Python を用いた統計的流跡線解析（付録）

流跡線作成や統計的流跡線解析に関連する Python のコードを以下に掲載する.

コード 1. 10 年分の流跡線ファイルを一括で作成するコード.

```python
# 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
from dateutil import tz
import pandas as pd

# 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/Okinawa_FY2012-FY2021_1H' # 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 = 'okinawa_'

altitudes = [500, 1000, 1500] # 高度のリスト。
location = (26.3769593331371, 127.83470843018222) # 沖縄局の緯度経度。
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。

JST = tz.gettz('Asia/Tokyo') # 日本標準時
UTC = tz.gettz('UTC') # 協定世界時
# 1 時間おき10 年度分の連続日付データ作成
dt_index = pd.date_range(start='2012-04-01 1:00:00', end='2022-04-01 0:00:00', freq='1H', 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. 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 geopandas as gpd
import xarray as xr
import shapely
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
from dateutil import tz
import re

# 流跡線ファイルの読み込み
trajgroup = pysplit.make_trajectorygroup(r'D:¥trajectories¥Okinawa_FY2012-FY2021_1H¥*')

# 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¥Okinawa_FY2012-FY2021_1H¥reversetraj¥okinawa_*')
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 3．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 1,2 が実行済みであることが必要。
# コード 3、4 は一つの関数なのでスペースの制約上分割しているが、別々にではなく一つのコードとして実行する必要があることに注意。

def traj_make_df(trajgroup):
    """
    「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 4.　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 3 の続き。
# コード 3、4 は一つの関数なのでスペースの制約上分割しているが、別々にではなく一つのコードとして実行する必要があることに注意。
    # 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)

    return df
```

コード 5. 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 1-4 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)

# ['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)

# 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)

# 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_[0,14:16]]], 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'])

df.to_csv('csv/okinawa_traj_info_FY2012-FY2021_1H.csv', index=False) # csv で保存
```

コード 6.　Xarrray のデータセットを作成するコード(1).

```
# Xarray のデータセットを作成するコード(1)。コード 5 まで実行済みであることが必要。
# 軸に設定するための経度・緯度・高度・時間・ステップ数(遡行時間)の情報を定義
lon = np.arange(90,160)
lat = np.arange(0,50)
alt = [500, 1000, 1500]
JST = tz.gettz('Asia/Tokyo') # 日本標準時
dt_index = pd.date_range(start='2012-04-01 1:00:00', end='2022-04-01 0:00:00', freq='1H', tz=JST)
attrs = {"units": "hours since 2012-04-01 01:00:00"}
step = trajgroup.trajectories[0].data.index

# 軸の長さの取得
lon_size = len(lon)
lat_size = len(lat)
alt_size = len(alt)
dt_size = len(dt_index)
step_size = len(step)

# numpy.ndarray を作成
data_count = np.random.rand(lon_size, lat_size, dt_size, alt_size)
traj_lat = np.random.rand(dt_size, alt_size, step_size)
traj_lon = np.random.rand(dt_size, alt_size, step_size)
traj_alt = np.random.rand(dt_size, alt_size, step_size)

# Xarray の次元軸(dims)とラベル(coords)を定義
dims1 = ('lon', 'lat', 'time', 'alt')
coords1 = {'lon': ('lon', lon),
          'lat': ('lat', lat),
          'time':  ('time', list(range(dt_size)), attrs),
          'alt': ('alt', alt)}
dims2 = ('time' , 'alt', 'step')
coords2 = {'time':  ('time', list(range(dt_size)), attrs),
           'alt': ('alt', alt),
           'step': ('step', step)}

# Xarray のデータアレイを作成
data_count = xr.DataArray(data_count, coords1, dims1)
traj_lat = xr.DataArray(traj_lat, coords2, dims2)
traj_lon = xr.DataArray(traj_lon, coords2, dims2)
traj_alt = xr.DataArray(traj_alt, coords2, dims2)

# グリッド用の経度・緯度範囲を設定
xedges = list(range(90, 161))
yedges = list(range(0, 51))

# 流跡線のデータから必要な情報を Xarray のデータアレイに抽出
for i,traj in enumerate(trajgroup):
    x = traj.data.geometry.x.values
    y = traj.data.geometry.y.values
    z = traj.data.geometry.z.values
    H, xedges, yedges = np.histogram2d(x, y, bins=(xedges, yedges))
    data_count[:,:,i//3,i%3] = H
    # ステップ数が 121 に満たないものは不足分を欠測値(np.nan)で穴埋め
    if len(x) != 121:
        x = np.pad(x, (0,121-len(x)), "constant", constant_values=np.nan)
        y = np.pad(y, (0,121-len(y)), "constant", constant_values=np.nan)
        z = np.pad(z, (0,121-len(z)), "constant", constant_values=np.nan)
    else:
        pass
    traj_lon[i//3,i%3,:] = x
    traj_lat[i//3,i%3,:] = y
    traj_alt[i//3,i%3,:] = z

# csv ファイルの読み込み
df2 = pd.read_csv('csv/Okinawa_st_2012-2021.csv')
# 2012 年から 2021 年の 10 年分の沖縄局 1 時間値データcsv。あらかじめポテンシャルオゾン(PO)も計算し、年度の情報も追加。先頭 3 行のみ表示
#           Date,  SO2,  NO,  NO2,  NOx,  Ox,  SPM,  PM2.5,  PO,    FY
# 2012/4/1 1:00,  0,    1,   2,    3,     53,  44,   21,     54.7,  2012
# 2012/4/1 2:00,  0,    1,   2,    3,     53,  41,   21,     54.7,  2012
# 2012/4/1 3:00,  0,    0,   1,    1,     54,  27,   21,     54.9,  2012

df2['Date'] = pd.to_datetime(df2['Date']) # データ型を datetime 型に変更
df2.index = df2['Date'] # 'Date'をインデックスに指定
# インデックスに指定した 'Date'を除いてデータフレームを再定義
df2 = df2.reindex(columns=['SO2', 'NO', 'NO2', 'NOx', 'Ox', 'SPM', 'PM2.5', 'PO', 'FY'])
df2 = df2.astype('float64') # データ型を float64 型に変更
df2.FY = df2.FY.astype('int64') # 'DFY'のみデータ型を int64 型に変更
```

コード 7.　Xarrray のデータセットを作成するコード(2).

```
# Xarray のデータセットを作成するコード(2)。コード 6 の続き。

# Xarray の次元軸(dims)とラベル(coords)を定義
dims3 = ('time')
dims4 = ('time', 'alt')
coords3 = {'time':  ('time', list(range(dt_size)), attrs)}
coords4 = {'time':  ('time', list(range(dt_size)), attrs),
           'alt': ('alt', alt)}

# 2012 年から 2021 年の 10 年分の沖縄局 1 時間値データ csv からデータを取得しデータアレイを作成
SO2 = xr.DataArray(df2['SO2'].values, coords3, dims3)
NO = xr.DataArray(df2['NO'].values, coords3, dims3)
NO2 = xr.DataArray(df2['NO2'].values, coords3, dims3)
NOx = xr.DataArray(df2['NOx'].values, coords3, dims3)
Ox = xr.DataArray(df2['Ox'].values, coords3, dims3)
SPM = xr.DataArray(df2['SPM'].values, coords3, dims3)
PM25 = xr.DataArray(df2['PM2.5'].values, coords3, dims3)
PO = xr.DataArray(df2['PO'].values, coords3, dims3)
FY = xr.DataArray(df2['FY'].values, coords3, dims3)

# 流跡線の情報をまとめたデータフレームから不適切な流跡線判定用のデータを取得しデータアレイを作成
Alt_check = xr.DataArray(df.Altitude_check.values.reshape(dt_size, alt_size), coords4, dims4)
r_Alt_check = xr.DataArray(df.r_Altitude_check.values.reshape(dt_size, alt_size), coords4, dims4)
Length = xr.DataArray(df.Length.values.reshape(dt_size, alt_size), coords4, dims4)
r_Length = xr.DataArray(df.r_Length.values.reshape(dt_size, alt_size), coords4, dims4)
error = xr.DataArray(df.Integration_error.values.reshape(dt_size, alt_size), coords4, dims4)
error_check = xr.DataArray(df.Relative_errors_check.values.reshape(dt_size, alt_size), coords4, dims4)

# Xarray のデータアレイをまとめて、Xarray のデータセットを作成
ds = xr.Dataset()
ds['data_count'] = data_count
ds['traj_lat'] = traj_lat
ds['traj_lon'] = traj_lon
ds['traj_alt'] = traj_alt
ds['SO2'] = SO2
ds['NO'] = NO
ds['NO2'] = NO2
ds['NOx'] = NOx
ds['Ox'] = Ox
ds['SPM'] = SPM
ds['PM25'] = PM25
ds['PO'] = PO
ds['FY'] = FY
ds['Alt_check'] = Alt_check
ds['r_Alt_check'] = r_Alt_check
ds['Length'] = Length
ds['r_Length'] = r_Length
ds['error'] = error
ds['error_check'] = error_check

ds = xr.decode_cf(ds) # Xarray の時間軸情報('time')を整数値から時間に変換
ds.to_netcdf(r'netcdf/Okinawa_st_2012-2021.nc') # Xarray のデータセットをファイル保存
```

コード 8. 統計的流跡線解析の準備（ライブラリの読み込みと関数定義）.

```
# 統計的流跡線解析の準備（ライブラリの読み込みと関数定義）を行うコード。Xarray データセットは作成済みであること。

# 使用するライブラリをインポート
import numpy as np
import pandas as pd
import geopandas as gpd
import xarray as xr
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib import colors
from matplotlib.patches import Patch
from matplotlib.colors import ListedColormap
from mpl_toolkits.axes_grid1 import make_axes_locatable
from mpl_toolkits.basemap import Basemap
import japanize_matplotlib

# 重みづけ関数の定義
def cal_weight(data_count, data, avg):
    """
    エンドポイントの頻度に応じて重みづけを行う関数
    [入力]
    data_count:エンドポイントの頻度
    data:重みづけ対象のデータ
    avg:エンドポイントの総セル平均値（全セル内の総エンドポイント数 ÷ 総セル数）
    [出力]
    data * w:重みづけ w を掛けたデータ
    """
    if data_count > 3 * avg:
        return data
    elif data_count > 1 * avg:
        return data * 0.7
    elif data_count > 0.5 * avg:
        return data * 0.42
    else:
        return data * 0.17

# 統計的流跡線解析の結果をプロットするための関数の定義
def plot_stat_traj(grid_data, text, cmap='Blues', log_axes = False, log_axes_max=1e6,
                   colorbar=True, projection='cyl', lat_0=35, lon_0=135,
                  lat_min=0, lat_max=50, lon_min=90, lon_max=160,
                  resolution='l', area_thresh=500):
    """
    CWT や PSCF の計算結果を地図上にプロットする関数
    [入力]
    grid_data:二次元の軸（緯度・経度）を持った「xarray.DataArray」オブジェクト。
    text:図に表示するテキスト。「japanize_matplotlib」ライブラリを用いれば日本語も可。
    cmap:カラーマップの種類。デフォルトは「'Blues'」。
    log_axes:カラーバーの軸を「True」で対数表示に切り替える。デフォルトは「False」（対数表示ではない）。
    log_axes_max:カラーバーが対数表示の際の最大値。デフォルトは「1e6（1,000,000）」。
    colorbar:カラーバーの表示・非表示を切り替える。デフォルトは「True」（表示する）。
    projection:地図の図法の設定。詳細は Basemap ライブラリのドキュメントを参照。デフォルトは「'cyl'」（正距円筒図法）。
    lat_0, lon_0:緯度と経度の原点座標の指定。デフォルトは「lat_0=35, lon_0=135」。
    lat_min, lat_max:経度の範囲指定。デフォルトは「lat_min=0, lat_max=50」。
    lon_min, lon_max:緯度の範囲指定。デフォルトは「lon_min=90, lon_max=160」。
    resolution:解像度の指定。解像度は低いほうから順に「 c (crude), l (low), i (intermediate), h (high), f (full) 」。
      高解像度を設定すると描画に時間がかかる。デフォルトは「'l'」。
    area_thresh:面積が指定した値 km^2 以下の島や湖が描画されなくなる。数字が小さいと描画に時間がかかる。デフォルトは「500」。
    [出力]
    fig:matplotlib の「Figure」オブジェクト（図の描画領域）。
    ax:matplotlib の「Axes」オブジェクト（グラフ）。
    m:Basemap の「Basemap」オブジェクト（地図描画）。
    """
```

コード 9. 統計的流跡線解析の準備（関数定義の続きとデータセット読み込み）

```
# 統計的流跡線解析の準備(関数定義の続きとデータセット読み込み)のコード。コード 8 の続き。
# コード 8、9 はスペースの制約上分割しているが、関数を分割して記述しているので一つのコードとして実行する必要があることに注意。
# 統計的流跡線解析の結果をプロットするための関数(plot_stat_traj)定義の続き

    # 「Figure」と「Axes」オブジェクトを作成
    fig = plt.figure(figsize=(12,8))
    ax = fig.add_subplot(111)

    # 「Basemap」オブジェクトを生成
    m = Basemap(projection=projection, lat_0=lat_0, lon_0=lon_0,
                llcrnrlat=lat_min, urcrnrlat=lat_max, llcrnrlon=lon_min,
                urcrnrlon=lon_max, resolution=resolution, area_thresh=area_thresh)

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

    # カラーバーの対数表示切替とカラープロットの作成
    if log_axes == True:
        fishnet = m.pcolor(grid_data.lon+0.5, grid_data.lat+0.5, grid_data.T, cmap=cmap,
                           norm=colors.LogNorm(vmin=0+1, vmax=log_axes_max+1))
    else:
        fishnet = m.pcolor(grid_data.lon+0.5, grid_data.lat+0.5, grid_data.T, cmap=cmap)

    # カラーバーの表示・非表示とカラーバーの調整
    if colorbar == True:
        divider = make_axes_locatable(ax) # ax に紐付いた AxesDivider を取得
        cax = divider.append_axes("right", size="3%", pad=0.1) # append_axes で新しい axes を作成
        cb = fig.colorbar(fishnet, cax=cax) # 新しく作成した axes である cax を渡す。
        cb.set_label(grid_data.name, 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
    else:
        pass

    # テキストの挿入
    ax.text(0.05, 0.05, r'{}'.format(text), transform=ax.transAxes, fontsize=32,
            bbox=dict(facecolor='w',edgecolor='k',alpha=0.75)).set_zorder(level=25)

    return fig, ax, m
# 統計的流跡線解析の結果をプロットするための関数(plot_stat_traj)定義の終了


# データセットの読み込みと基準を満たさないような流跡線を除く処理を行っている。
ds = xr.open_dataset('netcdf/Okinawa_st_2012-2021.nc')
ds = ds.where((ds.Length == 121) & (ds.r_Length == 121) & (ds.error_check == True) &
              (ds.Alt_check == True) & (ds.r_Alt_check == True))

# 各季節毎に抽出したデータセットを作成。月を指定することで各季節の抽出を行っている。
# ただし、大気モニタのデータは通常 1 時～24 時を 1 日としているため、そのままでは季節範囲にずれが 1 時間生じる。
# これを補正するためデータセットのメソッド shift()を使用。
# この場合先頭の値(2012-04-01 01:00:00)が欠測となるため値を fill_value=4 としている
ds_spring = ds.isel(time=ds.time.dt.month.shift(time=1,fill_value=4).isin([3,4,5]))
ds_summer = ds.isel(time=ds.time.dt.month.shift(time=1,fill_value=4).isin([6,7,8]))
ds_autumn = ds.isel(time=ds.time.dt.month.shift(time=1,fill_value=4).isin([9,10,11]))
ds_winter = ds.isel(time=ds.time.dt.month.shift(time=1,fill_value=4).isin([12,1,2]))

# プロットに用いるカラーマップの読み込みと自作のカラーマップ定義
Blues = mpl.colormaps['Blues']
Blues_list = Blues(np.linspace(0, 1, 6))
Blue6_cmap = ListedColormap(Blues_list, name='Blue6')
Blues_list = Blues(np.linspace(0, 1, 4))
Blue4_cmap = ListedColormap(Blues_list, name='Blue4')

# プロットに用いるための各種リストの作成
ds_list = [ds, ds_spring, ds_summer, ds_autumn, ds_winter]
name_list_file = ['', '_spring', '_summer', '_autumn', '_winter']
name_list_text = ['', '(春季)', '(夏季)', '(秋季)', '(冬季)']
name_list_text2 = ['', ',春季', ',夏季', ',秋季', ',冬季']
```

コード 10. 統計的流跡線の計算とプロットを作成するコード.

```
# 統計的流跡線の計算とプロットを作成するコード。コード 9 までの実行が必要。

# エンドポイントの頻度と重みづけの可視化
for i in range(len(ds_list)):
    data_count = ds_list[i].data_count.sum(dim=('time','alt'))
    data_count = data_count.rename('Frequancy')

    fig, ax, m = plot_stat_traj(grid_data=data_count, text='エンドポイントの頻度{}'.format(name_list_text[i]),
                                cmap=Blue6_cmap, log_axes=True, log_axes_max=1e6)
    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/Okinawa_st/okinawa_FY2012-FY2021_1H_frequency{}.png'.format(name_list_file[i]),
                dpi=600, bbox_inches='tight') # 図の保存

    avg = data_count.sum()/data_count.size
    ones = np.ones(shape=(70,50))
    n_max = data_count.max()
    W = xr.apply_ufunc(cal_weight, data_count, ones, avg, vectorize=True)
    W = W.rename('Weight')

    fig, ax, m = plot_stat_traj(grid_data=W, text='重みづけ{}'.format(name_list_text[i]), cmap=Blue4_cmap, colorbar=False)
    w_list = [1.0, 0.7, 0.42, 0.17]
    # 凡例の作成
    legend_elements = [Patch(edgecolor='black', facecolor=Blues_list[-(j+1)],
                             lw=1, label=w_list[j]) for j in range(4)]
    ax.legend(title="Weight", title_fontsize=24, 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/Okinawa_st/okinawa_FY2012-FY2021_1H_Weight{}'.format(name_list_file[i]),
                dpi=600, bbox_inches='tight') # 図の保存

# CWT 及び WCWT の実行と結果のプロット
for i in range(len(ds_list)):
    weighted_concentration_sum = (ds_list[i].data_count * ds_list[i].Ox).sum(dim=('time','alt'), skipna=True)
    data_count = ds_list[i].data_count.sum(dim=('time','alt'))
    CWT = np.divide(weighted_concentration_sum, data_count.where(data_count != 0, 1))
    CWT = CWT.rename('CWT')

    fig, ax, m = plot_stat_traj(grid_data=CWT, text='Ox(CWT{})'.format(name_list_text[i]))

    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/Okinawa_st/okinawa_FY2012-FY2021_1H_Ox_CWT{}.png'.format(name_list_file[i]),
                dpi=600, bbox_inches='tight') # 図の保存

    avg = data_count.sum()/data_count.size
    WCWT = xr.apply_ufunc(cal_weight, data_count, CWT, avg, vectorize=True)
    WCWT = WCWT.rename('WCWT')

    fig, ax, m = plot_stat_traj(grid_data=WCWT, text='Ox(WCWT{})'.format(name_list_text2[i]))

    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/Okinawa_st/okinawa_FY2012-FY2021_1H_Ox_WCWT{}.png'.format(name_list_file[i]),
                dpi=600, bbox_inches='tight') # 図の保存

# PSCF 及び WPSCF の実行と結果のプロット
for i in range(len(ds_list)):
    data_count = ds_list[i].data_count.sum(dim=('time','alt'))
    criterion_over_count = ds_list[i].where((ds_list[i].Ox > 60)).data_count.sum(dim=('time','alt'))
    PSCF = np.divide(criterion_over_count, data_count.where(data_count != 0, 1))
    PSCF = PSCF.rename('PSCF')

    fig, ax, m = plot_stat_traj(grid_data=PSCF, text='Ox(PSCF{})'.format(name_list_text[i]))

    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/Okinawa_st/okinawa_FY2012-FY2021_1H_Ox_PSCF{}.png'.format(name_list_file[i]),
                dpi=600, bbox_inches='tight') # 図の保存

    avg = data_count.sum()/data_count.size
    WPSCF = xr.apply_ufunc(cal_weight, data_count, PSCF, avg, vectorize=True)
    WPSCF = WPSCF.rename('WPSCF')

    fig, ax, m = plot_stat_traj(grid_data=WPSCF, text='Ox(WPSCF{})'.format(name_list_text2[i]))

    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/Okinawa_st/okinawa_FY2012-FY2021_1H_Ox_WPSCF{}.png'.format(name_list_file[i]),
                dpi=600, bbox_inches='tight') # 図の保存
```