フィルタリングして圧縮したデータは青で表示されます.

```
# オリジナルデータを赤で描画
map_f = clstdf[clstdf['uid']==10].plot_trajectory(zoom=12, weight=3, opacity=0.9,
  max_users=100, max_points=None, hex_color='#FF0000')
# 圧縮後のデータを青で描画
tdf[tdf['uid']==10].plot_trajectory(zoom=12, weight=3, opacity=0.5,
  max_users=100, max_points=None, hex_color='#0000FF', map_f=map_f)
```

```
<folium.folium.Map at 0x3a7088510>
```

■2.2.2.5 **データの処理過程の確認** オリジナルの Trajectory に対して様々なクリーニングと圧縮をしましたが，ある Trajectory がどのような処理を施されたのか確認したいことがあります.

TrajDataFrame の `parameters` メソッドで適用された処理を確認することができます.

```
clstdf.parameters
```

```
{'filter': {'function': 'filter',
  'max_speed_kmh': 100,
  'include_loops': True,
  'speed_kmh': 5.0,
  'max_loop': 6,
  'ratio_max': 0.25},
 'compress': {'function': 'compress', 'spatial_radius_km': 0.05}}
```

`clstdf` は時速 100km の速度フィルタリング，ループの削除，0.05km 範囲の圧縮がなされたデータであることがわかります.

■2.2.2.6　**クリーニングしたデータに対して速度を求める**　クリーニングしたデータの速度を求め，インタラクティブマップに表示してみましょう．このデータが最終的に採用されるデータになります．

まずは速度を求めます．

```
# clstdf を TrajectoryCollection に変換する
collection = mpd.TrajectoryCollection(clstdf.to_geodataframe(), 'uid', t='datetime')
# TrajectoryCollection の速度を計算する
collection.add_speed(overwrite=True, units=("km", "h"))
# TrajectoryCollection を日付でソートする
collection.trajectories.sort(key=lambda x: x.get_start_time())
```

次にインタラクティブマップに表示します．背景地図として空中写真も選べるようにしました．

```
mapcenter = [32.92, 131.1]

m = folium.Map(
    location=mapcenter,
    # tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
    # attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
    #     # 地理院タイル</a>',
    # name='淡色地図',
    zoom_start=11
)

# 地理院タイル（淡色地図）を追加
# '全国最新写真（シームレス）'も面白いかも
# https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg
tile_layer = folium.TileLayer(
    tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
    attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
        地理院タイル</a>',
    name='地理院地図'
)
tile_layer.add_to(m)

# 地理院タイル（淡色地図）を追加
# '全国最新写真（シームレス）'も面白いかも
# https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg
```

```
tile_layer = folium.TileLayer(
    tiles='https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg',
    attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
        地理院タイル</a>',
    name=' 全国最新写真（シームレス）'
)
tile_layer.add_to(m)

# カスタム CSS を追加してタイルレイヤーをモノクロ化
m.get_root().html.add_child(folium.Element("""
<style>
    .leaflet-tile {
        filter: grayscale(100%);
    }
</style>
"""))

# TrajectoryCollection の各 Trajectory をプロット
dt_fmt = '%Y-%m-%d'
for trajectory in collection.trajectories:
    start_time = trajectory.get_start_time().strftime(dt_fmt)
    trajectory.explore(
        m=m,
        column="speed",
        cmap='rainbow',
        vmin=0, vmax=100,
        legend=False,
        style_kwds={'weight': 5, 'opacity': 0.7},
        name=f'{start_time}'
    )


folium.LayerControl().add_to(m)
display(m)
```

```
<folium.folium.Map at 0x3a5f59b50>
```

2.3 GPS データからフローを作成する

クリーニングした Trajectory を使って，行動の発着地を結んだフローを作成します．

scikit-mobility には TrajDataFrame からフローを示す FlowDataFrame を作詞する機能があります．主に立ち寄り地と Tessellation の空間結合を行う `tdf.mapping` メソッドと，`tdf.mapping` を行った後に TrajDataFrame を FlowDataFrame に変換する `tdf.to_flowdataframe` メソッドを使うのですが，scikit-mobility に対応している GeoPandas のバージョンが古いため今回のプログラムでは動きません．このため，自前でフローを作成します．

フローの作成手順は以下になります．

1. TrajDataFrame から立ち寄り地点を抽出する
2. 近接している立ち寄り地点をクラスタリングする
3. クラスタリングした立ち寄り地点から発着地の集計単位になるテッセレーションを作成する
4. テッセレーションの代表点を求める
5. 立ち寄り地点間の OD 作成
6. OD からフローの作成

2.3.1 立ち寄り地の検出

クリーニングした TrajDataFrame（`clstdf`）に対して detection.stay_location‘を用いて立ち寄り地点を検出します．求めた立ち寄り地点は TrajDataFrame になります．

ここでは半径 200m（0.2km）以内に 5 分間以上止まった場合に立ち寄りとします．求めた立ち寄り地点の TrajDataFrame は `stops` とします．

```
stops = detection.stay_locations(clstdf, minutes_for_a_stop=5, spatial_radius_km=0.2)
```

確認のため立ち寄り地点をプロットします．

```
# 立ち寄り地をプロット
stops.plot_trajectory(zoom=12, weight=3, opacity=0.9, max_users=100, max_points=None)
```

```
<folium.folium.Map at 0x3b52ea030>
```

■2.3.1.1 **立ち寄り地のクラスタリング** `clustering.cluster` を使って同一地点と思われる立ち寄り地点をまとめます．属性 `cluster` の値は，0 ほどクラスターメンバーが多い事を示します．今回は半径 1km 以内を同一の立ち寄り地としました．クラスタリング済みの立ち寄り地点は `cstops` とします．

```
cstops = clustering.cluster(stops, cluster_radius_km=1, min_samples=1)
```

クラスタリングした立ち寄り地点を GeoDataFrame に変換し，インタラクティブマップに表示します．

```
mapcenter = [32.92, 131.1]

m = folium.Map(
    location=mapcenter,
    # tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
    # attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
    #     # 地理院タイル</a>',
    # name='淡色地図',
    zoom_start=11
)

# 地理院タイル（淡色地図）を追加
# '全国最新写真（シームレス）'も面白いかも
# https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg
tile_layer = folium.TileLayer(
    tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
    attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
        地理院タイル</a>',
    name='地理院地図'
)
tile_layer.add_to(m)

# カスタムCSSを追加してタイルレイヤーをモノクロ化
m.get_root().html.add_child(folium.Element("""
<style>
    .leaflet-tile {
        filter: grayscale(100%);
    }
</style>
"""))

cstops.to_geodataframe().explore(
    m=m,
    column='cluster',
    style_kwds={'weight': 5})

folium.LayerControl().add_to(m)
display(m)
```

■2.3.1.2　**メッシュ単位のフローデータの作成**　立ち寄り地点を空間集計するためにテッセレーションを作成します. テッセレーションは任意のポリゴンデータを使うことができますが, scikit-mobility の `tilers.tiler` を使って方形メッシュのテッセレーションを作成することもできます.

`tilers.tiler` はパラメーターとして範囲を示す `base_shape` を指定できます. `base_shape` として GeoDataFrame に変換した立ち寄り地点のデータ `cstops` を指定すれば, 立ち寄り地点の範囲のテッセレーションが作成できます.

ここでは `cstops` の範囲の 5km 正方メッシュのテッセレーションを作成します.

```
# クラスタリングした立ち寄り地からテッセレーションを作成する

# クラスタリングした立ち寄り地を GeoDataFrame に変換
cstops_gdf = cstops.to_geodataframe()
# GeoDataFrame を元にテッセレーションを作成
tessellation = tilers.tiler.get("squared", base_shape=cstops_gdf, meters=5000)
```

作成したテッセレーションとクラスタリングした立ち寄り地点を地図表示します.

```
mapcenter = [32.92, 131.1]

m = folium.Map(
    location=mapcenter,
    zoom_start=11
)

# 地理院タイル（淡色地図）を追加
# '全国最新写真（シームレス）'も面白いかも
# https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg
tile_layer = folium.TileLayer(
    tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
    attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
        地理院タイル</a>',
    name=' 地理院地図'
)
tile_layer.add_to(m)

# カスタム CSS を追加してタイルレイヤーをモノクロ化
m.get_root().html.add_child(folium.Element("""
<style>
```

```
    .leaflet-tile {
        filter: grayscale(100%);
    }
</style>
"""))

tessellation.explore(
    m=m,
    name='Tessellation',
    style_kwds={'fillColor': 'none'}
)

cstops_gdf.explore(
    m=m,
    column='cluster',
    name='Clustered Stops',
    style_kwds={'weight': 5}
)

folium.LayerControl().add_to(m)
display(m)
```

```
<folium.folium.Map at 0x3b5423850>
```

■2.3.1.3　**クラスタリングした立ち寄り地に** (Origin) **を設定する**　テッセレーションにはメッシュの ID となる tile_id があります．この tile_ID をクラスタリングした立ち寄り地点に空間結合し，どの立ち寄り地点がどのメッシュに含まれているか識別できるようにします．この tile_ID が，ある立ち寄り地点の「発地 ID (origin)」になります．

クラスタリングした立ち寄り地点とテッセレーションを sjoin メソッドで空間結合します．この際，結合は左結合（cstops に合わせて結合）で，空間関係は「含まれる（within)」になります．これで立ち寄り地点の属性にその立ち寄り地点を含むメッシュの属性が結合されます．結合後の属性名 tile_ID を origin に変更し，不要な列 index_right を削除します．最後に属性 origin の型を実数にキャストし，着地 ID を格納する属性 destination を確保します．

```
# 空間結合で発地を設定する．発地 ID（origin の値）は Tessellation の tile_ID を使用する．
temp = cstops_gdf.to_crs(epsg=4326).sjoin(tessellation, how='left', predicate='within')\
  .rename(columns={'tile_ID': 'origin'})\
  .drop('index_right', axis=1)
```

```
# 発地（origin）の型を float64 に変換
temp.astype({'origin': np.float64})
# 着地（destination）の列を追加
temp['destination'] = np.float64(0)
```

■2.3.1.4　**クラスタリングした立ち寄り地に着地 (Destination) を設定する**　次に着地（destination）を設定します．着地はある立ち寄り地点から次にどの着地点に移動するかを示したもので，発地と同様にメッシュの ID で表します．

ここである日の立ち寄り地点の一部を見てみます．

```
temp[temp['uid']==0].head(10)
```

```
         lat         lng   alt            datetime  uid    leaving_datetime  \
0  32.791131  130.731538  37.0 2016-08-31 08:46:21    0 2016-08-31 08:55:21
1  32.791495  130.734562  47.0 2016-08-31 08:55:21    0 2016-08-31 09:17:39
2  32.790454  130.735594  40.0 2016-08-31 09:17:39    0 2016-08-31 09:23:52
3  32.790172  130.733478  47.0 2016-08-31 09:23:52    0 2016-08-31 09:35:50
4  32.781906  130.741104  41.0 2016-08-31 09:43:11    0 2016-08-31 10:06:32
5  32.764800  130.754235  36.0 2016-08-31 10:14:50    0 2016-08-31 10:35:20
6  32.754117  130.771256  35.0 2016-08-31 10:42:07    0 2016-08-31 11:00:52
7  32.756231  130.779212  35.0 2016-08-31 11:02:14    0 2016-08-31 11:19:02
8  32.774180  130.783923  31.0 2016-08-31 11:29:36    0 2016-08-31 11:43:57
9  32.792864  130.793660  51.0 2016-08-31 12:01:50    0 2016-08-31 12:13:14

   cluster                    geometry origin  destination
0        1  POINT (130.73154 32.79113)     70          0.0
1        1   POINT (130.73456 32.7915)     70          0.0
2        1  POINT (130.73559 32.79045)     70          0.0
3        1  POINT (130.73348 32.79017)     70          0.0
4       10   POINT (130.7411 32.78191)     70          0.0
5        9   POINT (130.75423 32.7648)     70          0.0
6        5  POINT (130.77126 32.75412)     80          0.0
7        5  POINT (130.77921 32.75623)     80          0.0
8        8  POINT (130.78392 32.77418)     81          0.0
9        4  POINT (130.79366 32.79286)     81          0.0
```

最初の発地（origin）はメッシュ番号 70 です．datetime はその立ち寄り地点に着いた時刻，leaving_datetime は立ち寄り地点を離れた時刻です．データは時刻順に並んでいるため，ある立ち寄り地

点の着地は次の立ち寄り地点の発地（origin）になります．

従って，時系列順に発地のリストを作成し，一つずらしたものをその日の立ち寄り地の着地に入れれば良いことになります．最後の立ち寄り地点だけは着地が無いために削除します．

以下では上記を行い．最終的に発着地が加わった立ち寄り地点を od として保存します．

```
# 発着地 ID を含む GeoDataFrame を作成
od = gpd.GeoDataFrame()

# uid ごとに処理するため，uid のユニーク値をリスト化する
uids = temp['uid'].unique().astype(int).tolist()
# uid ごとに処理を行う
for uid in uids:
  # uid ごとのデータを temp から抽出
  temp_origin = temp[temp['uid'] == uid]
  # 着地のリストを作成する．着地は発地を 1 つずらしたもの
  temp_dest = np.roll(temp_origin['origin'], -1).astype(np.float64)
  # 最後のデータは着地がないため，nan にする
  temp_dest[-1] = np.nan
  # 着地の列を追加する
  temp_origin.loc[:, 'destination'] = temp_dest
  # 結果を od に追加する
  od = pd.concat([od, temp_origin])
# 着地のない行（着地が nan になっている行）を削除する
od.dropna(inplace=True)
# origin と destination の型を int64 に変換する
od = od.astype({'origin': np.int64, 'destination': np.int64})
```

作成した OD を確認してみます．ある立ち寄り地点の着地（destination）が次の地点の発地（'origin'）になっています．

```
od.head(10)
```

```
         lat         lng   alt            datetime  uid    leaving_datetime  \
0  32.791131  130.731538  37.0 2016-08-31 08:46:21    0 2016-08-31 08:55:21
1  32.791495  130.734562  47.0 2016-08-31 08:55:21    0 2016-08-31 09:17:39
2  32.790454  130.735594  40.0 2016-08-31 09:17:39    0 2016-08-31 09:23:52
3  32.790172  130.733478  47.0 2016-08-31 09:23:52    0 2016-08-31 09:35:50
4  32.781906  130.741104  41.0 2016-08-31 09:43:11    0 2016-08-31 10:06:32
5  32.764800  130.754235  36.0 2016-08-31 10:14:50    0 2016-08-31 10:35:20
6  32.754117  130.771256  35.0 2016-08-31 10:42:07    0 2016-08-31 11:00:52
```

```
7  32.756231  130.779212  35.0 2016-08-31 11:02:14    0 2016-08-31 11:19:02
8  32.774180  130.783923  31.0 2016-08-31 11:29:36    0 2016-08-31 11:43:57
9  32.792864  130.793660  51.0 2016-08-31 12:01:50    0 2016-08-31 12:13:14
```

```
   cluster                    geometry  origin  destination
0        1  POINT (130.73154 32.79113)      70           70
1        1   POINT (130.73456 32.7915)      70           70
2        1  POINT (130.73559 32.79045)      70           70
3        1  POINT (130.73348 32.79017)      70           70
4       10   POINT (130.7411 32.78191)      70           70
5        9   POINT (130.75423 32.7648)      70           80
6        5  POINT (130.77126 32.75412)      80           80
7        5  POINT (130.77921 32.75623)      80           81
8        8  POINT (130.78392 32.77418)      81           81
9        4  POINT (130.79366 32.79286)      81           81
```

発着地が揃った立ち寄り地点データが作成できたら，発着地の組み合わせごとにカウントしして縦持ちの OD 行列を作成します．

作成した OD 行列は odc として保存します．

```
# origin と destination の組み合わせごとのカウントを取得する
odc = od.groupby(['origin', 'destination']).size().rename('count').reset_index()
```

ODC も確認してみます．

```
odc.head(10)
```

```
   origin  destination  count
0      24           12      1
1      33           33      1
2      33           59      1
3      59           59      2
4      59           60      1
5      59           83      1
6      59          141      1
7      59          149      1
8      60           59      2
9      60           60     13
```

ちゃんとカウントできているようです．

■2.3.1.5　**フローデータの作成**　OD 行列（odc）の origin から destination へラインを引いてフローを作成します．origin と destination の座標は，テッセレーションの対応するメッシュの代表点とします．

テッセレーションの各メッシュの代表点の座標を求め,cts_cent に格納します．

```
# Tessellation の代表点の座標を取得する
# 重心だと，Tessellation の形状によっては外に出てしまうことがあるため，
# 代表点（representative_point）を使用する
cts_cent = gpd.GeoDataFrame(tessellation.representative_point())\
  .rename(columns={0: 'geometry'})\
  .set_geometry('geometry')\
  .set_crs(epsg=4326)
```

確認します．

```
cts_cent.head(10)
```

```
                     geometry
0   POINT (130.47822 32.6271)
1  POINT (130.47822 32.66492)
2  POINT (130.47822 32.70272)
3  POINT (130.47822 32.74051)
4  POINT (130.47822 32.77828)
5  POINT (130.47822 32.81604)
6  POINT (130.47822 32.85378)
7   POINT (130.47822 32.8915)
8  POINT (130.47822 32.92921)
9   POINT (130.47822 32.9669)
```

テッセレーションの代表点（cts_cent）とテッセレーション（tesselation）を空間結合し，テッセレーションの代表点（cts_cent）にの tile_ID を加えます．

```
# Tessellation の代表点に tile_ID を空間結合する
cts_id = cts_cent.sjoin(tessellation, how='left', predicate='within')\
  .astype({'tile_ID': np.int64})\
  .drop('index_right', axis=1)
```

確認します．

```
cts_id.head(10)
```

```
                    geometry  tile_ID
0   POINT (130.47822 32.6271)        0
1  POINT (130.47822 32.66492)        1
2  POINT (130.47822 32.70272)        2
3  POINT (130.47822 32.74051)        3
4  POINT (130.47822 32.77828)        4
5  POINT (130.47822 32.81604)        5
6  POINT (130.47822 32.85378)        6
7   POINT (130.47822 32.8915)        7
8  POINT (130.47822 32.92921)        8
9   POINT (130.47822 32.9669)        9
```

OD の発地（origin）と着地（destination）に座標を結合し，それぞれ odc_o と odc_d として保存します.

```
# origin の座標を結合
odc_o = gpd.GeoDataFrame(odc.merge(cts_id, left_on='origin', right_on='tile_ID', how='left'))
# destination の座標を結合
odc_d = gpd.GeoDataFrame(odc.merge(cts_id, left_on='destination', right_on='tile_ID', how='left'))
```

結果を確認します.

```
odc_o.head(10)
```

```
   origin  destination  count                    geometry  tile_ID
0      24           12      1  POINT (130.56805 32.70272)       24
1      33           33      1   POINT (130.61297 32.6271)       33
2      33           59      1   POINT (130.61297 32.6271)       33
3      59           59      2   POINT (130.7028 32.77828)       59
4      59           60      1   POINT (130.7028 32.77828)       59
5      59           83      1   POINT (130.7028 32.77828)       59
6      59          141      1   POINT (130.7028 32.77828)       59
7      59          149      1   POINT (130.7028 32.77828)       59
8      60           59      2   POINT (130.7028 32.81604)       60
9      60           60     13   POINT (130.7028 32.81604)       60
```

odc_d も確認します.

```
odc_d.head(10)
```

	origin	destination	count	geometry	tile_ID
0	24	12	1	POINT (130.52313 32.66492)	12
1	33	33	1	POINT (130.61297 32.6271)	33
2	33	59	1	POINT (130.7028 32.77828)	59
3	59	59	2	POINT (130.7028 32.77828)	59
4	59	60	1	POINT (130.7028 32.81604)	60
5	59	83	1	POINT (130.79263 32.85378)	83
6	59	141	1	POINT (131.01721 32.9669)	141
7	59	149	1	POINT (131.06212 32.85378)	149
8	60	59	2	POINT (130.7028 32.77828)	59
9	60	60	13	POINT (130.7028 32.81604)	60

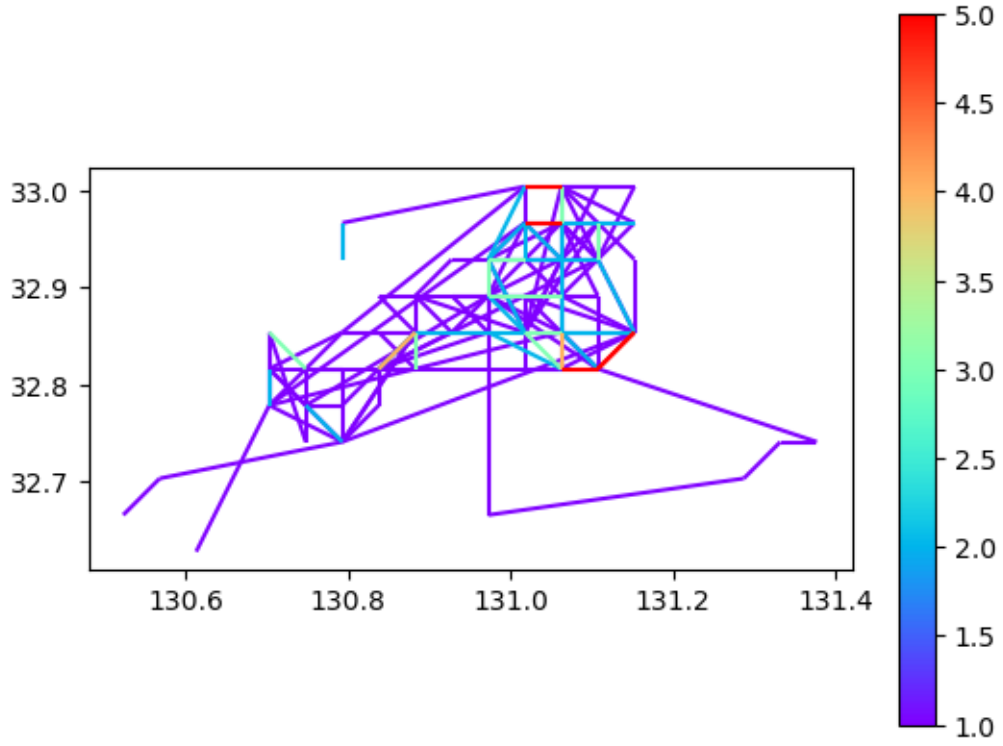
odc_o から odc_d にラインを引き，フローを作成（flow_gdf）します.

```
# flowを示すGeoDataFrameを作成する
flow_gdf = gpd.GeoDataFrame(columns=['geometry'], crs=odc_o.crs)
# originとdestinationの座標の組からflowを作成する
for ori, des in zip(odc_o.geometry, odc_d.geometry):
  # originとdestinationの座標からflowを作成
  segment = LineString([[ori.x, ori.y], [des.x, des.y]])
  # flowをGeoDataFrameに追加
  flow_gdf = pd.concat([flow_gdf, gpd.GeoDataFrame({'geometry': [segment]})])
# flowにcountを追加
flow_gdf['count'] = list(odc['count'])
```

作成したフローを表示してみましょう.

```
flow_gdf.sort_values('count', ascending=True)\
  .plot(column='count', cmap='rainbow', legend=True, vmax=5)
```

```
<Axes: >
```



インタラクティブマップでも表示してみます.

```
mapcenter = [32.92, 131.1]

m = folium.Map(
    location=mapcenter,
    zoom_start=11
)

# 地理院タイル（淡色地図）を追加
# '全国最新写真（シームレス）'も面白いかも
# https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg
tile_layer = folium.TileLayer(
    tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
    attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
        地理院タイル</a>',
    name=' 地理院地図'
)
tile_layer.add_to(m)

# カスタム CSS を追加してタイルレイヤーをモノクロ化
```

```
m.get_root().html.add_child(folium.Element("""
<style>
    .leaflet-tile {
        filter: grayscale(100%);
    }
</style>
"""))

tessellation.explore(
    m=m,
    name='Tessellation',
    style_kwds={'fillColor': 'none'}
)

cstops_gdf.explore(
    m=m,
    column='cluster',
    name='Clustered Stops',
    style_kwds={'weight': 5}
)

flow_gdf.explore(
    m=m,
    column='count',
    style_kwds={'weight': 3},
    name='flow',
    vmax=5
)

folium.LayerControl().add_to(m)
display(m)
```

```
<folium.folium.Map at 0x3b85b2c50>
```

■2.3.1.6　**結果を GeoJSON に保存する**　立ち寄り地（cstops_gdf），テッセレーション（tesselalation），フロー（flow_gdf）を GeoJSON でファイル出力します．すべて GeoDataFrame なので，to_file メソッドでファイル出力できます．今回は GeoJSON で出力しましたが，GeoPackage やシェープファイルでの出力も可能です．ただし，ファイル形式によってはデータの型が保持されないことがあるので注意が必要です．例

えばシェープファイルは時刻を保持する型がありません.

```
# 立ち寄り地（cstops_gdf）を GeoJSON で保存する
cstops_gdf.to_file(base_dir + 'cstops.geojson', driver='GeoJSON')
# Tessellation を GeoJSON で保存する
tessellation.to_file(base_dir + 'mesh_tessellation.geojson', driver='GeoJSON')
# フロー（flow_gdf）を GeoJSON で保存する
flow_gdf.to_file(base_dir + 'mesh_flow.geojson', driver='GeoJSON')
```

■2.3.1.7　**小地域単位のフローデータの作成**　正方メッシュのテッセレーションではなく，小地域をテッセレーションにした場合も実行してみましょう．ファイル r2ka.fgb は FlatGeoBuf 型式の小地域ポリゴンで，e-Stat からダウンロードした熊本県と宮崎県のデータから作成しました.

　立ち寄り地点は正方メッシュのテッセレーションのフロー作成の時に使ったものをそのまま利用します．一気に実行するようになっているので，プログラムを読み取ってみてください.

```
# 対象地域の小地域ポリゴンから tessellation を作成する

# 熊本県・宮崎県の小地域ポリゴンの parquet からデータを読み込み，
# 緯度経度を WGS84 に変換して GeoDataFrame に格納する
tessellation = gpd.read_file(base_dir + 'r2ka.fgb')\
  .to_crs(epsg=4326)\
  .reset_index()

# 必要な列のみを残す
tessellation = tessellation[['KEY_CODE', 'PREF_NAME', 'CITY_NAME', 'S_NAME', 'geometry']]
# index を tile_ID にする
tessellation['tile_ID'] = tessellation.index

# 空間結合で発地を設定する．発地 ID（origin の値）は Tessellation の tile_ID を使用する．
temp = cstops_gdf.to_crs(epsg=4326).sjoin(tessellation, how='left', predicate='within')\
  .rename(columns={'tile_ID': 'origin'})\
  .drop('index_right', axis=1)

# 発地（origin）の型を float64 に変換
temp.astype({'origin': np.float64})
# 着地（destination）の列を追加
temp['destination'] = np.float64(0)
```

```
# 発着地 ID を含む GeoDataFrame を作成
od = gpd.GeoDataFrame()

# uid ごとに処理するため，uid のユニーク値をリスト化する
uids = temp['uid'].unique().astype(int).tolist()
# uid ごとに処理を行う
for uid in uids:
  # uid ごとのデータを temp から抽出
  temp_origin = temp[temp['uid'] == uid]
  # 着地のリストを作成する．着地は発地を 1 つずらしたもの
  temp_dest = np.roll(temp_origin['origin'], -1).astype(np.float64)
  # 最後のデータは着地がないため，nan にする
  temp_dest[-1] = np.nan
  # 着地の列を追加する
  temp_origin.loc[:, 'destination'] = temp_dest
  # 結果を od に追加する
  od = pd.concat([od, temp_origin])
# 着地のない行（着地が nan になっている行）を削除する
od.dropna(inplace=True)
# origin と destination の型を int64 に変換する
od = od.astype({'origin': np.int64, 'destination': np.int64})

# origin と destination の組み合わせごとのカウントを取得する
odc = od.groupby(['origin', 'destination']).size().rename('count').reset_index()

# Tessellation の代表点の座標を取得する
cts_cent = gpd.GeoDataFrame(tessellation.representative_point())\
  .rename(columns={0: 'geometry'})\
  .set_geometry('geometry')\
  .set_crs(epsg=4326)

# Tessellation の代表点に tile_ID を空間結合する
cts_id = cts_cent.sjoin(tessellation, how='left', predicate='within')\
  .astype({'tile_ID': np.int64})\
  .drop('index_right', axis=1)

# origin の座標を結合
odc_o = gpd.GeoDataFrame(odc.merge(cts_id, left_on='origin', right_on='tile_ID', how='left'))
# destination の座標を結合
```

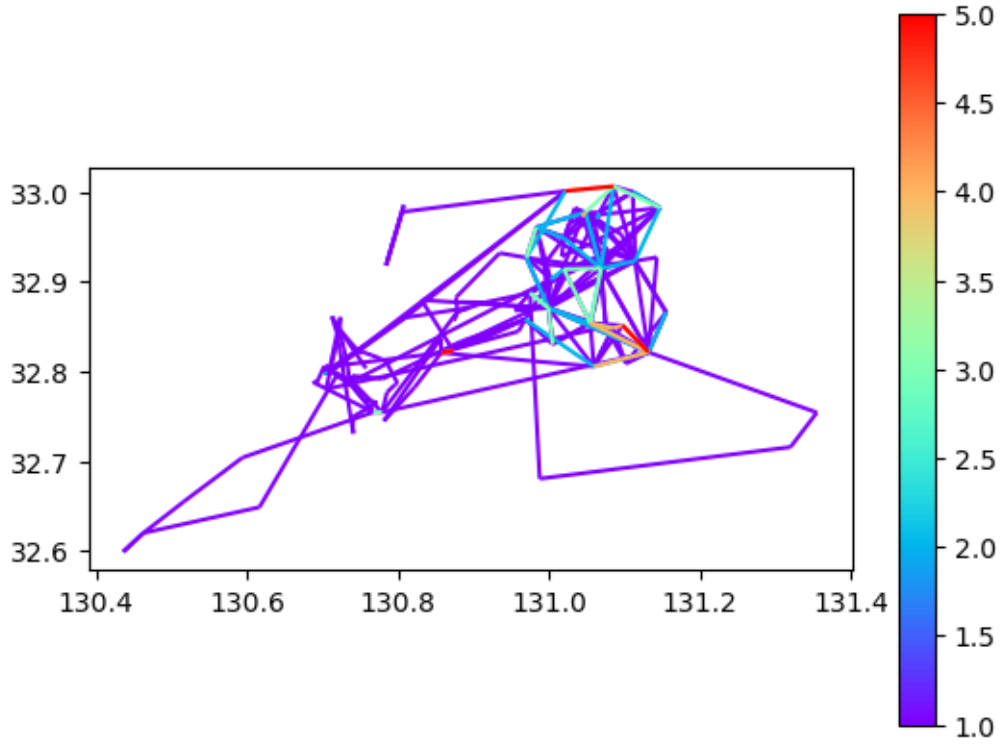
```
odc_d = gpd.GeoDataFrame(odc.merge(cts_id, left_on='destination', right_on='tile_ID', how='left'))

# flowを示すGeoDataFrameを作成する
flow_gdf = gpd.GeoDataFrame(columns=['geometry'], crs=odc_o.crs)
# originとdestinationの座標の組からflowを作成する
for ori, des in zip(odc_o.geometry, odc_d.geometry):
  # originとdestinationの座標からflowを作成
  segment = LineString([[ori.x, ori.y], [des.x, des.y]])
  # flowをGeoDataFrameに追加
  flow_gdf = pd.concat([flow_gdf, gpd.GeoDataFrame({'geometry': [segment]})])
# flowにcountを追加
flow_gdf['count'] = list(odc['count'])
```

とりあえずプロットして結果を確認してみましょう.

```
flow_gdf.sort_values('count', ascending=True)\
  .plot(column='count', cmap='rainbow', legend=True, vmax=5)
```

```
<Axes: >
```



インタラクティブマップでも表示してみます.

```
mapcenter = [32.92, 131.1]

m = folium.Map(
    location=mapcenter,
    zoom_start=11
)

# 地理院タイル（淡色地図）を追加
# '全国最新写真（シームレス）'も面白いかも
# https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg
tile_layer = folium.TileLayer(
    tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
    attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
        地理院タイル</a>',
    name=' 地理院地図'
)
tile_layer.add_to(m)

# カスタム CSS を追加してタイルレイヤーをモノクロ化
m.get_root().html.add_child(folium.Element("""
<style>
    .leaflet-tile {
        filter: grayscale(100%);
    }
</style>
"""))

tessellation.explore(
    m=m,
    name='Tessellation',
    style_kwds={'fillColor': 'none'}
)

cstops_gdf.explore(
    m=m,
    column='cluster',
    name='Clustered Stops',
    style_kwds={'weight': 5}
)
```

```
flow_gdf.sort_values('count', ascending=True).explore(
    m=m,
    column='count',
    cmap='rainbow',
    style_kwds={'weight': 3, 'opacity': 0.7},
    name='flow',
    vmax=5
)

folium.LayerControl().add_to(m)
display(m)
```

```
<folium.folium.Map at 0x3b5423350>
```

2.3.1.7.1 **立ち寄り地，Tessellation，flowline を GeoJSON で保存する** こちらの結果も GeoJSON でファイル保存します.

```
# 立ち寄り地（cstops_gdf）を GeoJSON で保存する
cstops_gdf.to_file(base_dir + 'cstops.geojson', driver='GeoJSON')
# Tessellation を GeoJSON で保存する
tessellation.to_file(base_dir + 'area_tessellation.geojson', driver='GeoJSON')
# フロー（flow_gdf）を GeoJSON で保存する
flow_gdf.to_file(base_dir + 'area_flow.geojson', driver='GeoJSON')
```

3 MAS データの分析

MAS はマルチエージェントシミュレーションの略で，個々に行動ルールを持ち互いに影響し合う多数のエージェントが行動することにより，全体としてどのような振る舞いが生じるかをシミュレートするものです.

この章では，熊本県阿蘇市内牧地区を対象に行った MAS の結果を用いて，データのクリーニングおよび縮約を行い，行動のボトルネックとなる場所を抽出することを目的とします.

MAS の結果をアニメーション化したものは，以下のリンクからご覧下さい.

講習用動画（YouTube）

使用した MAS プログラムについては，GIS Day in 東京 2023 E コースマルチエージェントシミュレーションを使った人の動きのシミュレーション」をご覧下さい.

3.1 ファイルの読み込み

MAS の結果は FlatGeoBuf 型式で保存されています．このファイルを読み込んで GeoDataFrame にします．

```
# 読み込むファイル名
mas_file = 'gisday2024_agent_vars.fgb'
# ファイルの読み込み
indata = gpd.read_file(base_dir + mas_file).to_crs(epsg=4326)
# ゴール後のワープ先を削除
mas_data = indata[indata['Goal'] == 0]
# スタート前の滞留を削除
mas_data = mas_data[mas_data['start_move']==0]
```

読み込んだデータを確認します．

```
len(mas_data)
```

```
468482
```

読み込んだデータを TrajDataFrame に変換します．

```
df = pd.DataFrame()
df['lon'] = mas_data.geometry.x
df['lat'] = mas_data.geometry.y
df['datetime'] = mas_data['Time']
df['id'] = mas_data['AgentID']
df['evac_no'] = mas_data['Evac_no']

tdf = skmob.TrajDataFrame(
  df,
  latitude='lat',
  longitude = 'lon',
  datetime='datetime',
  user_id='id'
)
```

Trajectory を表示します．

```
tdf.plot_trajectory(zoom=12, weight=3, opacity=0.9, max_users=1000, start_end_markers=False)
```

3.2 経路データのクリーニング

時速 100km 以上とループを削除します.

```
ftdf = filtering.filter(tdf, max_speed_kmh=100, include_loops=True)
```

```
print(f'オリジナル: {len(tdf)}')
print(f'ループを含む速度フィルタリング後: {len(ftdf)}')
```

オリジナル: 468482
ループを含む速度フィルタリング後: 460887

プロットして確認します.

```
ftdf.plot_trajectory(zoom=12, weight=3, opacity=0.9, max_users=1000, start_end_markers=False)
```

```
/Users/daichi/.pyenv/versions/miniforge3-24.9.2-0/envs/gisday2024/lib/python3.13/site-
packages/skmob/core/trajectorydataframe.py:569: UserWarning: If necessary, trajectories will be down
sampled to have at most `max_points` points. To avoid this, specify `max_points=None`.
  return plot.plot_trajectory(self, map_f=map_f, max_users=max_users, max_points=max_points, style_f
```

```
<folium.folium.Map at 0x34d202050>
```

3.3 経路データからの滞留ポイント抽出

半径 20m 以内に 2 分以上滞留していたものを検出します.

```
stdf = detection.stay_locations(
  ftdf, stop_radius_factor=0.5, minutes_for_a_stop=2.0,
  spatial_radius_km=0.02, leaving_time=True
)
```

結果を確認します.

```
stdf.head(10)
```

	lng	lat	datetime	uid	evac_no	leaving_datetime
0	131.041725	32.975735	2020-01-01 00:15:30	0	0	2020-01-01 00:17:45
1	131.041725	32.975690	2020-01-01 00:17:45	0	0	2020-01-01 00:21:05
2	131.041457	32.975826	2020-01-01 00:21:05	0	0	2020-01-01 00:25:15

```
3  131.041297  32.975826 2020-01-01 00:25:15   0        0 2020-01-01 00:28:10
4  131.039157  32.975511 2020-01-01 00:22:45   1        0 2020-01-01 00:26:40
5  131.039157  32.975646 2020-01-01 00:26:40   1        0 2020-01-01 00:28:45
6  131.039157  32.975466 2020-01-01 00:31:15   1        0 2020-01-01 00:43:20
7  131.039157  32.975736 2020-01-01 00:43:20   1        0 2020-01-01 00:50:30
8  131.039157  32.975646 2020-01-01 00:51:10   1        0 2020-01-01 00:57:55
9  131.039264  32.975826 2020-01-01 00:57:55   1        0 2020-01-01 01:00:50
```

滞留ポイント数も確認します.

```
print(f'滞留ポイント数: {len(stdf)}')
```

滞留ポイント数: 3514

GeoDataFrame に変換して滞留ポイントをインタラクティブマップに表示します.

```
stay_gdf = stdf.to_geodataframe()

mapcenter = [32.975, 131.04]

m = folium.Map(
  location=mapcenter,
  tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
  attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">地理院タイル</a>',
  name='淡色地図',
  zoom_start=15
)

stay_gdf.to_crs(epsg=3857).explore(
  m=m,
  color = 'red',
  style_kwds={'weight': 5}
)
```

```
/Users/daichi/.pyenv/versions/miniforge3-24.9.2-0/envs/gisday2024/lib/python3.13/site-
packages/pyproj/crs/crs.py:143: FutureWarning: '+init=<authority>:<code>' syntax is deprecated. '<au
order-changes-in-proj-6
  in_crs_string = _prepare_from_proj_string(in_crs_string)
```

```
<folium.folium.Map at 0x34d200e50>
```

速度を計算するため TrajDataFrame を MovingDataFrame の TrajectoryCollection に変換します.

```
ftdf_gdf = ftdf.to_geodataframe()
collection = mpd.TrajectoryCollection(ftdf_gdf, 'uid', t='datetime')
```

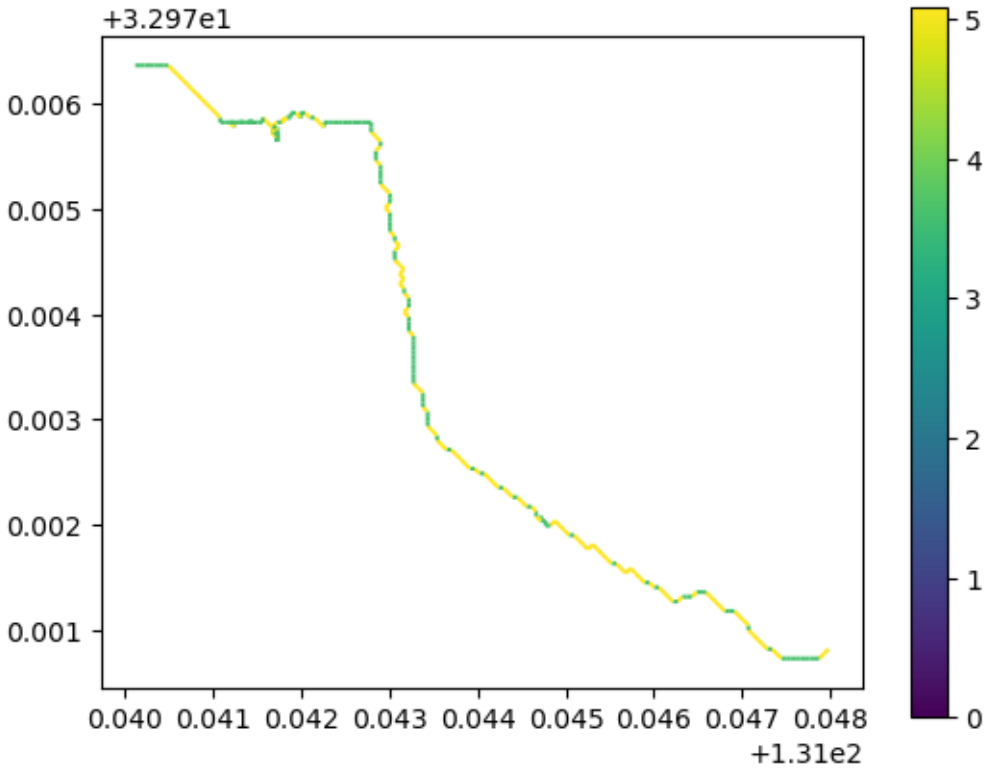
TrajectoryCollection に対して速度を求めます.

```
collection.add_speed(overwrite=True, units=("km", "h"))
```

すべての trajectory の速度をプロットすると大事になるから最初の一つだけプロットします.

```
collection.trajectories[0].plot(column="speed", legend=True)
```

```
<Axes: >
```



3.4 経路のスムージングと集計

MovingPandas ではデータの間引きではなく関数で補間することによる経路のスムージングができます. 試しに経路の一つにパラメーターを変えてスムージングを行い, 結果を比較してみます.

```
# デフォルトの値でスムージングしてみる
smooth_01 = mpd.KalmanSmootherCV(collection.trajectories[12]).smooth(
    process_noise_std=0.5,
```

```
    measurement_noise_std=1
)
# スムージングしてみる
smooth_02 = mpd.KalmanSmootherCV(collection.trajectories[12]).smooth(
    process_noise_std=10,
    measurement_noise_std=1
)
# スムージングしてみる
smooth_03 = mpd.KalmanSmootherCV(collection.trajectories[12]).smooth(
    process_noise_std=0.5,
    measurement_noise_std=50
)
# スムージングしてみる
smooth_04 = mpd.KalmanSmootherCV(collection.trajectories[12]).smooth(
    process_noise_std=10,
    measurement_noise_std=50
)
```

スムージング前のデータとスムージング後のデータの速度をプロットして比較します.

```
mapcenter = [32.92, 131.1]

m = folium.Map(
    location=mapcenter,
    zoom_start=11
)

# 地理院タイル（淡色地図）を追加
# '全国最新写真（シームレス）'も面白いかも
# https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg
tile_layer = folium.TileLayer(
    tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
    attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
        地理院タイル</a>',
    name='地理院地図'
)
tile_layer.add_to(m)

# カスタムCSSを追加してタイルレイヤーをモノクロ化
m.get_root().html.add_child(folium.Element("""
```

```
<style>
    .leaflet-tile {
        filter: grayscale(100%);
    }
</style>
"""))
collection.trajectories[12].explore(
    m=m,
    cmap='rainbow',
    style_kwds={'weight': 5, 'opacity': 0.7},
    column='speed',
    name='original'
)

smooth_01.explore(
    m=m,
    cmap='rainbow',
    style_kwds={'weight': 5, 'opacity': 0.7},
    column='speed',
    name='sm01, pn=0.5, mn=1',
    legend=False
)

smooth_02.explore(
    m=m,
    cmap='rainbow',
    style_kwds={'weight': 5, 'opacity': 0.7},
    column='speed',
    name='sm02, pn=10, mn=1',
    legend=False
)

smooth_03.explore(
    m=m,
    cmap='rainbow',
    style_kwds={'weight': 5, 'opacity': 0.7},
    column='speed',
    name='sm03, pn=0.5, mn=50',
    legend=False
```

```
)

smooth_04.explore(
    m=m,
    cmap='rainbow',
    style_kwds={'weight': 5, 'opacity': 0.7},
    column='speed',
    name='sm04, pn=10, mn=50',
    legend=False
)

folium.LayerControl().add_to(m)
display(m)
```

```
<folium.folium.Map at 0x34d201850>
```

データ全体をスムージングすると 18 分くらいかかるため，ここでは全体へのスムージングは行わない．行う場合は以下を実行する．

```
# process_noise=0.5，measurement_noise=50 でスムージングする
smooth = mpd.KalmanSmootherCV(generalized_gdf).smooth(
    process_noise_std=0.5,
    measurement_noise_std=50
)
```

スムージングの代わりに generalizer でデータを間引いてみます．MinTimeDeltaGeneralizer を使うと時間で間引くことができます．ここでは 30 秒間隔で間引きます．

```
generalized = mpd.MinTimeDeltaGeneralizer(collection).generalize(tolerance=timedelta(seconds=30))
```

時間で間引いた経路の一部を地図表示してみる．

```
mapcenter = [32.92, 131.1]

m = folium.Map(
    location=mapcenter,
    zoom_start=11
)

# 地理院タイル（淡色地図）を追加
# '全国最新写真（シームレス）'も面白いかも
```

```
# https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg
tile_layer = folium.TileLayer(
    tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
    attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
        地理院タイル</a>',
    name=' 地理院地図'
)
tile_layer.add_to(m)

# カスタム CSS を追加してタイルレイヤーをモノクロ化
m.get_root().html.add_child(folium.Element("""
<style>
    .leaflet-tile {
        filter: grayscale(100%);
    }
</style>
"""))

collection.trajectories[12].explore(
    m=m,
    cmap='rainbow',
    style_kwds={'weight': 5, 'opacity': 0.7},
    column='speed',
    name='original'
)

generalized.trajectories[12].explore(
    m=m,
    cmap='rainbow',
    style_kwds={'weight': 5, 'opacity': 0.7},
    column='speed',
    name='generalized',
    legend=False
)

folium.LayerControl().add_to(m)
display(m)
```

```
<folium.folium.Map at 0x34d201750>
```

時間で間引いたデータをスムージングしてみましょう.

```
# process_noise=0.5, measurement_noise=50 でスムージングする
generalized_smooth = mpd.KalmanSmootherCV(generalized).smooth(
    process_noise_std=0.5,
    measurement_noise_std=50
)
```

間引いてスムージングした経路をマップ表示してみます.

```
mapcenter = [32.92, 131.1]

m = folium.Map(
    location=mapcenter,
    zoom_start=11
)

# 地理院タイル（淡色地図）を追加
# '全国最新写真（シームレス）'も面白いかも
# https://cyberjapandata.gsi.go.jp/xyz/seamlessphoto/{z}/{x}/{y}.jpg
tile_layer = folium.TileLayer(
    tiles='https://cyberjapandata.gsi.go.jp/xyz/pale/{z}/{x}/{y}.png',
    attr='&copy; <a href="https://maps.gsi.go.jp/development/ichiran.html">\
        地理院タイル</a>',
    name=' 地理院地図'
)
tile_layer.add_to(m)

# カスタム CSS を追加してタイルレイヤーをモノクロ化
m.get_root().html.add_child(folium.Element("""
<style>
    .leaflet-tile {
        filter: grayscale(100%);
    }
</style>
"""))

collection.trajectories[12].explore(
    m=m,
    cmap='rainbow',
```

```
    style_kwds={'weight': 5, 'opacity': 0.7},
    column='speed',
    name='original'
)

generalized.trajectories[12].explore(
    m=m,
    cmap='rainbow',
    style_kwds={'weight': 5, 'opacity': 0.7},
    column='speed',
    name='generalized',
    legend=False
)

generalized_smooth.trajectories[12].explore(
    m=m,
    cmap='rainbow',
    style_kwds={'weight': 5, 'opacity': 0.7},
    column='speed',
    name='smooth',
    legend=False
)

folium.LayerControl().add_to(m)
display(m)
```

```
<folium.folium.Map at 0x34d201350>
```

3.5 計算結果を出力する

generalize した経路と smooth した経路を GeoDataFrame に変換し，滞留ポイントと合わせて保存します．これらのデータは一つの GeoPackage に別レイヤーとして保存します．

```
# GeoDataFrame に変換する
generalized_gdf = generalized.to_line_gdf()
generalized_smooth_gdf = generalized_smooth.to_line_gdf()

# GeoDataFrame を GeoPackage として保存する
generalized_gdf.to_file(base_dir + 'MAS_analysis.gpkg', layer='generalized', driver='GPKG')
```

```
generalized_smooth_gdf.to_file(base_dir + 'MAS_analysis.gpkg', layer='generalized_smooth', driver='G
stay_gdf.to_file(base_dir + 'MAS_analysis.gpkg', layer='stay', driver='GPKG')
```

GIS Day in 東京 2024
Eコース資料

scikit-mobilityを用いた
行動パターンの分析入門

2024年12月14日発行

著者：中山大地

発行：東京都立大学 都市環境学部 地理学教室
東京都八王子市南大沢1-1