Python による流跡線解析

田崎盛也

Analysis of Trajectories using Python

Moriya TASAKI

要旨： Python の PySPLIT というライブラリでは Python 上で NOAA（米国海洋気象局）の HYSPLIT の呼び出しと実行により多数の流跡線の作成が可能となる．解析した流跡線は PySPLIT 利用により必要な期間の流跡線を抽出して表示が可能となる他，流跡線解析時に取得できる降雨情報などの情報をプロットすることも可能である．多数の流跡線の作成では解析に不適切な流跡線を手作業で確認することは難しいが，PySPLIT では逆解析流跡線を用いた誤差評価で誤差の大きい不適切な流跡線を除外できる．また，Python の地理解析ライブラリを用いることで特定の地域の通過による分類も可能になる．これにより，特定の期間における通過域による割合表示や 1 日毎の区分地域の自動判別に利用することもでき，大気中における物質輸送経路等の経時変化を把握することに役立てることが可能と思われる．

Key words: 沖縄県，流跡線解析，Python，HYSPLIT，PySPLIT

Ⅰ　はじめに

流跡線解析は大気環境の分野ではよく利用されているが，多数の流跡線の解析を行うことやそれらの流跡線データを使用してさらに二次的な解析を行うことは容易ではない．Python のライブラリである PySPLIT[1,2]を用いると Python 上で NOAA（米国海洋気象局）の HYSPLIT の呼び出しと実行により，多数の流跡線の重ね書きのような可視化が可能となる．さらに Python の別のライブラリを用いることにより，流跡線の日付・属性などによる分類も可能となる．今回は PySPLIT 等のライブラリを用いた Python での流跡線解析例を紹介する．

Ⅱ　方法

流跡線解析の対象となる地点と期間について，地点は沖縄県衛生環境研究所（うるま市），期間は 2017 年度から 2020 年度の四半期ごとに微小粒子状物質（$PM_{2.5}$）の成分分析対象期間である 14 日間を対象とし，開始時間は日本標準時で 12 時，18 時，0 時，6 時の 4 つとした．流跡線作成時間は 5 日間（120 時間）とし，気象データは GDAS を用い，出発高度は 500 m，1000 m，1500 m の 3 つとした．

流跡線の作成やその後の解析には Anaconda ディストリビューション(https://www.anaconda.com)により解析環境を構築し，プログラミング言語 Python（3.7.12）と PySPLIT（0.3.6）を使用した．その他のライブラリについては表 1 に示す．また PySPLIT の使用にはデスクトップ版の HYSPLIT が必要となるためこれも導入しておく必要がある．流跡線作成のコードはコード 1 を参照（Python コードはこの報告の付録に掲載）．コード 1 の実行によりより 2688 個の流跡線ファイルを作成した．流跡線作成は PC の性能にもよるが 20 分～1 時間程度で完了する．

表 1．使用した Python と主なライブラリのバージョン．

Name	Version
Python	3.7.12
Numpy	1.21.5
pandas	1.3.5
GeoPandas	0.10.2
Shapely	1.8.0
descartes	1.1.0
Matplotlib	3.5.1
Basemap	1.2.2
PySPLIT	0.3.6

Ⅲ　結果および考察

1．流跡線のプロット

作成した流跡線は PySPLIT の MapDesign クラスによってプロットできる．MapDesign クラスは Matplotlib で地図を描写できるライブラリ Basemap を用いて地図描写と流跡線のプロットを可能にしている．コード 2 で流跡線ファイルの読み込みと地図描画設定，流跡線のプロット時の設定を行い，コード 3 で成分分析の測定日毎に 12 本（6 時間毎 4 本/日×高度別 3 本/日）の流跡線をプロットし，同様な図を 224 日分（一四半期 14 日×4 季節×4

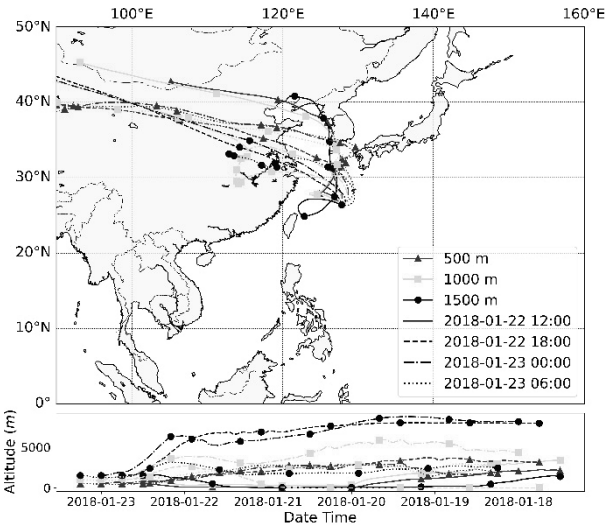

図 1. 2018 年 1 月 22 日 12 時～23 日 6 時の流跡線.

年）作成することができる．コード 2 と 3 によって作成した流跡線のプロットを図 1 に示す．

図 1 では上部に地図，下部に日時と高度を軸とするプロット領域がある．これに色とマーカーの種類で出発高度を，線の種類で作成時間を区別できるようにし，凡例には出発高度と日時の情報を適切に表示した．

プロットできる情報は流跡線の経路だけではなく，その流跡線の持つ情報を可視化することもできる．ここではその一例として流跡線解析時に取得できる降雨情報を扱う．コード 4 は流跡線と降雨情報をプロットするためのコードであり，コード 2 と 4 によって作成したプロットを図 2 に示す．図 2 では図 1 と同様な情報に加えて，降雨があった場合に塗りつぶし円をプロットさせ，降雨量に合わせて色を変化させた．

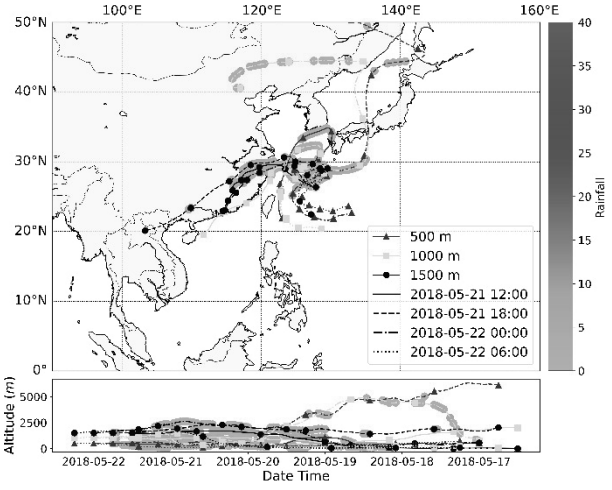

図 2. 2018 年 5 月 21 日 12 時～22 日 6 時の流跡線と降雨情報.

２．流跡線の誤差評価

今回のように流跡線を多数作成する場合，いくつかの流跡線については流跡線の作成自体に問題がある場合や，解析結果に大きな誤差を含むような流跡線が含まれる可能性があるため，作成した流跡線が妥当か検証を行う必要がある．解析結果に問題がある例としては流跡線が途中で解析範囲の高度条件を逸脱している場合などがあげられる．少数の流跡線であれば目視で確認することもできるが，流跡線が多数ある場合は何らかの基準によって評価を自動化することが望ましい．このような評価指標として，流跡線の誤差を用いる方法がある．流跡線の終点を新たな起点とする逆解析を行い，逆解析の終点と元の流跡線の違いから流跡線の誤差を算出できる．コード 5 では流跡線の誤差の計算とプロットの準備を行ってお

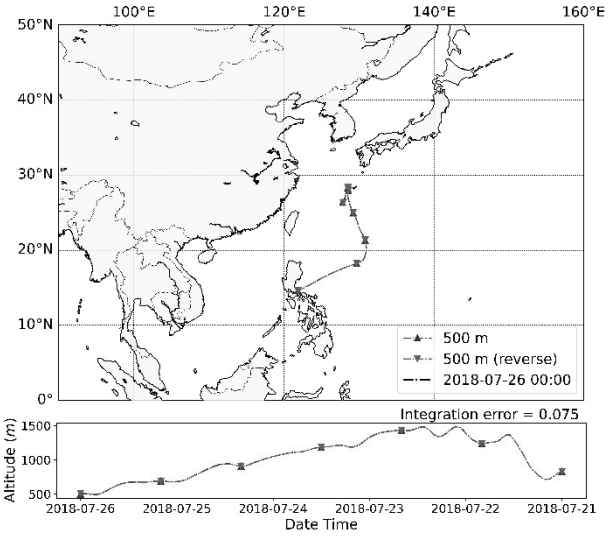

図 3. 2018 年 7 月 26 日 0 時出発高度 500 m の流跡線とその逆解析流跡線.

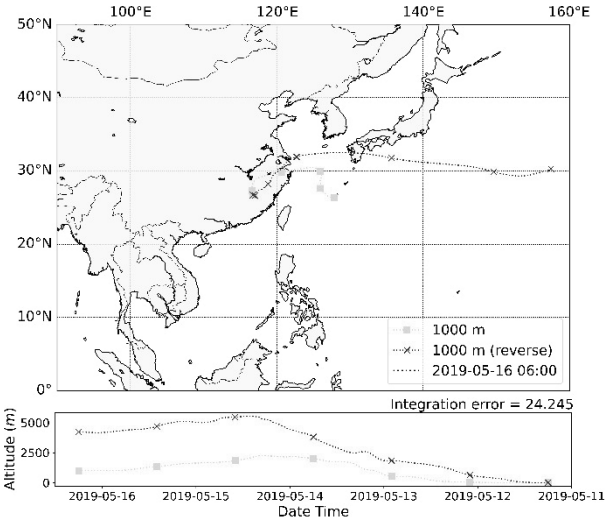

図 4. 2019 年 5 月 16 日 6 時出発高度 1000 m の流跡線とその逆解析流跡線.

り，コード 6 では流跡線と逆解析流跡線をプロットし誤差の表示も行える．

コード 5 と 6 によって作成したプロットを図 3 と図 4 に示す．図 3 は誤差がほとんどない例，図 4 は誤差の大きい例である．凡例の下に記載している「Integration error」は流跡線の誤差である．なお，逆解析の機能は HYSPLIT にもあるが，誤差の計算機能は有していない．誤差は相対誤差を採用した．PySPLIT のドキュメントでは流跡線群の相対誤差の平均値から，標準偏差の 2 倍以上となる流跡線を「Bad」と示しており [2]，この例でもそれに従った．これらの検証により不適切とした 193 個の流跡線を除外し，2495 個の流跡線を有効とした．

３．地理情報ライブラリを活用した地域区分

Python の地理情報ライブラリを利用すれば，流跡線の経路からどの地域を通ったかを自動判別し，通過地域別に分類することが可能となる．今回の例では地域を東アジア（EA），日本（J），東南アジア（SA），太平洋（PO）の 4 つの地域に区分した（図 5）．このような判別を行うためには，前準備としてマスク領域を用いた判別情報やその他流跡線の情報を取り込むデータフレーム（表）を作成する必要がある．表はコード 7－13 までを実行して作成することができ，csv の形式で出力される．この表を利用した通過地域による分類の結果を図 6 に示す．4 つの地域の組み合わせは 16 通りあり，それぞれ類似した経路を持つ流跡線が集まっているのが見て取れる．

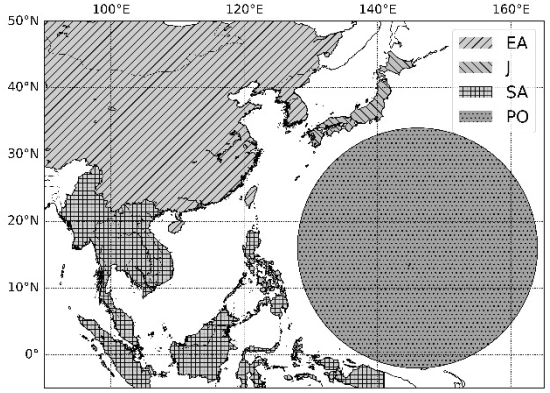

図 5. 流跡線分類のための地域区分設定.

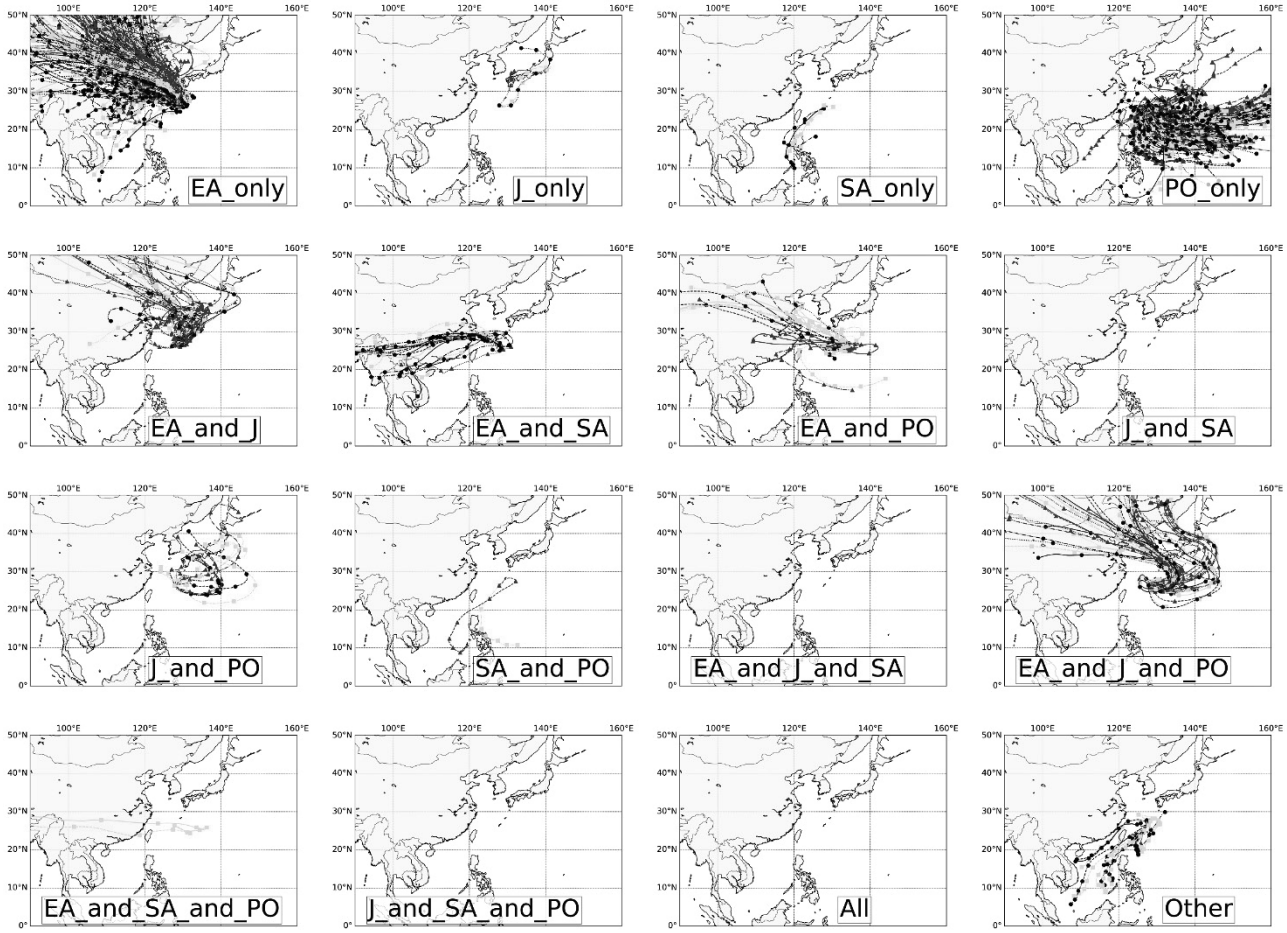

図 6. 流跡線通過地域による 2020 年度流跡線の分類.

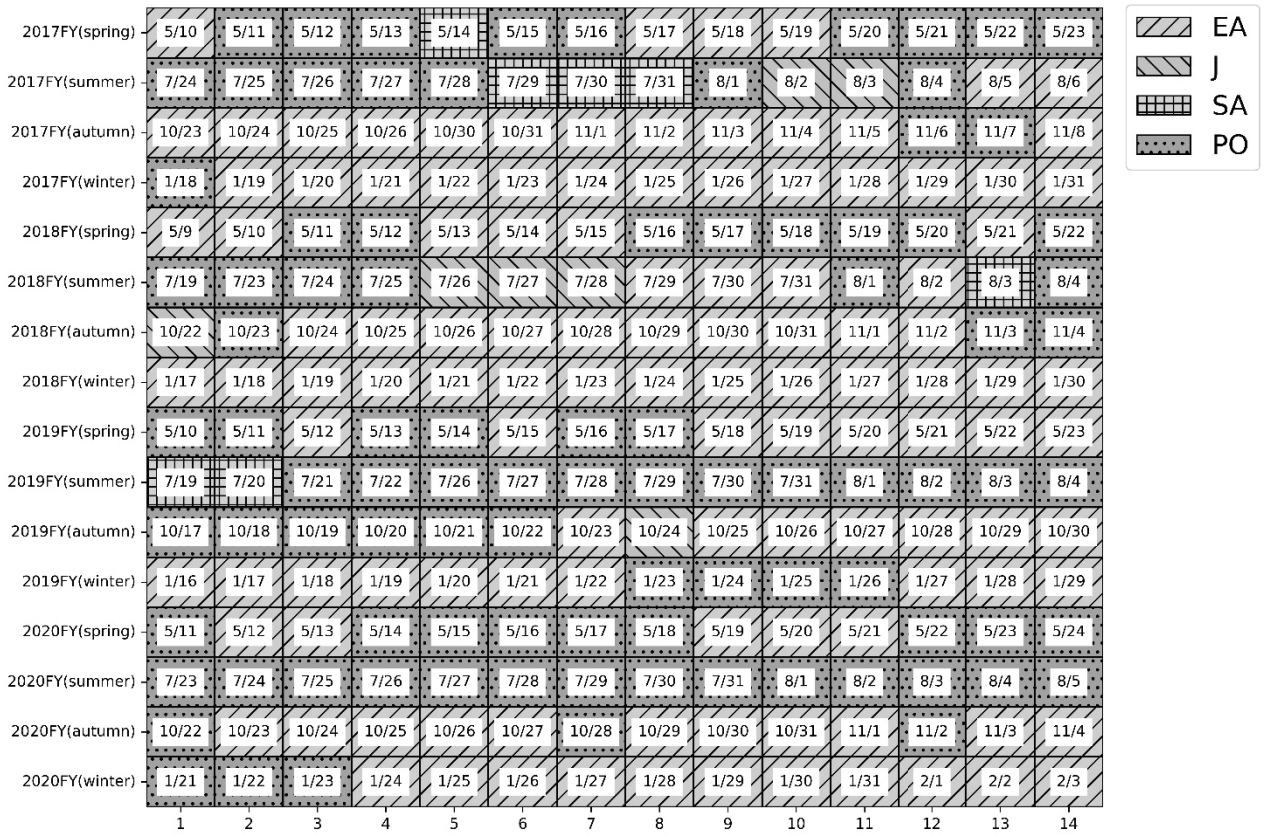

図 7. 流跡線通過地域による 1 日ごとに主な地域由来を表示したヒートマップ.

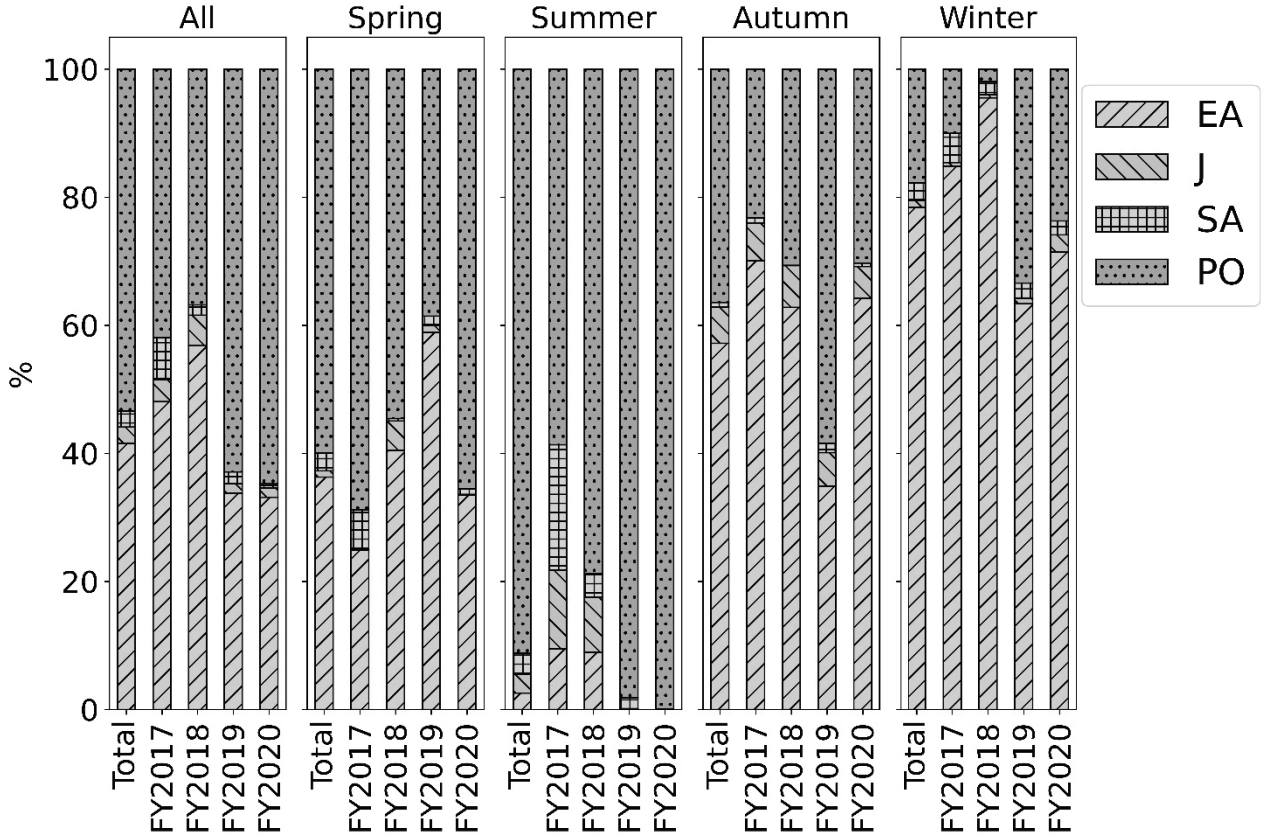

図 8. 流跡線通過地域による地域区分割合表示の棒グラフ（通過点個数による集計）.

このような分類の他に，流跡線の通過点が区分地域内にいくつ点在するかの個数を計数し，その数が多い地域を流跡線の主な由来地域として分類する方法もある．それにより 1 日毎に流跡線が多く通過した地域を，その日の代表として自動判別し，ヒートマップ形式で表示した例が図 7, 1 日の地域代表の判定はせず，全ての区分地域通過点個数をそのまま集計し，年度や季節毎に流跡線の地域由来割合を棒グラフで表示したものが図 8 となる．このように Python のライブラリを活用し，適当な基準を定めれば効率的に地域の分類を行うことが可能となる．

Ⅳ まとめ

Python のライブラリである PySPLIT の利用により，多数の流跡線作成が可能となり，流跡線の経路のみならず降雨などの情報も可視化できる．また，流跡線の誤差の評価により解析結果に問題があるような流跡線を除くことができる. 加えて Python の地理情報ライブラリを用いることで流跡線通過地域による地域区分が可能となり，1 日毎の区分地域の自動分類や特定期間における区分地域通過点個数の集計などで，大気中における物質輸送経路や主な通過地域の経時変化を把握することに役立てることが可能と思われる．

この報告では 224 日分（一四半期 14 日×4 季節×4 年）の期間を対象に，2688 個（224 日×6 時間毎 4 本/日×高度別 3 本/日）の流跡線ファイルを作成したが，コード 1 のコメントにある通年解析用のコードを実行すれば 4 年度分の通年の流跡線が作成でき, 期間を変更すれば 10 年やさらに長期間の解析も可能である．扱う流跡線の数が膨大になっても可視化や誤差評価，地域区分の作業は自動化されているため，計算・処理時間以外の労力は変わらず，多数の流跡線解析を行う際には非常に有用であると考えられる．

Ⅴ 参考文献

1) M. Cross, PySPLIT: A Package for the Generation Analysis and Visualizations of HYSPLIT Air Parcel Trajectories, Proc. 14th Ann. Scientific Computing with Python Conf. (SciPy 15), 2015.

2) M. S. C. Warner. PySPLIT. https://github.com/mscross/pysplit. 2022 年 8 月アクセス

Ⅵ 付録

流跡線作成や流跡線のプロット，作成した流跡線の情報を取り込む表を作成するコードを掲載する．その他，本文中では取り上げなかったコードも掲載している．

コード 1. 複数の流跡線データを一括で作成するコード.

```
# 流跡線を作成するコード。
# pysplit など地理情報関係のライブラリを利用する場合に「KeyError: 'PROJ_LIB'」が表示される場合がある。
# その場合下記のコメントのコードのように、地理情報ライブラリをインポートする前に
# 環境変数'PROJ_LIB'に pyproj ライブラリの場所を設定する必要がある。
# pyproj の場所はインストール方法や環境設定の方法により異なるのでそれに合わせて適切に書き換える。
# これ以降はこの記述は省略する。
# import os
# os.environ['PROJ_LIB'] = 'D:¥Anaconda3¥pkgs¥pyproj-3.2.1-py37h9f67652_5¥Lib¥site-packages¥pyproj'

import pysplit

# ディレクトリ等の指定
working_dir = r'C:/hysplit4/working' # HYSPLIT のワーキングディレクトリ
storage_dir = r'D:/trajectories/eiken_FY2017-FY2020' # 流跡線ファイルの保存先ディレクトリ
meteo_dir = r'F:/gdas' # 気象ファイルの保存ディレクトリ(GDAS を使用)取得元:ftp://ftp.arl.noaa.gov/pub/archives/gdas1/

# basename は地点の名前としておくと複数地点解析する際にファイル名で内容を区別可能。
basename = 'eiken_'

# Python の range 関数の開始は指定した通りだが、終了は指定した値の 1 つ手前で終了することに注意。
y_list = list(range(2017, 2022)) # 年のリスト。開始は 2017 年、最後は 2021 年。

# 月のリスト。やむを得ず測定を一時中断した月があるため、同一年で同じ月が存在する。
m_list = [[5, 7, 8, 10, 10, 11], # 2017 年
          [1, 5, 7, 7, 8, 10, 11], # 2018 年
          [1, 5, 7, 7, 8, 10], # 2019 年
          [1, 5, 7, 8, 10, 11], # 2020 年
          [1, 2]] # 2021 年
# 通年で解析が目的の場合は以下のように変更すればよい。
# m_list = list(range(1, 13))

# 日付のリスト。開始日が指定したい日よりも-1 となっていることと、HYSPLIT の時間は UTC のため、JST とのずれを考慮して日付を設定している。
d_list = [[[9,23],[23,32],[0,6],[22,26],[29,32],[0,8]], # 2017 年
          [[17,32],[8,22],[18,19],[22,32],[0,4],[21,32],[0,4]], # 2018 年
          [[16,30],[9,23],[18,22],[25,32],[0,4],[16,30]], # 2019 年
          [[15,29],[10,24],[22,32],[0,5],[21,32],[0,4]], # 2020 年
          [[20,32],[0,3]]] # 2021 年

# 時間のリスト。UTC による指定。JST は UTC+9h なので、[12, 18, 24, 30]、つまり[0, 6, 12, 18]を指定したこととなる。
hours = [3, 9, 15, 21]

altitudes = [500, 1000, 1500] # 高度のリスト。
location = (26.3769593331371, 127.83470843018222) # 沖縄県衛生環境研究所の緯度経度。
runtime = -120 # 120 時間 = 5 日。符号が正なら前方解析、負なら後方解析となる。

# リストを利用した for ループで指定の日時の流跡線を作成している。
for i,y in enumerate(y_list):
    for j,m in enumerate(m_list[i]):
        years = [y]
        months = [m]
        hours = hours
        monthslice = slice(d_list[i][j][0], d_list[i][j][1], 1) # d_list で指定したデータを作成する設定。

        # 流跡線ファイルの作成。ファイル名の例:eiken_may0500spring2017051003。
        # ファイル名は basename + 3-Letter month + altitude + season + year + month + day + hour(YYYYMMDDHH)。
        pysplit.generate_bulktraj(basename, working_dir, storage_dir, meteo_dir,
                                  years, months, hours, altitudes, location, runtime,
                                  monthslice=monthslice, get_reverse=True, get_clipped=True)

# 通年で解析が目的の場合は追加のライブラリを読み込み、連続日付データを作成し、for ループを以下のように変更すればよい。
# from dateutil import tz
# import pandas as pd

# JST = tz.gettz('Asia/Tokyo') # 日本標準時
# UTC = tz.gettz('UTC') # 協定世界時
## 6 時間おき 4 年度分の連続日付データ作成
# dt_index = pd.date_range(start='2017-04-01 00:00:00', end='2021-03-31 18:00:00', freq='6H', tz=JST)

# for i in range(len(dt_index)):
#     years = [dt_index.tz_convert(UTC).year[i]]
#     months = [dt_index.tz_convert(UTC).month[i]]
#     hours = [dt_index.tz_convert(UTC).hour[i]]
#     monthslice = slice(dt_index.tz_convert(UTC).day[i]-1, dt_index.tz_convert(UTC). day [i], 1)

#     pysplit.generate_bulktraj(basename, working_dir, storage_dir, meteo_dir,
#                               years, months, hours, altitudes, location, runtime,
#                               monthslice=monthslice, get_reverse=True, get_clipped=True)
```

コード2．複数の流跡線をプロットするための準備を行うコード．

```
# 流跡線をプロットするために流跡線ファイルの読み込みとプロットの設定を行うコード
# コード1で流跡線は作成済みであることが必要(これ以降のコードもこれが前提となっている)

# 使用するライブラリをインポート
import pysplit
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import gridspec
from matplotlib.lines import Line2D
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
import re
import itertools as it

# 流跡線ファイルの読み込み
trajgroup = pysplit.make_trajectorygroup(r'D:¥trajectories¥eiken_FY2017-FY2020¥eiken_*')

# 逆解析流跡線ファイルの読み込みとソート
reversetrajgroup = pysplit.make_trajectorygroup(r'D:¥trajectories¥eiken_FY2017-FY2020¥reversetraj¥eiken_*')
reversetrajgroup.trajectories.sort(key=lambda x: (re.search(r'¥d{10}', x.trajid).group(),
                                                  int(re.search(r'(500|1[0-5]00)', x.trajid).group())))
reversetrajgroup.trajids.sort(key=lambda x: (re.search(r'¥d{10}', x).group(),
                                             int(re.search(r'(500|1[0-5]00)', x).group())))

# 地図描画領域や投影方法、地図解像度、描画する最小面積、緯度経度のフォントサイズを指定
mapcorners =  [90, 0, 160, 50] # 値は左下の経度、左下の緯度、右上の経度、右上の緯度を表す
standard_pm = None # 地図の投影方法がcyl(円筒形、メルカトル図法)では必要ないがそれ以外の場合に設定が必要
param_dict = {'projection':'cyl','resolution':'l', 'area_threshold':500, 'latlon_fs':18}

# 地図投影方法をlcc(ランベルト正角円錐図法)にしたときの設定例
# standard_pm = [130, 25, 0, 50] # [``lon_0``, ``lat_0``, ``lat_1``, ``lat_2``]
# param_dict = {'projection':'lcc','resolution':'l', 'area_threshold':500, 'latlon_fs':18}


# 地図のパラメータを設定
bmap_params = pysplit.MapDesign(mapcorners, standard_pm, **param_dict)

# 色、線、マーカーの種類を設定
# この例では流跡線の種類と高さに色とマーカーを対応させ、時間に線の種類を対応させているが
# 色とマーカーを紐づける必要はない
# 流跡線の種類と高さによって色が変わるように設定
color_dict = {'traj':{500.0 : 'blue', 1000.0 : 'gold', 1500.0 : 'black'},
              'reversetraj':{500.0 : 'red', 1000.0 : 'purple', 1500.0 : 'cyan'}}

# 流跡線の時刻によって線種が変わるように設定
ls_dict = {12:'-', 18:'--', 0:'-.', 6:':'}

# 流跡線の色によってマーカーが変わるように設定
marker_dict = {'blue' : '^', 'gold' : 's', 'black' : 'o',
               'red' : 'v', 'purple' : 'x', 'cyan' : '*'}

# 流跡線ファイルから高さの情報を取得しそれに応じて線の色を変更
for i in range(trajgroup.trajcount):
    altitude0 = trajgroup[i].data.geometry.z[0]
    trajgroup[i].trajcolor = color_dict['traj'][altitude0]
    reversetrajgroup[i].trajcolor = color_dict['reversetraj'][altitude0]
```

コード3．流跡線を成分分析測定日毎にプロットするコード．

```
# 流跡線を成分分析測定日毎にプロットするコード。コード2は実行済みであることが必要。
# trajgroupは日時順、高さでソートされている。初めの7つの並びは以下参照。
# "eiken_may0500spring2017051003", "eiken_may1000spring2017051003",
# "eiken_may1500spring2017051003", "eiken_may0500spring2017051009",
# "eiken_may1000spring2017051009", "eiken_may1500spring2017051009",
# "eiken_may0500spring2017051015".

# 変数の定義(コード4でこの変数を参照するが紙面の都合上省略しているため、
# コード4実行の際はコード3全体を実行するか、この変数の定義部分だけを実行する必要がある)
altitudes_num = 3 # 出発高度の種類数
hours_num = 4 # 開始時間の種類数
day_traj_num = 12 # 1日あたりの流跡線の数

# ファイルの並びの規則性により必要な流跡線のまとまりを対象としてプロットが可能である。
# このプロットでは成分分析測定日の1日分をプロットの対象にしているので
# 1日4つの時間×3つの高度で12本分の流跡線をプロットする必要がある。
# そのため流跡線の総数を1日あたりの流跡線の数の12で割っている。
for i in range(trajgroup.trajcount // day_traj_num):
    fig = plt.figure(figsize=[12, 11]) # 図のサイズ指定
    spec = gridspec.GridSpec(ncols=1, nrows=2,hspace=0.01,height_ratios=[5, 1]) # 描画領域を2つに分割
    ax0 = fig.add_subplot(spec[0]) # 地図を描画する領域として使用
    ax1 = fig.add_subplot(spec[1]) # 高度変化を描画する領域として使用
    bmap = bmap_params.make_basemap(ax=ax0) # basemapによる地図の描画
    ax1.invert_xaxis() # x軸(日時、Date Time)の反転
    # 開始日を取得
    startday = pd.to_datetime(trajgroup[i * day_traj_num].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')[0]

    # 12本中初めの3本の流跡線については高さ別の凡例を作成するため処理を分けている。
    for j in range(i * day_traj_num, i * day_traj_num + altitudes_num):
        # 流跡線の日時と高さ情報の取得
        datetime = pd.to_datetime(trajgroup[j].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
        altitude = trajgroup[j].data.geometry.z
        # 地図に流跡線をプロット
        bmap.plot(*trajgroup[j].path.xy, c=trajgroup[j].trajcolor, latlon=True, zorder=20,
                  ax=ax0, ls=ls_dict[datetime[0].hour], marker=marker_dict[trajgroup[j].trajcolor],
                  markersize=8, markevery=20, label=str(int(altitude[0])) +' m')
        # 流跡線の日時と高さ情報をプロット
        ax1.plot(datetime, altitude, color=trajgroup[j].trajcolor, ls=ls_dict[datetime[0].hour],
                 marker=marker_dict[trajgroup[j].trajcolor], markersize=8, markevery=20)

    # 凡例に線の種類による時刻情報を追加。
    handles, _ = ax0.get_legend_handles_labels() # 凡例情報を取得
    lines = [Line2D([0], [0], color='black', linewidth=2, linestyle=l,
                    label=str(pd.to_datetime(
                        trajgroup[i * day_traj_num + altitudes_num * k].data['DateTime'],utc=True).dt.tz_convert(
                        'Asia/Tokyo')[0].strftime('%Y-%m-%d %H:%M'))) for k,l in zip(range(hours_num), ls_dict.values())]
    handles.extend(lines) # 凡例を追加
    legend = ax0.legend(handles=handles, loc='best', fontsize=18) # 凡例の場所とフォントサイズを調整
    legend.set_zorder(level=25) # 凡例が隠れないようにzorderを調整

    # 残りの9本の流跡線については凡例作成処理を除きプロットする。
    for j in range(i * day_traj_num + altitudes_num, i * day_traj_num + day_traj_num):
        datetime = pd.to_datetime(trajgroup[j].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
        altitude = trajgroup[j].data.geometry.z
        bmap.plot(*trajgroup[j].path.xy, c=trajgroup[j].trajcolor, latlon=True, zorder=20,
                  ax=ax0, ls=ls_dict[datetime[0].hour], marker=marker_dict[trajgroup[j].trajcolor],
                  markersize=8, markevery=20)
        ax1.plot(datetime,altitude, color=trajgroup[j].trajcolor, ls=ls_dict[datetime[0].hour],
                 marker=marker_dict[trajgroup[j].trajcolor], markersize=8, markevery=20)

    # 目盛りや軸ラベルのフォントサイズ調整。
    ax1.tick_params(labelsize=15)
    ax1.set_xlabel('Date Time', fontsize=18)
    ax1.set_ylabel('Altitude ($m$)', fontsize=18)

    # 画像ファイルの保存。日付の情報は流跡線から取得(元の流跡線開始日基準)。
    plt.savefig('img/plot_day/traj/eiken_{year}{month}{day}.png'.format(
        year=str(startday.year),
        month=str(startday.month).zfill(2),
        day=str(startday.day).zfill(2)
    ), dpi=600, bbox_inches='tight', pad_inches=0)
    plt.close() # 画像が多数あるため表示出力しない設定
```

コード4．流跡線と降雨情報を合わせて成分分析測定日毎にプロットするコード．

```
# 流跡線と降雨情報を成分分析測定日毎にプロットするコード。コード2 は実行済みで、コード3 の実行またはコード3 の変数の定義を行っていることが必要。
for i in range(trajgroup.trajcount // day_traj_num):
    fig = plt.figure(figsize=[12, 11]) # 図のサイズ指定
    spec = gridspec.GridSpec(ncols=1, nrows=2,hspace=0.01,height_ratios=[5, 1]) # 描画領域を2 つに分割
    ax0 = fig.add_subplot(spec[0]) # 地図を描画する領域として使用
    ax1 = fig.add_subplot(spec[1]) # 高度変化を描画する領域として使用
    bmap = bmap_params.make_basemap(ax=ax0) # basemap による地図の描画
    ax1.invert_xaxis() # x 軸（日時、Date Time）の反転
    # 開始日を取得
    startday = pd.to_datetime(trajgroup[i * day_traj_num].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')[0]

    # 12 本中初めの3 本の流跡線については高さ別の凡例を作成するため処理を分けている。
    for j in range(i * day_traj_num, i * day_traj_num + altitudes_num):
        datetime = pd.to_datetime(trajgroup[j].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
        altitude = trajgroup[j].data.geometry.z
        bmap.plot(*trajgroup[j].path.xy, c=trajgroup[j].trajcolor, latlon=True, zorder=20,
                  ax=ax0, ls=ls_dict[datetime[0].hour], marker=marker_dict[trajgroup[j].trajcolor],
                  markersize=8, markevery=20, label=str(int(altitude[0])) +' m')
        # 地図に降雨の情報をプロット。降雨がないときはプロットを表示しない設定。
        # また、vmax（colormap の最大値）の40 は対象期間の1 時間降雨最大値36.7mm から設定
        mappable = pysplit.traj_scatter(trajgroup.trajectories[j].data.Rainfall.values,
                                trajgroup.trajectories[j].data.geometry.x.values,
                                trajgroup.trajectories[j].data.geometry.y.values,
                                bmap, colormap=plt.cm.cool,
                                sizedata=np.array(list(map(lambda x : 5 if (x > 0) else 0,
                                                           trajgroup.trajectories[j].data.Rainfall.values))),
                                vmin=0, vmax=40, levels=2, suppress_printmsg=True)
        ax1.plot(datetime, altitude, color=trajgroup[j].trajcolor, ls=ls_dict[datetime[0].hour],
                 marker=marker_dict[trajgroup[j].trajcolor], markersize=8, markevery=20)
        # 日時に降雨情報をプロット
        ax1.scatter(datetime, altitude, c=trajgroup.trajectories[j].data.Rainfall.values, cmap=plt.cm.cool,
                    s=np.array(list(map(lambda x : 100 if (x > 0) else 0,
                                        trajgroup.trajectories[j].data.Rainfall.values))), vmin=0, vmax=40)

    # 凡例に線の種類による時刻情報を追加。
    handles, _ = ax0.get_legend_handles_labels() # 凡例情報を取得
    lines = [Line2D([0], [0], color='black', linewidth=2, linestyle=l,
                    label=str(pd.to_datetime(
                        trajgroup[i * day_traj_num + altitudes_num * k].data['DateTime'],utc=True).dt.tz_convert(
                        'Asia/Tokyo')[0].strftime('%Y-%m-%d %H:%M'))) for k,l in zip(range(hours_num), ls_dict.values())]
    handles.extend(lines) # 凡例を追加
    legend = ax0.legend(handles=handles, loc='best', fontsize=18) # 凡例の場所とフォントサイズを調整
    legend.set_zorder(level=25) # 凡例が隠れないようにzorder を調整

    # 残りの流跡線については凡例作成処理を除きプロットする。
    for j in range(i * day_traj_num + altitudes_num, i * day_traj_num + day_traj_num):
        datetime = pd.to_datetime(trajgroup[j].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
        altitude = trajgroup[j].data.geometry.z
        bmap.plot(*trajgroup[j].path.xy, c=trajgroup[j].trajcolor, latlon=True, zorder=20, ax=ax0,
                  ls=ls_dict[datetime[0].hour], marker=marker_dict[trajgroup[j].trajcolor], markersize=8, markevery=20)
        mappable = pysplit.traj_scatter(trajgroup.trajectories[j].data.Rainfall.values,
                                trajgroup.trajectories[j].data.geometry.x.values,
                                trajgroup.trajectories[j].data.geometry.y.values,
                                bmap, colormap=plt.cm.cool,
                                sizedata=np.array(list(map(lambda x : 5 if (x > 0) else 0,
                                                           trajgroup.trajectories[j].data.Rainfall.values))),
                                vmin=0, vmax=40, levels=2, suppress_printmsg=True)
        ax1.plot(datetime, altitude, color=trajgroup[j].trajcolor, ls=ls_dict[datetime[0].hour],
                 marker=marker_dict[trajgroup[j].trajcolor], markersize=8, markevery=20)
        ax1.scatter(datetime, altitude, c=trajgroup.trajectories[j].data.Rainfall.values, cmap=plt.cm.cool,
                    s=np.array(list(map(lambda x : 100 if (x > 0) else 0,
                                        trajgroup.trajectories[j].data.Rainfall.values))), vmin=0, vmax=40)

    # 目盛りや軸ラベルのフォントサイズ調整
    ax1.tick_params(labelsize=15)
    ax1.set_xlabel('Date Time', fontsize=18)
    ax1.set_ylabel('Altitude ($m$)', fontsize=18)
    # カラーバー配置設定
    axins = inset_axes(ax0, width="5%", height="100%", loc='lower left',
                       bbox_to_anchor=(1.01, -0.01, 1, 1), bbox_transform=ax0.transAxes)
    cb = fig.colorbar(mappable, cax=axins) # カラーバー追加
    cb.set_label('Rainfall', fontsize=15) # カラーバーラベル追加
    cb.ax.tick_params(labelsize=15) # カラーバー目盛りサイズ調整

    # 画像ファイルの保存。日付の情報は流跡線から取得（開始日基準）。
    plt.savefig('img/plot_day/rainy/eiken_rainy_{year}{month}{day}.png'.format(
        year=str(startday.year), month=str(startday.month).zfill(2), day=str(startday.day).zfill(2)
    ), dpi=600, bbox_inches='tight', pad_inches=0)
    plt.close() # 画像が多数あるため表示出力しない設定
```

コード5. 流跡線の誤差情報の計算とプロットの準備を行うコード.

```
# 流跡線の誤差情報の計算とプロットの準備を行うコード
# 使用するライブラリをインポート
import pysplit
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import gridspec
from matplotlib.lines import Line2D
import re

trajgroup = pysplit.make_trajectorygroup(r'D:¥trajectories¥eiken_FY2017-FY2020¥eiken_*') # 流跡線ファイルの読み込み
# 逆解析流跡線の読みこみ
for traj in trajgroup:
    traj.load_reversetraj()
# 積分誤差計算
for traj in trajgroup:
    traj.calculate_integrationerr()

error_id_list = [] # 相対誤差の結果がエラーとなる流跡線のリスト
relative_errors = [] # 流跡線の相対誤差を記録するリストを作成
# 相対誤差を計算。相対誤差の計算結果がエラーとなる場合、error の流跡線としループを続ける。
for traj in trajgroup:
    try:
        relative_errors.append(traj.integration_error)
    except:
        error_id_list.append(traj.trajid)
        continue

# カットオフ値の設定
cutoff = np.mean(relative_errors) + (np.std(relative_errors) * 2) # この例では 8.720559346578291
# error の流跡線を元の「TrajectoryGroup」から取り除く。この例では error の流跡線は 8 個。
error_trajgroup = trajgroup.pop(trajid=error_id_list)

bad = [] # カットオフ値を上回る相対誤差の流跡線のリストを作成
# カットオフ値と相対誤差を比較し bad の流跡線を分類する
for traj in trajgroup:
    if traj.integration_error > cutoff:
        bad.append(traj.trajid)

# bad の流跡線を元の「TrajectoryGroup」から取り除く。この例では bad の流跡線は 120 個。
bad_trajgroup = trajgroup.pop(trajid=bad)

# error と bad の流跡線をソート
error_trajgroup.trajectories.sort(key=lambda x: (x.data.DateTime[0],
                                                 (int(re.search(r'(500|1[0-5]00)', x.trajid).group()))))
bad_trajgroup.trajectories.sort(key=lambda x: (x.data.DateTime[0],
                                               (int(re.search(r'(500|1[0-5]00)', x.trajid).group()))))
# カットオフ値以下の流跡線から逆解析流跡線を作成
traj_id_list_r = [traj.rfullpath for traj in trajgroup]
reversetrajgroup = pysplit.make_trajectorygroup(traj_id_list_r)
reversetrajgroup.trajectories.sort(key=lambda x: (re.search(r'¥d{10}', x.trajid).group(),
                                                  (int(re.search(r'(500|1[0-5]00)', x.trajid).group()))))
# error の流跡線から逆解析流跡線を作成
error_id_list_r = [traj.rfullpath for traj in error_trajgroup]
error_trajgroup_r = pysplit.make_trajectorygroup(error_id_list_r)
error_trajgroup_r.trajectories.sort(key=lambda x: (re.search(r'¥d{10}', x.trajid).group(),
                                                   (int(re.search(r'(500|1[0-5]00)', x.trajid).group()))))
# bad の流跡線から逆解析流跡線を作成
bad_r = [traj.rfullpath for traj in bad_trajgroup]
bad_trajgroup_r = pysplit.make_trajectorygroup(bad_r)
bad_trajgroup_r.trajectories.sort(key=lambda x: (re.search(r'¥d{10}', x.trajid).group(),
                                                 (int(re.search(r'(500|1[0-5]00)', x.trajid).group()))))

# 地図描画領域や投影方法、地図解像度、描画する最小面積、緯度経度のフォントサイズを指定
mapcorners =  [90, 0, 160, 50] # 値は左下の経度、左下の緯度、右上の経度、右上の緯度を表す
standard_pm = None # 地図の投影方法が cyl（円筒形、メルカトル図法）では必要ないがそれ以外の場合に設定が必要
param_dict = {'projection':'cyl','resolution':'l', 'area_threshold':500, 'latlon_fs':18}

# 地図のパラメータを設定
bmap_params = pysplit.MapDesign(mapcorners, standard_pm, **param_dict)

# 色、線、マーカーの種類を設定
color_dict = {'traj':{500.0 : 'blue', 1000.0 : 'gold', 1500.0 : 'black'},
              'reversetraj':{500.0 : 'red', 1000.0 : 'purple', 1500.0 : 'cyan'}}

ls_dict = {12:'-', 18:'--', 0:'-.', 6:':'}

marker_dict = {'blue' : '^', 'gold' : 's', 'black' : 'o', 'red' : 'v', 'purple' : 'x', 'cyan' : '*'}
```

コード6. 流跡線と逆解析流跡線をプロットし相対誤差を表示するコード.

```
# 流跡線と逆解析流跡線をプロットし相対誤差を表示するコード。コード5 は実行済みであることが必要。
# error や bad の流跡線も流跡線の変数名や保存するファイル名を差し替えれば同様にプロットできるが、
# error の流跡線は相対誤差が計算できず存在しないので
#「流跡線の相対誤差を表示」の箇所は実行しないようにする必要がある

# 流跡線ファイルから高さの情報を取得しそれに応じて線の色を変更
for i in range(trajgroup.trajcount):
    altitude0 = trajgroup[i].data.geometry.z[0]
    trajgroup[i].trajcolor = color_dict['traj'][altitude0]
    reversetrajgroup[i].trajcolor = color_dict['reversetraj'][altitude0]

for i in range(trajgroup.trajcount):
    fig = plt.figure(figsize=[12, 11]) # 図のサイズ指定
    spec = gridspec.GridSpec(ncols=1, nrows=2,hspace=0.1,height_ratios=[5, 1]) # 描画領域を 2 つに分割
    ax0 = fig.add_subplot(spec[0]) # 地図を描画する領域として使用
    ax1 = fig.add_subplot(spec[1]) # 高度変化を描画する領域として使用
    bmap = bmap_params.make_basemap(ax=ax0) # basemap による地図の描画
    ax1.invert_xaxis() # x 軸（日時、Date Time）の反転
    # 開始時間を取得
    starttime = pd.to_datetime(trajgroup[i].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')[0]

    # 流跡線の日時と高さ情報の取得
    datetime = pd.to_datetime(trajgroup[i].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
    datetime_r = pd.to_datetime(reversetrajgroup[i].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
    altitude = trajgroup[i].data.geometry.z
    altitude_r = reversetrajgroup[i].data.geometry.z
    # 地図に流跡線をプロット
    bmap.plot(*trajgroup[i].path.xy, c=trajgroup[i].trajcolor, latlon=True, zorder=20, ax=ax0,
              ls=ls_dict[datetime[0].hour], marker=marker_dict[trajgroup[i].trajcolor],
              markersize=8, markevery=20, label=str(int(altitude[0])) +' m')
    bmap.plot(*reversetrajgroup[i].path.xy, c=reversetrajgroup[i].trajcolor, latlon=True, zorder=20, ax=ax0,
              ls=ls_dict[datetime[0].hour], marker=marker_dict[reversetrajgroup[i].trajcolor],
              markersize=8, markevery=20, label=str(int(altitude[0])) +' m (reverse)')
    # 流跡線の日時と高さ情報をプロット
    ax1.plot(datetime, altitude, color=trajgroup[i].trajcolor, ls=ls_dict[datetime[0].hour],
             marker=marker_dict[trajgroup[i].trajcolor], markersize=8, markevery=20)
    ax1.plot(datetime_r, altitude_r, color=reversetrajgroup[i].trajcolor, ls=ls_dict[datetime[0].hour],
             marker=marker_dict[reversetrajgroup[i].trajcolor], markersize=8, markevery=20)

    # 凡例に線の種類による時刻情報を追加。
    handles, _ = ax0.get_legend_handles_labels() # 凡例情報を取得
    line = Line2D([0], [0], color='black', linewidth=2, linestyle=ls_dict[datetime[0].hour],
                  label=str(starttime.strftime('%Y-%m-%d %H:%M')))
    handles.append(line) # 凡例を追加
    legend = ax0.legend(handles=handles, loc='lower right', fontsize=18) # 凡例の場所とフォントサイズを調整
    legend.set_zorder(level=25) # 凡例が隠れないように zorder を調整
    # 流跡線の相対誤差を表示。
    ax0.text(0.65, -0.05,'Integration error = ' + str(round(trajgroup.trajectories[i].integration_error, 3)),
             transform=ax0.transAxes, fontsize=18).set_zorder(level=25)

    # 目盛りや軸ラベルのフォントサイズ調整。
    ax1.tick_params(labelsize=15)
    ax1.set_xlabel('Date Time', fontsize=18)
    ax1.set_ylabel('Altitude ($m$)', fontsize=18)

    # 画像ファイルの保存。日付の情報は流跡線から取得（元の流跡線開始時間基準）。
    plt.savefig('img/plot_hour/good/eiken_reverse_{year}{month}{day}{hour}_{z}m.png'.format(
        year=str(starttime.year),
        month=str(starttime.month).zfill(2),
        day=str(starttime.day).zfill(2),
        hour=str(starttime.hour).zfill(2),
        z=str(int(trajgroup[i].data.geometry.z[0])),
    ),dpi=600, bbox_inches='tight', pad_inches=0)
    plt.close()
```

コード 7. 空気塊の起源を判別するためのマスク領域を作成するコード.

```
# 流跡線の情報を記録したデータフレームを作成するために空気塊の起源を判定するためのマスク領域を作成するコード。
# 使用するライブラリをインポート
import pysplit
import numpy as np
import pandas as pd
import geopandas as gpd
from shapely.geometry import Point
import matplotlib.pyplot as plt
from matplotlib import gridspec
import re
import itertools as it

# GADM シェープファイルを読み込み GADM:(https://gadm.org/index.html)
gadm0 = gpd.GeoDataFrame.from_file(r"D:¥gadm36_levels_shp¥gadm36_0.shp") # レベル 0 の行政区画(国レベル)
gadm1 = gpd.GeoDataFrame.from_file(r"D:¥gadm36_levels_shp¥gadm36_1.shp") # レベル 1 の行政区画(日本では都道府県レベル)
gadm2 = gpd.GeoDataFrame.from_file(r"D:¥gadm36_levels_shp¥gadm36_2.shp") # レベル 2 の行政区画(日本では市町村と特別区、離島レベル)

# 空気塊の起源を判定するためのマスク領域を作成
# 東アジア(EA)のマスク領域を作成
# 元の GADM ファイルから東アジアの各国を抽出する
East_Asia = gadm0.query("NAME_0 == 'China' | NAME_0 == 'Mongolia' |¥
                         NAME_0 == 'North Korea' | NAME_0 == 'South Korea' | NAME_0 == 'Taiwan'").reset_index()
East_Asia = East_Asia['geometry'].unary_union # 各国の領域を結合し東アジアのマスク領域とする

# 日本(J)のマスク領域を作成。流跡線の解析地点を含む沖縄や太平洋(PO)と被る小笠原、
# 沖縄から近い鹿児島の離島を除く操作も行う。
Japan = gadm0.query("NAME_0 == 'Japan'").reset_index() # 元の GADM ファイルから日本を抽出する
Okinawa = gadm1.query("NAME_1 == 'Okinawa'").reset_index() # 元の GADM ファイルから沖縄を抽出する
Ogasawara = gadm2.query("NAME_2 == 'Unknown7'").reset_index() # 元の GADM ファイルから小笠原を抽出する
Kagoshima = gadm2.query("NAME_1 == 'Kagoshima'") # 元の GADM ファイルから鹿児島を抽出する
# 鹿児島から離島を抽出する
Kagoshima_remote_island = Kagoshima.query("NAME_2 == 'Mishima' | NAME_2 == 'Nishinoomote' | NAME_2 == 'Nakatane' |¥
                                           NAME_2 == 'Minamitane' | NAME_2 == 'Yakushima' | NAME_2 == 'Toshima' |¥
                                           NAME_2 == 'Tatsugō' | NAME_2 == 'Yamato' | NAME_2 == 'Kikai' |¥
                                           NAME_2 == 'Amami' | NAME_2 == 'Uken' | NAME_2 == 'Setouchi' |¥
                                           NAME_2 == 'Amagi' | NAME_2 == 'Tokunoshima' | NAME_2 == 'Isen' |¥
                                           NAME_2 == 'Wadomari' | NAME_2 == 'China' | NAME_2 == 'Yoron'").reset_index()
Kagoshima_remote_island = Kagoshima_remote_island['geometry'].unary_union # 離島領域を結合
Japan = Japan.difference(Okinawa, align=False) # 日本と沖縄の差分
Japan = Japan.difference(Ogasawara, align=False) # 日本と小笠原の差分
Japan = Japan.difference(Ogasawara, align=False) # 1 回では望む結果が返らないため、同じ操作を 2 回続けて行っている
Japan = Japan.difference(Kagoshima_remote_island, align=False) # 日本と鹿児島離島の差分
Japan = Japan.difference(Kagoshima_remote_island, align=False)[0] # 1 回では望む結果が返らないため、同じ操作を 2 回続けて行っている

# 東南アジア(SA)のマスク領域を作成
# 元の GADM ファイルから東南アジアの各国を抽出する
Southeast_Asia = gadm0.query("NAME_0 == 'Indonesia' | NAME_0 == 'Myanmar' | NAME_0 == 'Thailand' |¥
                              NAME_0 == 'Vietnam' | NAME_0 == 'Malaysia' | NAME_0 == 'Philippines' |¥
                              NAME_0 == 'Laos' | NAME_0 == 'Cambodia' | NAME_0 == 'East Timor' |¥
                              NAME_0 == 'Brunei' | NAME_0 == 'Singapore'").reset_index()
Southeast_Asia = Southeast_Asia['geometry'].unary_union # 各国の領域を結合し東南アジアのマスク領域とする

# 太平洋(PO)のマスク領域を作成
point = Point(146, 16) # 太平洋上に点を指定
Pacific_ocean = point.buffer(18.0) # 点を中心とした円の領域を設定し太平洋のマスク領域とする

# この後の処理で for ループを行うためリストを作成
mask_list = [East_Asia, Japan, Southeast_Asia, Pacific_ocean]
maskname_list = ['EA', 'J', 'SA', 'PO']
```

コード 8. 流跡線の読み込みと誤差情報の計算を行うコード.

```
# 流跡線の情報を記録したデータフレームを作成するために流跡線の読み込みと誤差情報の計算を行うコード。コード 7 は実行済みであることが必要。
# 流跡線ファイルの読み込み
trajgroup = pysplit.make_trajectorygroup(r'D:¥trajectories¥eiken_FY2017-FY2020¥eiken_*')

# 雨の情報、逆解析流跡線の読みこみ、積分誤差計算
for traj in trajgroup:
    traj.set_rainstatus()
    traj.load_reversetraj()
# 積分誤差計算
for traj in trajgroup:
    traj.calculate_integrationerr()

# 相対誤差と絶対誤差の取得
error_id_list = []
relative_errors = []
abs_errors = []

for traj in trajgroup:
    try:
        relative_errors.append(traj.integration_error)
        abs_errors.append(traj.integration_error_abs)
    except:
        error_id_list.append(traj.trajid)
        relative_errors.append(np.nan)
        abs_errors.append(np.nan)
        continue

# カットオフ値の設定
cutoff = np.nanmean(relative_errors) + (np.nanstd(relative_errors) * 2)

# 相対誤差がカットオフ値を下回っているかの判定
relative_errors_check = []
for traj in trajgroup:
    try:
        if traj.integration_error > cutoff:
            relative_errors_check.append(False)
        else:
            relative_errors_check.append(True)
    except:
        relative_errors_check.append(False)
        continue

# 逆解析流跡線ファイルの読み込みとソート
reversetrajgroup = pysplit.make_trajectorygroup(r'D:¥trajectories¥eiken_FY2017-FY2020¥reversetraj¥eiken_*')
reversetrajgroup.trajectories.sort(key=lambda x: (re.search(r'¥d{10}', x.trajid).group(),
                                                  int(re.search(r'(500|1[0-5]00)', x.trajid).group())))
reversetrajgroup.trajids.sort(key=lambda x: (re.search(r'¥d{10}', x).group(),
                                             int(re.search(r'(500|1[0-5]00)', x).group())))
```

コード9．データフレームを作成する関数を定義するコード(1).

```
# 流跡線の情報を記録したデータフレームを作成するためにデータフレームを作成する関数を定義するコード(1)。コード7,8 が実行済みであることが必要。

def traj_make_df(trajgroup, mask_list, maskname_list):
    """
    「TrajectoryGroup」内の「Trajectory」オブジェクトの情報を取得しデータフレーム（DataFrame、表形式）でまとめる関数
    [入力]
    trajgroup:「TrajectoryGroup」オブジェクト。「Trajectory」オブジェクトの集合。pysplit.make_trajectorygroup()で作成できる。
    mask_list:マスク用の shapely の「Polygon」または「MultiPolygon」オブジェクトの入ったリスト。
        shapely の利用や GADM データを加工して作成する。
    maskname_list:マスク用ファイルの名前が文字列オブジェクトで入っているリスト。列の命名に利用。
    [出力]
    df:pandas の「DataFrame」オブジェクト。ファイル名や日付、出発高度などの流跡線の情報やマスク領域に含まれるかの判定、
    マスク領域に含まれる流跡線の地点数などが格納されている。
    """

    def calc_DateTime(trajgroup_trajectories):
        """
        「Trajectory」オブジェクトの日付を取り出す関数。日付の型は Pandas の Timestamp 型。
        先頭の流跡線の種類で処理を変えている。
        """
        if re.search(r'REVERSE', trajgroup_trajectories.trajid):
            return pd.to_datetime(re.search(r'¥d{10}', trajgroup_trajectories.trajid).group(), format='%Y%m%d%H', utc=True)
        else:
            return pd.to_datetime(trajgroup_trajectories.data['DateTime'][0], utc=True)

    def calc_Altitude(trajgroup_trajectories):
        """
        「Trajectory」オブジェクトの初期高さを取り出す関数。
        先頭の流跡線の種類で処理を変えている。
        """
        if re.search(r'REVERSE', trajgroup_trajectories.trajid):
            return int(re.search(r'(500|1[0-5]00)', trajgroup_trajectories.trajid).group())
        else:
            return trajgroup_trajectories.data.geometry.z[0]

    def calc_Altitude_check(trajgroup_trajectories):
        """
        「Trajectory」オブジェクトの高さでゼロ未満がないか判定する関数。
        全てがゼロよりも大きい正の値なら True、一つ以上ゼロ未満があるならば False が出力される。
        """
        return all(trajgroup_trajectories.data.geometry.z > 0)

    def rainy_sum(trajgroup_trajectories):
        """
        「Trajectory」オブジェクトの降雨合計値を計算する関数。
        """
        return trajgroup_trajectories.data.Rainfall.sum()

    def rainy_max(trajgroup_trajectories):
        """
        「Trajectory」オブジェクトの降雨最大値を取り出す関数。
        """
        return trajgroup_trajectories.data.Rainfall.max()

    def rainy_check(trajgroup_trajectories):
        """
        「Trajectory」オブジェクトの降雨を判定する関数。
        降雨があれば True、なければ False が出力される。
        """
        return trajgroup_trajectories.rainy

    def get_len_data(trajgroup_trajectories):
        """
        「Trajectory」オブジェクトの data の長さを取得する関数。
        """
        return len(trajgroup_trajectories.data)
```

コード 10. データフレームを作成する関数を定義するコード(2).

```
# 流跡線の情報を記録したデータフレームを作成するためにデータフレームを作成する関数を定義するコード(2)。コード 9 の続き。
    # この前に定義した関数内関数などを利用し、流跡線のファイル名や時間、出発高度、雨、流跡線の作成時間を取得
    # 先頭の流跡線の種類で作成する列と名前の処理を変えている。
    if re.search(r'REVERSE', trajgroup.trajectories[0].trajid):
        id_series = pd.Series(trajgroup.trajids)
        df = pd.DataFrame({'r_Data_Name': id_series})
        DateTime = pd.Series(map(calc_DateTime, trajgroup.trajectories), name='r_DateTime')
        df_time = pd.DataFrame({'r_Year': DateTime.dt.year, 'r_Month': DateTime.dt.month,
                           'r_Day': DateTime.dt.day, 'r_Hour': DateTime.dt.hour, 'r_Weekday_name': DateTime.dt.day_name()})
        DateTime_JST = DateTime.dt.tz_convert('Asia/Tokyo')
        DateTime_JST = DateTime_JST.rename('r_DateTime_JST')
        df_time_JST = pd.DataFrame({'r_Year_JST': DateTime_JST.dt.year, 'r_Month_JST': DateTime_JST.dt.month,
                        'r_Day_JST': DateTime_JST.dt.day, 'r_Hour_JST': DateTime_JST.dt.hour,
                                    'r_Weekday_name_JST': DateTime_JST.dt.day_name()})
        Altitude = pd.Series(map(calc_Altitude, trajgroup.trajectories), name='r_Altitude')
        Altitude_check = pd.Series(map(calc_Altitude_check, trajgroup.trajectories), name='r_Altitude_check')
        Length = pd.Series(map(get_len_data, trajgroup.trajectories), name='r_Length')
        df = pd.concat([df, DateTime, df_time, DateTime_JST, df_time_JST, Altitude, Altitude_check, Length], axis=1)

    else:
        id_series = pd.Series(trajgroup.trajids)
        df = pd.DataFrame({'Data_Name': id_series})
        DateTime = pd.Series(map(calc_DateTime, trajgroup.trajectories), name='DateTime')
        df_time = pd.DataFrame({'Year': DateTime.dt.year, 'Month': DateTime.dt.month,
                           'Day': DateTime.dt.day, 'Hour': DateTime.dt.hour, 'Weekday_name': DateTime.dt.day_name()})
        DateTime_JST = DateTime.dt.tz_convert('Asia/Tokyo')
        DateTime_JST = DateTime_JST.rename('DateTime_JST')
        df_time_JST = pd.DataFrame({'Year_JST': DateTime_JST.dt.year, 'Month_JST': DateTime_JST.dt.month,
                        'Day_JST': DateTime_JST.dt.day, 'Hour_JST': DateTime_JST.dt.hour,
                                    'Weekday_name_JST': DateTime_JST.dt.day_name()})
        Altitude = pd.Series(map(calc_Altitude, trajgroup.trajectories), name='Altitude')
        Altitude_check = pd.Series(map(calc_Altitude_check, trajgroup.trajectories), name='Altitude_check')
        Rainy_sum = pd.Series(map(rainy_sum, trajgroup.trajectories), name='Rainy_sum')
        Rainy_max = pd.Series(map(rainy_max, trajgroup.trajectories), name='Rainy_max')
        Rainy_check = pd.Series(map(rainy_check, trajgroup.trajectories), name='Rainy_check')
        Length = pd.Series(map(get_len_data, trajgroup.trajectories), name='Length')
        df = pd.concat([df, DateTime, df_time, DateTime_JST, df_time_JST, Altitude, Altitude_check,
                        Rainy_sum, Rainy_max, Rainy_check, Length], axis=1)

    # マスク領域に含まれる地点数のカウントと流跡線がマスク領域に含まれるかの判定結果を取得
    for i in range(len(mask_list)):
        check_count = [] # マスク領域内の流跡線地点カウント数を格納するリスト
        check_TF = [] # マスク領域内に流跡線が含まれるか判定結果を格納するリスト

        for t in trajgroup:
            clipped = gpd.clip(gdf=t.data, mask=mask_list[i]) # マスク領域で流跡線データを切り取る
            clipped_count = len(clipped) # マスク領域内の流跡線地点カウント数を計算
            TF = len(clipped) != 0 # マスク領域内に流跡線が含まれるか判定
            check_count.append(clipped_count)
            check_TF.append(TF)
        if re.search(r'REVERSE', t.trajid):
            checkcheck_count_series = pd.Series(check_count, name='r_' + maskname_list[i] + '_count')
            check_TF_series = pd.Series(check_TF, name='r_' + maskname_list[i])
        else:
            checkcheck_count_series = pd.Series(check_count, name=maskname_list[i] + '_count')
            check_TF_series = pd.Series(check_TF, name=maskname_list[i])

        df = pd.concat([df, checkcheck_count_series, check_TF_series], axis=1)

    return df
```

コード 11. データフレーム作成関数を用いて流跡線の情報を取得するコード.

```
# データフレーム作成関数を用いて流跡線の情報を取得するコード。コード 7-10 が実行済みであることが必要。
# 定義した関数を用いてデータフレームを作成
df = traj_make_df(trajgroup=trajgroup, mask_list=mask_list, maskname_list=maskname_list)

# ['Other_count']（マスク領域に含まれていない地点の数）の列を作成
series = pd.Series(df[['Length', 'EA_count', 'J_count','SA_count', 'PO_count']].apply(
    lambda x: x[0]-np.sum(x[1:]), axis=1), name='Other_count')
df = pd.concat([df, series], axis=1)

# ['FY']（年度）の列を[DateTime_JST']を元に作成
df.loc[df['DateTime_JST'].dt.month >= 4, 'FY'] = df['DateTime_JST'].dt.year
df.loc[df['DateTime_JST'].dt.month < 4, 'FY'] = df['DateTime_JST'].dt.year - 1
# ['FY']列を float 型から int 型に変換
df['FY'] = df['FY'].astype(int)

# ['Season']（季節）の列を作成
season_check = []
s_list = ['spring', 'summer', 'autumn', 'winter']
for d in df['Data_Name']:
    for s in s_list:
        if s in d:
            season_check.append(s)
series = pd.Series(season_check, name='Season')
df = pd.concat([df, series], axis=1)

# 流跡線の誤差関係の列を作成
series = pd.Series(relative_errors, name='Integration_error')
df = pd.concat([df, series], axis=1)
series = pd.Series(abs_errors, name='Integration_error_abs')
df = pd.concat([df, series], axis=1)
series = pd.Series(relative_errors_check, name='Relative_errors_check')
df = pd.concat([df, series], axis=1)

# マスク領域の判定を元に細かい分類を作成
vec = np.arange(4) # [0, 1, 2, 3]の np.array
# 1 つのマスク領域のみに属する流跡線
for i in range(len(maskname_list)):
    ind = np.ones(4, dtype=bool) # 全て True の np.array
    ind[i] = False # 取り出したい領域のインデックスのみ False に変更
    # ind を一か所 False にすることでその領域以外が False となる条件と取り出したい領域が True の条件で場合分けができる
    df['{}_only'.format(maskname_list[i])] = ((df[maskname_list[vec[ind][0]]] == df[maskname_list[vec[ind][1]]]) &
                                              (df[maskname_list[vec[ind][1]]] == df[maskname_list[vec[ind][2]]]) &
                                              (df[maskname_list[vec[ind][0]]] == False) &
                                              (df[maskname_list[i]] == True))

cmt = list(it.combinations(maskname_list,2)) # 2 つのマスク領域の組み合わせを網羅したリスト
# 2 つのマスク領域にまたがる場合の流跡線
for i in range(len(cmt)):
    remove_list = [cmt[i][0], cmt[i][1]]
    other = [j for j in maskname_list if j not in remove_list]
    # other に False にしたい領域、cmt に True にしたい領域を指定することで場合分けができる
    df['{}_and_{}'.format(cmt[i][0], cmt[i][1])] = ((df[other[0]] == df[other[1]]) & (df[other[0]] == False) &
                                                    (df[cmt[i][0]] == df[cmt[i][1]]) & (df[cmt[i][0]] == True))

cmt = list(it.combinations(maskname_list,3)) # 3 つのマスク領域の組み合わせを網羅したリスト
# 3 つのマスク領域にまたがる場合の流跡線
for i in range(len(cmt)):
    remove_list = [cmt[i][0], cmt[i][1], cmt[i][2]]
    other = [j for j in maskname_list if j not in remove_list]
    # other に False にしたい領域、cmt に True にしたい領域を指定することで場合分けができる
    df['{}_and_{}_and_{}'.format(cmt[i][0], cmt[i][1], cmt[i][2])] = ¥
    ((df[other[0]] == False) & (df[cmt[i][0]] == True) &
     (df[cmt[i][0]] == df[cmt[i][1]]) & (df[cmt[i][1]] == df[cmt[i][2]]))

cmt = list(it.combinations(maskname_list,4)) # 全てのマスク領域のリスト
# 全てのマスク領域に含まれる場合の流跡線
for i in range(len(cmt)):
    df['All'] = ((df[cmt[i][0]] == df[cmt[i][1]]) & (df[cmt[i][1]] == df[cmt[i][2]]) &
                 (df[cmt[i][2]] == df[cmt[i][3]]) &(df[cmt[i][0]] == True))

# 全てのマスク領域のどこにも含まれない場合の流跡線
for i in range(len(cmt)):
    df['Other'] = ((df[cmt[i][0]] == df[cmt[i][1]]) & (df[cmt[i][1]] == df[cmt[i][2]]) &
                 (df[cmt[i][2]] == df[cmt[i][3]]) &(df[cmt[i][0]] == False))
```

コード 12. データフレーム作成関数を用いて逆解析流跡線の情報を取得とデータフレームを結合するコード.

```
# データフレーム作成関数を用いて逆解析流跡線の情報を取得とデータフレームを結合するコード。コード 7-11 が実行済みであることが必要。
# 定義した関数を用いてデータフレームを作成
r_df = traj_make_df(trajgroup=reversetrajgroup, mask_list=mask_list, maskname_list=maskname_list)

# ['Other_count']（マスク領域に含まれていない地点の数）の列を作成
series = pd.Series(r_df[['r_Length', 'r_EA_count', 'r_J_count', 'r_SA_count', 'r_PO_count']].apply(
    lambda x: x[0] - np.sum(x[1:]), axis=1), name='r_Other_count')
r_df = pd.concat([r_df, series], axis=1)

# マスク領域の判定を元に細かい分類を作成
vec = np.arange(4) # [0, 1, 2, 3]の np.array
r_maskname_list = ['r_EA', 'r_J', 'r_SA', 'r_PO']

# 1 つのマスク領域のみに属する流跡線
for i in range(len(r_maskname_list)):
    ind = np.ones(4, dtype=bool) # 全て True の np.array
    ind[i] = False # 取り出したい領域のインデックスのみ False に変更
    # ind を一か所 False にすることでその領域以外が False となる条件と取り出したい領域が True の条件で場合分けができる
    r_df['{}_only'.format(r_maskname_list[i])] = ¥
    ((r_df[r_maskname_list[vec[ind][0]]] == r_df[r_maskname_list[vec[ind][1]]]) &
     (r_df[r_maskname_list[vec[ind][1]]] == r_df[r_maskname_list[vec[ind][2]]]) &
     (r_df[r_maskname_list[vec[ind][0]]] == False) &
     (r_df[r_maskname_list[i]] == True))

cmt = list(it.combinations(r_maskname_list,2)) # 2 つのマスク領域の組み合わせを網羅したリスト
# 2 つのマスク領域にまたがる場合の流跡線
for i in range(len(cmt)):
    remove_list = [cmt[i][0], cmt[i][1]]
    other = [j for j in r_maskname_list if j not in remove_list]
    # other に False にしたい領域、cmt に True にしたい領域を指定することで場合分けができる
    r_df['{}_and_{}'.format(cmt[i][0], cmt[i][1])] = ((r_df[other[0]] == r_df[other[1]]) & (r_df[other[0]] == False) &
                                                      (r_df[cmt[i][0]] == r_df[cmt[i][1]]) & (r_df[cmt[i][0]] == True))

cmt = list(it.combinations(r_maskname_list,3)) # 3 つのマスク領域の組み合わせを網羅したリスト
# 3 つのマスク領域にまたがる場合の流跡線
for i in range(len(cmt)):
    remove_list = [cmt[i][0], cmt[i][1], cmt[i][2]]
    other = [j for j in r_maskname_list if j not in remove_list]
    # other に False にしたい領域、cmt に True にしたい領域を指定することで場合分けができる
    r_df['{}_and_{}_and_{}'.format(cmt[i][0], cmt[i][1], cmt[i][2])] = ¥
    ((r_df[other[0]] == False) & (r_df[cmt[i][0]] == True) &
     (r_df[cmt[i][0]] == r_df[cmt[i][1]]) & (r_df[cmt[i][1]] == r_df[cmt[i][2]]))

cmt = list(it.combinations(r_maskname_list,4)) # 全てのマスク領域のリスト
# 全てのマスク領域に含まれる場合の流跡線
for i in range(len(cmt)):
    r_df['r_All'] = ((r_df[cmt[i][0]] == r_df[cmt[i][1]]) & (r_df[cmt[i][1]] == r_df[cmt[i][2]]) &
                (r_df[cmt[i][2]] == r_df[cmt[i][3]]) & (r_df[cmt[i][0]] == True))
# 全てのマスク領域のどこにも含まれない場合の流跡線
for i in range(len(cmt)):
    r_df['r_Other'] = ((r_df[cmt[i][0]] == r_df[cmt[i][1]]) & (r_df[cmt[i][1]] == r_df[cmt[i][2]]) &
                (r_df[cmt[i][2]] == r_df[cmt[i][3]]) & (r_df[cmt[i][0]] == False))

# 2 つのデータフレームの結合。不要な列は指定せず省いている。
df = pd.concat([df, r_df.iloc[:,np.r_[0,14:41]]], axis=1)

# df.columns # 列名を確認

# 列の並び替えと不要な列は指定せず省いている。
df = df.reindex(columns=['Data_Name', 'r_Data_Name', 'DateTime', 'Year', 'Month', 'Day',
               'Hour', 'Weekday_name', 'Season','FY', 'DateTime_JST', 'Year_JST', 'Month_JST',
               'Day_JST', 'Hour_JST','Weekday_name_JST', 'Altitude', 'Altitude_check',
               'r_Altitude_check', 'Rainy_sum', 'Rainy_check', 'Rainy_max', 'Length', 'r_Length',
               'Integration_error', 'Integration_error_abs', 'Relative_errors_check',
               'EA_count', 'J_count','SA_count', 'PO_count', 'Other_count',
               'r_EA_count', 'r_J_count', 'r_SA_count', 'r_PO_count', 'r_Other_count',
               'EA_only', 'J_only', 'SA_only', 'PO_only', 'EA_and_J', 'EA_and_SA', 'EA_and_PO',
               'J_and_SA', 'J_and_PO', 'SA_and_PO', 'EA_and_J_and_SA',
               'EA_and_J_and_PO', 'EA_and_SA_and_PO', 'J_and_SA_and_PO', 'All',
               'Other', 'r_EA_only', 'r_J_only', 'r_SA_only', 'r_PO_only', 'r_EA_and_r_J',
               'r_EA_and_r_SA', 'r_EA_and_r_PO', 'r_J_and_r_SA', 'r_J_and_r_PO', 'r_SA_and_r_PO',
               'r_EA_and_r_J_and_r_SA', 'r_EA_and_r_J_and_r_PO',
               'r_EA_and_r_SA_and_r_PO', 'r_J_and_r_SA_and_r_PO', 'r_All', 'r_Other'])

df.to_csv('csv/eiken_traj_info.csv', index=False) # csv で保存
```

コード 13. 地域区分の割合を図示した棒グラフをプロットするコード（1）.

```
# 作成したデータフレームからマスク領域のカウント数を抜き出し地域区分の割合を図示した棒グラフをプロットするコード。
# コード7-12 によりcsv ファイルが作成済みであることが必要。
# ['Altitude_check']、['r_Altitude_check']または['Relative_errors_check']がFalse のデータと
# ['Length']、['r_Length']が121 でないデータを削除(5 日、120h)。

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import rc
from matplotlib.patches import Patch
import seaborn as sns

df = pd.read_csv('csv/eiken_traj_info.csv') # csv ファイルの読み込み

# 変数の定義
day_traj_num = 12 # 1 日あたりの流跡線の数

# ['Altitude_check']、['r_Altitude_check']または['Relative_errors_check1']がFalse のデータと
# ['Length']、['r_Length']が121h でないデータを削除
check = df.where((df['Altitude_check'] == True) & (df['r_Altitude_check'] == True) &
                 (df['Relative_errors_check'] == True) & (df['Length'] == 121) & (df['r_Length'] == 121))
# データフレームからカウント値を抜き出す。
df_count = check[['EA_count', 'J_count','SA_count', 'PO_count']].groupby(check.index // day_traj_num).sum()
# 列名を修正
df_count = df_count.rename(columns={'EA_count': 'EA', 'J_count':'J', 'SA_count':'SA', 'PO_count':'PO'})

# マスク領域でカウント数が最も多い地域を取得
Region_list = []
for i in range(len(df_count) // day_traj_num):
    Region_list.append(list(df_count.iloc[i * day_traj_num: i * day_traj_num + day_traj_num].sum()[
        df_count.iloc[i * day_traj_num: i * day_traj_num + day_traj_num].sum() ==
        df_count.iloc[i * day_traj_num: i * day_traj_num + day_traj_num].sum().max()
    ].index))
Region_list

# 取得した地域名を列として追加
series = pd.Series(df_count.idxmax(axis=1),name='region')
df_count = pd.concat([df_count, series], axis=1)

# データフレームから必要な日時を抜き出す.高さと時間は設定は任意の一種ををを指定すればよい。
df_time = df.query('Altitude == 500 & Hour_JST == 12')[['FY','Year_JST','Month_JST',
                                                        'Day_JST','Season']].reset_index(drop=True)
# カウント値と日時のデータフレームを結合
df = pd.concat([df_count, df_time], axis=1)

# 年度と季節でグループ化し、必要な列を抽出
df2 = df.groupby(['FY','Season'])[['EA', 'J', 'SA', 'PO']].sum()
# グループ化したインデックスをリセットし、行を並び替え
df2 = df2.reset_index().reindex([1, 2, 0, 3, 5, 6, 4, 7,9, 10, 8, 11, 13,
                                 14, 12, 15], axis=0).reset_index(drop=True)

# plot の準備
n = [0,1,2,3,4]

r_list = ['EA', 'J', 'SA', 'PO']
s_list = ['spring', 'summer', 'autumn', 'winter']

# 変数を作成(領域と季節毎に整理)
for s in s_list:
    for r in r_list:
        # 指定の(領域、季節)の年度毎合計値
        exec("{0}{1}_sum = df2[df2['Season'] == '{1}']['{0}'].reset_index(drop=True)".format(r,s))
        exec("{0}{1}_sum[4] = {0}{1}_sum.sum()".format(r,s)) # 指定の(領域、季節)の全期間の合計を計算
        exec("{0}{1}_sum = {0}{1}_sum.reindex([4, 0, 1, 2, 3]).reset_index(drop=True)".format(r,s)) # 値の並び替え

# 変数を作成(領域の季節合計を足して領域の通年の全期間の合計と年度毎合計値を計算)
for r in r_list:
    exec("{0}_sum = {0}spring_sum + {0}summer_sum + {0}autumn_sum + {0}winter_sum".format(r))

# 変数を作成(各季節の合計値を計算し、地域区分の割合を計算)
for s in s_list:
    exec("total_{0} = [i+j+k+l for i,j,k,l in zip(EA{0}_sum, J{0}_sum, SA{0}_sum, PO{0}_sum)]".format(s))
    exec("total_{0} = [1 if i == 0 else i for i in total_{0}]".format(s)) # 割合を計算するので0 の場合は1 にしておく
    for r in r_list[:4]:
        exec("{0}{1}_Bars = [i / j * 100 for i,j in zip({0}{1}_sum, total_{1})]".format(r,s)) # 割合を計算

total = [i+j+k+l for i,j,k,l in zip(EA_sum, J_sum, SA_sum, PO_sum)] # 全期間の合計
total = [1 if i == 0 else i for i in total] # 割合を計算するので0 の場合は1 にしておく
```

コード 14. 地域区分の割合を図示した棒グラフをプロットするコード（2）.

```
# 地域区分の割合を図示した棒グラフをプロットするコード(2)。コード14 の続き。

# 割合を計算
EA_Bars = [i / j * 100 for i,j in zip(EA_sum, total)]
J_Bars = [i / j * 100 for i,j in zip(J_sum, total)]
SA_Bars = [i / j * 100 for i,j in zip(SA_sum, total)]
PO_Bars = [i / j * 100 for i,j in zip(PO_sum, total)]

# 図のサイズ指定と描画領域を5 つに分割、y 軸を共有
fig, ax = plt.subplots(ncols=5, figsize=(12, 8), sharey="all")

barWidth = 0.5
names = ('Total', 'FY2017', 'FY2018', 'FY2019', 'FY2020')

# 全期間の合計部分(All)のプロット
# Create EA Bars
ax[0].bar(n, EA_Bars, color='#b5ffb9',
        edgecolor='black', width=barWidth, label='EA', hatch= '//')
# Create J Bars
ax[0].bar(n, J_Bars, bottom=EA_Bars, color='#f9bc86',
        edgecolor='black', width=barWidth, label='J', hatch=r'¥¥')
# Create SA Bars
ax[0].bar(n, SA_Bars, bottom=[i+j for i,j in zip(EA_Bars, J_Bars)], color='#f1cbff',
        edgecolor='black', width=barWidth, label='SA', hatch='++')
# Create PO Bars
ax[0].bar(n, PO_Bars, bottom=[i+j+k for i,j,k in zip(EA_Bars, J_Bars, SA_Bars)], color='#a3acff',
        edgecolor='black', width=barWidth, label='PO', hatch='..')
# 目盛りや軸ラベルのフォントサイズ等調整。
ax[0].set_xticks(n, names)
ax[0].set_xticklabels(names, rotation=90, ha='center', fontsize= 20)
ax[0].tick_params(axis='y', labelsize=20)
ax[0].set_ylabel("%", fontsize= 20)
ax[0].set_title("All", fontsize= 20)

# 各季節集計部分のプロット
for z,s in enumerate(s_list, 1):
    # Create EA Bars
    exec("ax[z].bar(n, EA{}_Bars, color='#b5ffb9',¥
        edgecolor='black', width=barWidth, label='EA', hatch= '//')".format(s))
    # Create J Bars
    exec("ax[z].bar(n, J{0}_Bars, bottom=EA{0}_Bars, color='#f9bc86',¥
        edgecolor='black', width=barWidth, label='J', hatch=r'¥¥¥¥')".format(s))
    # Create SA Bars
    exec("ax[z].bar(n, SA{0}_Bars, bottom=[i+j for i,j in zip(EA{0}_Bars, J{0}_Bars)],¥
        color='#f1cbff', edgecolor='black', width=barWidth, label='SA', hatch='++')".format(s))
    # Create PO Bars
    exec("ax[z].bar(n, PO{0}_Bars, bottom=[i+j+k for i,j,k in zip(EA{0}_Bars, J{0}_Bars, SA{0}_Bars)],¥
        color='#a3acff', edgecolor='black', width=barWidth, label='PO', hatch='..')".format(s))
    # 目盛りや軸ラベルのフォントサイズ等調整。
    ax[z].set_xticks(n, names)
    ax[z].set_xticklabels(names, rotation=90, ha='center', fontsize= 20)
    exec("ax[z].set_title('{}', fontsize= 20)".format(s.capitalize()))

plt.legend(loc=(1.03,0.6), fontsize=24) # 凡例を追加
plt.tight_layout() # 図のはみだしがないようにレイアウト調整
plt.savefig('img/barplot.png', dpi=600) # 画像ファイルの保存。
plt.show()
```

コード 15. 測定日ごとに空気塊の起源を判定した結果をヒートマップ形式でプロットするコード.

```
# 測定日ごとに空気塊の起源を判定した結果をヒートマップ形式でプロットするコード(2)。コード13-14 から続けて実行する必要がある。

value_to_int = {j:i for i,j in enumerate(r_list)} # ヒートマップ用に文字列を数字に変換
# ヒートマップ用のデータフレームを作成。16 は対象期間の四半期の数、14 は四半期毎の測定日の数である。
df3 = pd.DataFrame(df.region.values.reshape(16, 14)).replace(value_to_int)

# ヒートマップ用のハッチや色を準備
hatch_patterns = ['//', r'¥¥', '++', '..']
color_taple = ('#b5ffb9', '#f9bc86', '#f1cbff', '#a3acff')
# ヒートマップ内に表示する文字を準備。数字の 224 は全測定日の数、16 は対象期間の四半期の数、14 は四半期毎の測定日の数である。
annot = np.array(['{}/{}'.format(df.Month_JST[i], df.Day_JST[i]) for i in range(224)]).reshape(16, 14)

# 要素の数とハッチ追加時の pcolor()メソッドで使用する x,y 変数を定義
n = len(value_to_int)
x = np.arange(df3.shape[1] + 1) # ヒートマップ用のデータフレームより要素数が1大きい必要がある
y = np.arange(df3.shape[0] + 1) # ヒートマップ用のデータフレームより要素数が1大きい必要がある

cmap = sns.color_palette(color_taple, n) # ヒートマップ用のカラーパレットを作成

# プロット領域の作成
fig = plt.figure(figsize=(12,8))
ax = fig.add_subplot(111)

# ヒートマップの作成
sns.heatmap(df3, cmap=cmap, linewidths=1, linecolor='black', annot=annot,
            annot_kws={'color': 'black', 'size': 10, 'backgroundcolor':'w'}, fmt='', ax=ax, cbar=False)

# ハッチの追加
for i in range(4):
    ax.pcolor(x, y, np.where((df3 == i), df3, np.nan), hatch=hatch_patterns[i], alpha=0)

# y 軸の目盛りラベルを作成
y_label = ['{}FY({})'.format(df.FY[i*14], df.Season[i*14]) for i in range(16)]

# 目盛りや軸ラベルの設定
ax.set_xticklabels(list((range(1, 15))), rotation=0, ha='center')
ax.set_yticklabels(y_label, rotation=0, ha='right')

# 凡例の作成
legend_elements = [Patch(edgecolor='black', facecolor=color_taple[i], hatch=hatch_patterns[i],
                         lw=1, label=r_list[i]) for i in range(4)]
ax.legend(handles=legend_elements, bbox_to_anchor=(1.01, 1.02), loc='upper left', fontsize=18) # 凡例の追加

plt.tight_layout() # 図のはみだしがないようにレイアウト調整
plt.savefig('img/heatmap.png', dpi=600) # 画像ファイルの保存。
plt.show()
```

コード 16. 逆解析流跡線を成分分析測定日毎にプロットするコード.

```
# 逆解析流跡線を成分分析測定日毎にプロットするコード。コード 2 が実行済みであることが必要。
# reverstrajgroup は日時順、高さでソート済み。初めの 7 つの並びは以下参照。
# "eiken_may0500spring2017051003REVERSE", "eiken_may1000spring2017051003REVERSE",
# "eiken_may1500spring2017051003REVERSE", "eiken_may0500spring2017051009REVERSE",
# "eiken_may1000spring2017051009REVERSE", "eiken_may1500spring2017051009REVERSE",
# "eiken_may0500spring2017051015REVERSE".

# 変数の定義
altitudes_num = 3 # 出発高度の種類数
hours_num = 4 # 開始時間の種類数
day_traj_num = 12 # 1 日あたりの流跡線の数

# ファイルの並びの規則性により必要な流跡線のまとまりを対象としてプロットが可能である。
# このプロットでは成分分析測定日の 1 日分をプロットの対象にしているので
# 1 日 4 つの時間×3 つの高度で 12 本分の流跡線をプロットする必要がある。
# そのため流跡線の総数を 1 日あたりの流跡線の数の 12 で割っている。
for i in range(trajgroup.trajcount // day_traj_num):
    fig = plt.figure(figsize=[12, 11]) # 図のサイズ指定
    spec = gridspec.GridSpec(ncols=1, nrows=2,hspace=0.01,height_ratios=[5, 1]) # 描画領域を 2 つに分割
    ax0 = fig.add_subplot(spec[0]) # 地図を描画する領域として使用
    ax1 = fig.add_subplot(spec[1]) # 高度変化を描画する領域として使用
    bmap = bmap_params.make_basemap(ax=ax0) # basemap による地図の描画
    ax1.invert_xaxis() # x 軸（日時、Date Time）の反転
    # 開始日を取得（逆解析流跡線では元の流跡線の日時ができない場合もあるのでファイル名から取得）
    startday = pd.to_datetime(re.search(r'¥d{10}', reversetrajgroup[i * day_traj_num].trajid).group(),
                              format='%Y%m%d%H', utc=True).tz_convert('Asia/Tokyo')

    # 12 本中初めの 3 本の流跡線については高さ別の凡例を作成するため処理を分けている。
    for j in range(i * day_traj_num, i * day_traj_num + altitudes_num):
        # 流跡線の日時と高さ情報の取得。ただし逆解析流跡線から元の流跡線の時刻取得ができない場合もあるので分けて処理している
        datetime0 = pd.to_datetime(re.search(r'¥d{10}', reversetrajgroup[j].trajid).group(),
                                   format='%Y%m%d%H', utc=True).tz_convert('Asia/Tokyo')
        datetime = pd.to_datetime(reversetrajgroup[j].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
        altitude = reversetrajgroup[j].data.geometry.z
        # 地図に流跡線をプロット
        bmap.plot(*reversetrajgroup[j].path.xy, c=reversetrajgroup[j].trajcolor,
                  latlon=True, zorder=20, ax=ax0, ls=ls_dict[datetime0.hour],
                  marker=marker_dict[reversetrajgroup[j].trajcolor], markersize=8, markevery=20,
                  label=str(int(re.search(r'(500|1[0-5]00)', reversetrajgroup[j].trajid).group())) +' m (reverse)')
        # 流跡線の日時と高さ情報をプロット
        ax1.plot(datetime, altitude, color=reversetrajgroup[j].trajcolor, ls=ls_dict[datetime0.hour],
                 marker=marker_dict[reversetrajgroup[j].trajcolor], markersize=8, markevery=20)

    # 凡例に線の種類による時刻情報を追加。
    handles, _ = ax0.get_legend_handles_labels() # 凡例情報を取得
    lines = [Line2D([0], [0], color='black', linewidth=2, linestyle=l,
                    label=str(pd.to_datetime(
                        re.search(r'¥d{10}', reversetrajgroup[i* day_traj_num + altitudes_num * k].trajid).group(),
                        format='%Y%m%d%H', utc=True).tz_convert(
                        'Asia/Tokyo').strftime('%Y-%m-%d %H:%M'))) for k,l in zip(range(hours_num), ls_dict.values())]
    handles.extend(lines) # 凡例を追加
    legend = ax0.legend(handles=handles, loc='best', fontsize=18) # 凡例の場所とフォントサイズを調整
    legend.set_zorder(level=25) # 凡例が隠れないように zorder を調整

    # 残りの 9 本の流跡線については凡例作成処理を除きプロットする。
    for j in range(i * day_traj_num + altitudes_num, i * day_traj_num + day_traj_num):
        datetime0 = pd.to_datetime(re.search(r'¥d{10}', reversetrajgroup[j].trajid).group(),
                                   format='%Y%m%d%H', utc=True).tz_convert('Asia/Tokyo')
        datetime = pd.to_datetime(reversetrajgroup[j].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
        altitude = reversetrajgroup[j].data.geometry.z
        bmap.plot(*reversetrajgroup[j].path.xy, c=reversetrajgroup[j].trajcolor,
                  latlon=True, zorder=20, ax=ax0, ls=ls_dict[datetime0.hour],
                  marker=marker_dict[reversetrajgroup[j].trajcolor], markersize=8, markevery=20)
        ax1.plot(datetime, altitude, color=reversetrajgroup[j].trajcolor, ls=ls_dict[datetime0.hour],
                 marker=marker_dict[reversetrajgroup[j].trajcolor], markersize=8, markevery=20)

    # 目盛りや軸ラベルのフォントサイズ調整。
    ax1.tick_params(labelsize=15)
    ax1.set_xlabel('Date Time', fontsize=18)
    ax1.set_ylabel('Altitude ($m$)', fontsize=18)

    # 画像ファイルの保存。日付の情報は流跡線から取得（元の流跡線開始日基準）。
    plt.savefig('img/plot_day/reverse/eiken_reverse_{year}{month}{day}.png'.format(
        year=str(startday.year),
        month=str(startday.month).zfill(2),
        day=str(startday.day).zfill(2)
    ), dpi=600, bbox_inches='tight', pad_inches=0)
    plt.close() # 画像が多数あるため表示出力しない設定
```

コード 17. 流跡線と逆解析流跡線を合わせて成分分析測定日毎にプロットするコード.

```
# 流跡線と逆解析流跡線を成分分析測定日毎にプロットするコード。コード 2 が実行済みであることが必要。

# 変数の定義
altitudes_num = 3 # 出発高度の種類数
hours_num = 4 # 開始時間の種類数
day_traj_num = 12 # 1 日あたりの流跡線の数

for i in range(trajgroup.trajcount // day_traj_num):
    fig = plt.figure(figsize=[12, 11]) # 図のサイズ指定
    spec = gridspec.GridSpec(ncols=1, nrows=2,hspace=0.01,height_ratios=[5, 1]) # 描画領域を 2 つに分割
    ax0 = fig.add_subplot(spec[0]) # 地図を描画する領域として使用
    ax1 = fig.add_subplot(spec[1]) # 高度変化を描画する領域として使用
    bmap = bmap_params.make_basemap(ax=ax0) # basemap による地図の描画
    ax1.invert_xaxis() # x 軸（日時、Date Time）の反転
    # 開始日を取得
    startday = pd.to_datetime(trajgroup[i * day_traj_num].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')[0]

    # 各種類ごとの流跡線のうち、12 本中初めの 3 本の流跡線については高さ別の凡例を作成するため処理を分けている。
    for j in range(i * day_traj_num, i * day_traj_num + altitudes_num):
        datetime = pd.to_datetime(trajgroup[j].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
        datetime_r = pd.to_datetime(reversetrajgroup[j].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
        altitude = trajgroup[j].data.geometry.z
        altitude_r = reversetrajgroup[j].data.geometry.z
        # 地図に流跡線をプロット
        bmap.plot(*trajgroup[j].path.xy, c=trajgroup[j].trajcolor, latlon=True, zorder=20, ax=ax0,
                  ls=ls_dict[datetime[0].hour], marker=marker_dict[trajgroup[j].trajcolor], markersize=8,
                  markevery=20, label=str(int(altitude[0])) +' m')
        bmap.plot(*reversetrajgroup[j].path.xy, c=reversetrajgroup[j].trajcolor, latlon=True, zorder=20,ax=ax0,
                  ls=ls_dict[datetime[0].hour], marker=marker_dict[reversetrajgroup[j].trajcolor], markersize=8,
                  markevery=20, label=str(int(altitude[0])) +' m (reverse)')
        # 流跡線の日時と高さ情報をプロット
        ax1.plot(datetime, altitude, color=trajgroup[j].trajcolor, ls=ls_dict[datetime[0].hour],
                 marker=marker_dict[trajgroup[j].trajcolor], markersize=8, markevery=20)
        ax1.plot(datetime_r, altitude_r, color=reversetrajgroup[j].trajcolor, ls=ls_dict[datetime[0].hour],
                 marker=marker_dict[reversetrajgroup[j].trajcolor], markersize=8, markevery=20)

    # 凡例に線の種類による時刻情報を追加。
    handles, labels = ax0.get_legend_handles_labels()
    lines = [Line2D([0], [0], color='black', linewidth=2, linestyle=l, label=str(pd.to_datetime(
        trajgroup[i * day_traj_num + altitudes_num * k].data['DateTime'], utc=True).dt.tz_convert(
        'Asia/Tokyo')[0].strftime('%Y-%m-%d %H:%M'))) for k,l in zip(range(hours_num), ls_dict.values())]
    handles.extend(lines) # 凡例を追加
    legend = ax0.legend(handles=handles, loc='best', fontsize=18, ncol=2) # 凡例の場所とフォントサイズを調整
    legend.set_zorder(level=25) # 凡例が隠れないように zorder を調整

    # 残りの流跡線については凡例作成処理を除きプロットする。
    for j in range(i * day_traj_num + altitudes_num, i * day_traj_num + day_traj_num):
        datetime = pd.to_datetime(trajgroup[j].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
        datetime_r = pd.to_datetime(reversetrajgroup[j].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
        altitude = trajgroup[j].data.geometry.z
        altitude_r = reversetrajgroup[j].data.geometry.z
        bmap.plot(*trajgroup[j].path.xy, c=trajgroup[j].trajcolor, latlon=True, zorder=20, ax=ax0,
                  ls=ls_dict[datetime[0].hour], marker=marker_dict[trajgroup[j].trajcolor], markersize=8, markevery=20)
        bmap.plot(*reversetrajgroup[j].path.xy, c=reversetrajgroup[j].trajcolor, latlon=True, zorder=20, ax=ax0,
                  ls=ls_dict[datetime[0].hour], marker=marker_dict[reversetrajgroup[j].trajcolor],
                  markersize=8, markevery=20)
        ax1.plot(datetime, altitude, color=trajgroup[j].trajcolor, ls=ls_dict[datetime[0].hour],
                 marker=marker_dict[trajgroup[j].trajcolor], markersize=8, markevery=20)
        ax1.plot(datetime_r, altitude_r, color=reversetrajgroup[j].trajcolor,  ls=ls_dict[datetime[0].hour],
                 marker=marker_dict[reversetrajgroup[j].trajcolor], markersize=8, markevery=20)

    # 目盛りや軸ラベルのフォントサイズ調整
    ax1.tick_params(labelsize=15)
    ax1.set_xlabel('Date Time', fontsize=18)
    ax1.set_ylabel('Altitude ($m$)', fontsize=18)

    # 画像ファイルの保存。日付の情報は流跡線から取得（元の流跡線開始日基準）。
    plt.savefig('img/plot_day/traj_and_reverse/eiken_traj_and_reverse_{year}{month}{day}.png'.format(
        year=str(startday.year),
        month=str(startday.month).zfill(2),
        day=str(startday.day).zfill(2)
    ),dpi=600, bbox_inches='tight', pad_inches=0)
    plt.close() # 画像が多数あるため表示出力しない設定
```

コード 18. マスク領域を表示するコード.

```
# マスク領域を表示するコード。コード 7 のマスク領域作成のコードが実行済みであることが必要
# 使用するライブラリをインポート
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
from matplotlib.collections import PatchCollection
from matplotlib.patches import Patch
import shapely
from descartes import PolygonPatch
import numpy as np

# プロット領域の作成
fig = plt.figure(figsize=(12,8))
ax = fig.add_subplot(111)

# Basemap を呼び出し、インスタンスを生成
m = Basemap(projection='cyl', lat_0=35, lon_0=135,
            llcrnrlat=-5, urcrnrlat=50, llcrnrlon=90,
            urcrnrlon=165, resolution='l', area_thresh=500)

# 海岸線と国境線を描く
m.drawcoastlines()
m.drawcountries()

# 緯度線と経度線を引く。フォントサイズとラベル表示位置も指定。
m.drawmeridians(np.arange(0, 360, 20), color="k", fontsize=18,
                labels=[False, False, True, False])
m.drawparallels(np.arange(-90, 90, 10), color="k", fontsize=18,
                labels=[True, False, False, False])

# カラーリストとハッチリストを作成
color_list = ['#b5ffb9', '#f9bc86', '#f1cbff', '#a3acff']
hatch_list = ['//', r'¥¥', '++', '..']

# for ループでマスク領域をプロットする
# if 関数等で Polygon と MultiPolygon およびそれ以外で処理を変化
# zip 関数利用でカラーとハッチのリストも同時に変化させている
for poly, color, hatch in zip(mask_list, color_list, hatch_list):
    patches = []
    if poly.geom_type == 'Polygon':
        mpoly = shapely.ops.transform(m, poly)
        patches.append(PolygonPatch(mpoly))
    elif poly.geom_type == 'MultiPolygon':
        for subpoly in poly.geoms:
            mpoly = shapely.ops.transform(m, subpoly)
            patches.append(PolygonPatch(mpoly))
    else:
        print(poly, 'はポリゴンでもマルチポリゴンでもありません。')
    # マスク領域の塗りつぶしとハッチの追加
    ax.add_collection(PatchCollection(patches, match_original=False, edgecolors='k',facecolors=color, hatch=hatch))

# 凡例の要素作成
legend_elements = [Patch(facecolor=color_list[0], hatch=hatch_list[0], lw=6, label='EA'),
                   Patch(facecolor=color_list[1], hatch=hatch_list[1], lw=6, label='J'),
                   Patch(facecolor=color_list[2], hatch=hatch_list[2], lw=6, label='SA'),
                   Patch(facecolor=color_list[3], hatch=hatch_list[3], lw=6, label='PO')]
plt.legend(handles=legend_elements, loc='upper right', fontsize=24) # 凡例の追加
plt.tight_layout() # 図のはみだしがないようにレイアウト調整
plt.savefig('img/basemap.png', dpi=600) # 図の保存
plt.show()
```

コード 19. 流跡線をマスク領域別にプロットするコード.

```
# 流跡線をマスク領域別にプロットするためのコード。
# 流跡線の情報を記録したデータフレームを作成するコード 7-13 が実行済みであることが必要。
# 2020 年度と指定の領域をリストに抽出する部分を書き換えれば、他の年度や全期間のマスク領域別プロットが可能。
# 使用するライブラリをインポート

import pysplit
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('csv/eiken_traj_info.csv') # csv ファイルの読み込み
# 流跡線ファイルの読み込み
trajgroup = pysplit.make_trajectorygroup(r'D:¥trajectories¥eiken_FY2017-FY2020¥eiken_*')

# 地図描画領域や投影方法、地図解像度、描画する最小面積、緯度経度のフォントサイズを指定
mapcorners =  [90, 0, 160, 50] # 値は左下の経度、左下の緯度、右上の経度、右上の緯度を表す
standard_pm = None # 地図の投影方法が cyl（円筒形、メルカトル図法）では必要ないがそれ以外の場合に設定が必要
param_dict = {'projection':'cyl','resolution':'l', 'area_threshold':500, 'latlon_fs':18}

# 地図のパラメータを設定
bmap_params = pysplit.MapDesign(mapcorners, standard_pm, **param_dict)

# 色、線、マーカーの種類を設定
# この例では流跡線の種類と高さに色とマーカーを対応させ、時間に線の種類を対応させているが
# 色とマーカーを紐づける必要はない
# 流跡線の種類と高さによって色が変わるように設定
color_dict = {'traj':{500.0 : 'blue', 1000.0 : 'gold', 1500.0 : 'black'},
              'reversetraj':{500.0 : 'red', 1000.0 : 'purple', 1500.0 : 'cyan'}}

# 流跡線の時刻によって線種が変わるように設定
ls_dict = {12:'-', 18:'--', 0:'-.', 6:':'}

# 流跡線の色によってマーカーが変わるように設定
marker_dict = {'blue' : '^', 'gold' : 's', 'black' : 'o',
               'red' : 'v', 'purple' : 'x', 'cyan' : '*'}

# 図のサイズ指定と描画領域を 4×4 の 16 の領域に分割
fig, ax = plt.subplots(ncols=4,nrows=4, figsize=(40, 30))
ax = ax.flatten() # ax にアクセスしやすいように二次元配列を 1 次元配列化

# 流跡線ファイルから高さの情報を取得しそれに応じて線の色を変更
for i in range(trajgroup.trajcount):
    altitude0 = trajgroup[i].data.geometry.z[0]
    trajgroup[i].trajcolor = color_dict['traj'][altitude0]

# データフレームの領域の細かい分類の列名を取り出して for ループへ
for z,j in enumerate(df.columns[37:53]):
    index_list = df.query('{} == True & FY == 2020'.format(j)).index.tolist() # 2020 年度と指定の領域をリストに抽出
    bmap = bmap_params.make_basemap(ax=ax[z]) # basemap による地図の描画

    # 3 つのマスク領域にまたがる場合の流跡線のテキスト挿入
    if j in df.columns[47:51]:
        ax[z].text(0.05, 0.05,j, transform=ax[z].transAxes, fontsize=54,
                   bbox=dict(facecolor='w',edgecolor='k',alpha=0.75)).set_zorder(level=25)
    # 2 つのマスク領域にまたがる場合の流跡線のテキスト挿入
    elif j in df.columns[41:47]:
        ax[z].text(0.45, 0.05,j, transform=ax[z].transAxes, fontsize=54,
                   bbox=dict(facecolor='w',edgecolor='k',alpha=0.75)).set_zorder(level=25)
    # その他の領域の流跡線のテキスト挿入
    else:
        ax[z].text(0.6, 0.05,j, transform=ax[z].transAxes,fontsize=54,
                   bbox=dict(facecolor='w',edgecolor='k',alpha=0.75)).set_zorder(level=25)
    # 領域毎に地図へ流跡線をプロット
    for i in index_list:
        datetime = pd.to_datetime(trajgroup[i].data['DateTime'], utc=True).dt.tz_convert('Asia/Tokyo')
        bmap.plot(*trajgroup[i].path.xy, c=trajgroup[i].trajcolor, latlon=True, zorder=20,
                  ls=ls_dict[datetime[0].hour], marker=marker_dict[trajgroup[i].trajcolor],
                  markersize=8, markevery=20)

plt.tight_layout() # 図のはみだしがないようにレイアウト調整
plt.savefig('img/region_plot_FY2020.png', dpi=600) # 画像ファイルの保存。
```