

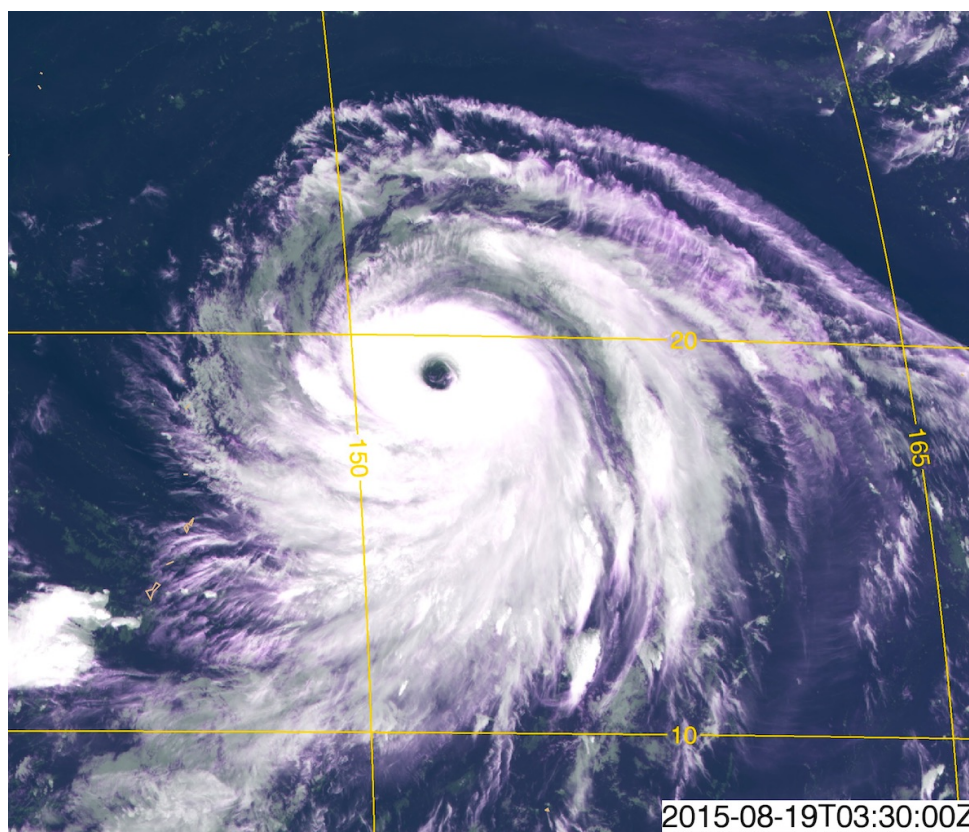
第 12 回 VL 講習会

～雲と大気成分で診る地球～

2018 年 9 月 10-11 日

B コース「衛星データで台風の雲を診る」

実習用マニュアル



台風 16 号 (2015 年)

東北大学 大学院理学研究科
大気海洋変動観測研究センター
気候物理学分野

目次

1. 実習について.....	3
1.1 概要	3
1.2 目的.....	5
1.3 使用データ	5
1.3.1 ひまわり 8 号由来のデータ	5
1.3.2 CloudSat/CALIPSO 由来のデータ	6
1.4 実習の流れ	8
2. PYTHON の使い方.....	9
2.1 PYTHON の基本要素	9
2.1.1 変数	9
2.1.2 リスト	9
2.1.3 ディクショナリ	9
2.1.4 タプル	10
2.1.5 if, else	10
2.1.6 for, while	10
2.1.7 関数	10
2.1.8 Numpy	11
2.2 データのプロット	15
2.2.1 一次元データのプロット	15
2.2.2 二次元データのプロット	16
2.3 データの補間.....	17
3. 衛星データ解析.....	19
3.1 NETCDF ファイルの扱いとデータの可視化	19
3.2 台風の雲の空間分布と日変化の解析	23
3.3 台風の雲の鉛直分布の解析	35

1. 実習について

1.1 概要

第 12 回バーチャルラボラトリー (VL) 講習会「雲と大気成分で診る地球」の B コースでは、台風の雲を解析対象として、雲のリモートセンシングの基礎を学び、衛星観測データ解析を体験する。実習は各自のパソコンで行う。本マニュアルでは、二日間かけて行う実習の内容についてまとめられている。実習は Python を用いて行うため、初日は主に衛星データ解析の観点から Python の基本操作を学び、二日目に実習課題に取り組む。

本実習では、2015 年 8 月に発生した台風 16 号 (アッサニー) を解析対象とする。日本時間 2015 年 8 月 15 日 3 時 (協定世界時 14 日 18 時) に北緯 15 度 10 分、東経 161 度 25 分で熱帯低気圧が発達し台風となり、16 日 21 時に暴風域が発生し、19 日には大型の台風となり、25 日に温帯低気圧に変わった。アッサニー台風の発達から衰退までの過程を図 1.1 に示す。また、台風のベストトラックデータから得られた中心位置と中心気圧の時系列を図 1.2 に示す。

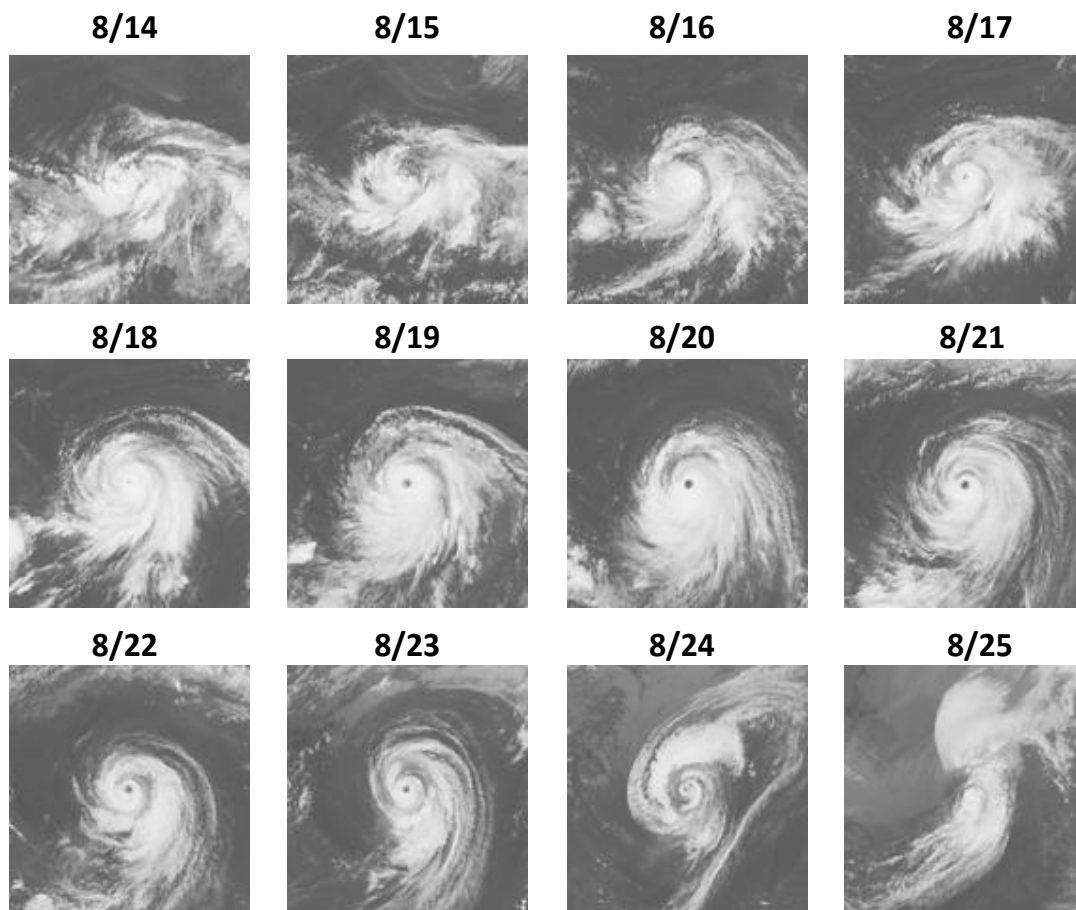


図 1.1 2015 年 8/14 から 8/25 の 12:00 JST におけるひまわり 8 号の赤外バンド (10.4 μm) 画像。

Credits : デジタル台風 (<http://agora.ex.nii.ac.jp/digital-typhoon/summary/wnp/s/201516.html.ja>)

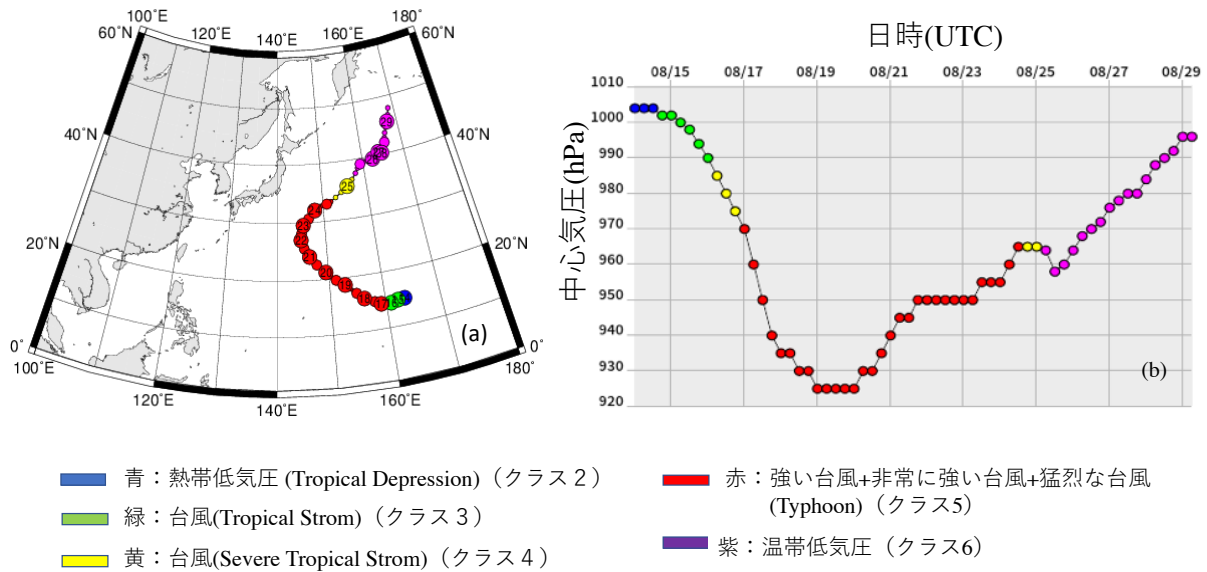


図 1.2 (a)Atsani 台風のベストトラックより得られた中心位置と (b) 中心気圧の時系列。
Credits : デジタル台風 (<http://agora.ex.nii.ac.jp/digital-typhoon/summary/wnp/s/201516.html.ja>)

2015 年 8 月 19 日 00:00 UTC にアッサニー台風が強い台風に変った後、およそ 3:29 UTC に A-train(衛星コンステレーション)がその台風の目の西側を通過した (図 1.3) 。そのため A-train に所属し、能動型測器を搭載した CloudSat と CALIPSO 衛星が台風の目周辺の鉛直構造の観測に成功した。本実習では、19 日の大型台風を中心とした 5 日間の静止気象衛星ひまわり 8 号のデータと CloudSat/CALIPSO 衛星の雲の鉛直プロファイルのデータを利用し、台風の雲の水平分布と鉛直分布の構造を見る。

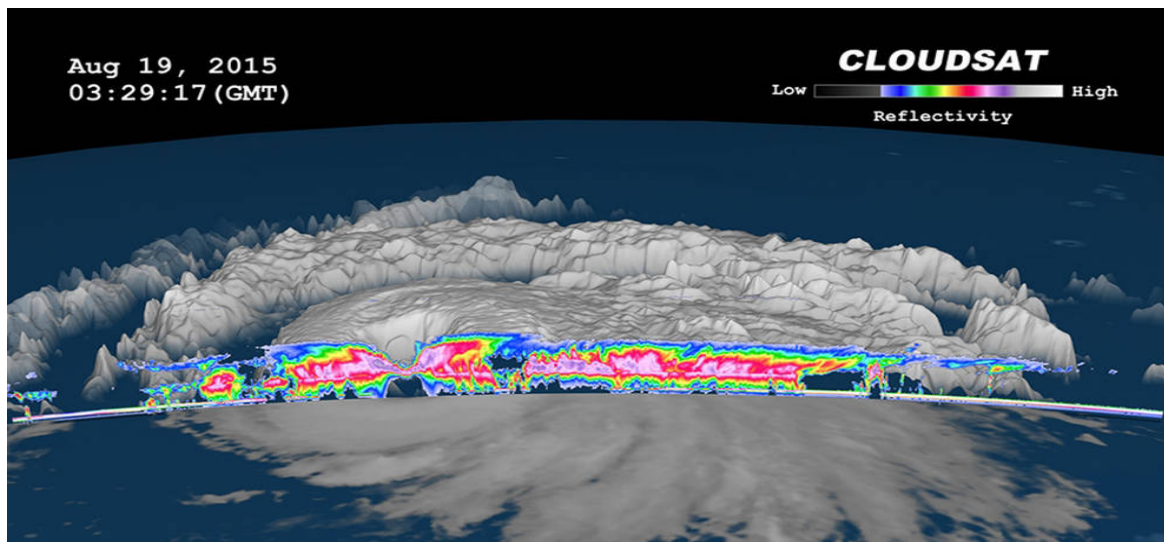


図 1.3 ひまわり静止気象衛星と CloudSat 衛星の観測データを用いたコンポジット画像。赤系色は多量の水と氷を示す。青系色は小さな雲粒子を示す。Credits: JAXA/NASA/Colorado State Univ., Natalie D. Tourville (<https://www.nasa.gov/feature/goddard/17w-northwest-pacific-ocean>)

1.2 目的

本実習の主な目的は、以下の通りである。

- Python の基本的な使い方と Python を用いた衛星データ (HDF, netCDF 形式) の処理方法を学ぶ。
- ひまわり 8 号および CloudSat/CALIPSO のデータ解析を通して、台風の雲の水平・鉛直構造と日変化を観察する。

1.3 使用データ

1.3.1 ひまわり 8 号由来のデータ

ひまわり 8 号搭載の可視赤外放射計 (Advanced Himawari Imager AHI) の 16 バンド (可視 3, 短波長赤外 4, 赤外 9 バンド) のうち、表 1 に示した 8 バンドを用いた Integrated Cloud Analysis System ICAS (Iwabuchi et al., 2016) による雲プロダクトを使用する。ICAS では、多層大気中での多重散乱を考慮した放射伝達モデルを計算に用い、大気データ (気温や湿度など) は MERRA-2 大気再解析プロダクトを、地表面データ (温度と射出率) は MODIS level-3 プロダクト等を用いている。雲特性の推定は、物理モデルを観測データにフィッティングする最適推定法 (Rodgers, 2000) に基づく。ICAS のアルゴリズムの詳細は Iwabuchi et al. (2016; 2018) と Khatri et al. (2018) に記述されている。

表 1. ICAS が用いる AHI バンド

Band number	Central wavelength (μm)	Horizontal resolution (km)
9	6.94	2
10	7.35	2
11	8.59	2
12	9.63	2
13	10.4	2
14	11.2	2
15	12.4	2
16	13.3	2

ICAS プロダクトでは、以下のデータが含まれる：

- 緯度・経度
- インバージョン状態を表すフラグ値 (mstat)
- 雲の巨視的・光学・微物理的特性
 - 雲頂気圧・温度・高度 (cloud-top pressure/temperature/height; CTP/CTT/CTH)
 - 雲の光学的厚さ (cloud optical thickness; COT)
 - 鉛直積算雲水量 (cloud water path; CWP)
 - 雲粒有効半径 (cloud effective particle radius; CER)
 - 雲相の判定結果
- 地表面温度 (surface temperature)
- 単層・複層雲の判定結果

- 推定した雲特性の不確定性を表す診断量（誤差共分散行列や自由度など）

単層雲の場合は $\log(\text{CTP})$, $\log(\text{COT})$, $\log(\text{CER})$, surface temperature が推定される。複数（二層）雲の場合は上層雲の $\log(\text{CTP})$, $\log(\text{COT})$, $\log(\text{CER})$, および下層雲の $\log(\text{CTP})$ が推定される。

上に述べたデータのうち、本実習では表 2 に示す変数のデータを使用する。アッサニー台風のベストトラックを追跡し、2015 年 8 月 17 日から 2015 年 8 月 24 日（5 日間）までの一時間毎のひまわり 8 号のデータを ICAS により解析し、日ごとの netCDF ファイルを準備した。また、CloudSat/CALIPSO 観測時刻に合わせた 2015 年 8 月 19 日の 03:30:00 UTC における 1 グラニュールの netCDF ファイルも準備した。

表 2. 本実習で使用する ICAS データ

Name	Description	Unit	Type	Fill	Comments
latitude	Latitude	degree	float32	-999999.0	Array[1861,2052]
longitude	Longitude	degree	float32	-999999.0	Array[1861,2052]
cot	Cloud optical thickness	-	float32	-999999.0	Array[24,1861,2052]
ctp	Cloud top pressure	hPa	float32	-999999.0	Array[24,1861,2052]
cth	Cloud top height	m	Float32	-999999.0	Array[24,1861,2052]
mts	Inversion status	-	float32	-999999.0	Array[24,1861,2052]

Mstat の値の意味は次のようになっている。

```

mtstat < = -3: Inversion discarded/failed
mstat = -2: Double-layer cloud, non-optimal
mstat T = -1: Single-layer cloud, non-optimal
mstat = 0: Clear sky
mstat = 1: Single-layer cloud, optimal
mstat = 2: Double-layer cloud, optimal

```

1.3.2 CloudSat/CALIPSO 由来のデータ

CloudSat 衛星搭載の 94 GHz の雲レーダー Cloud Profiling Radar (CPR) と CALIPSO 衛星搭載のライダー CALIOP は、およそ 15 秒間隔で同じ軌道を飛行してほぼ直下を観測する。これらの観測データを利用して、DARDAR (raDAR/liDAR) 雲プロダクト (Delanoë and Hogan, 2010) が作成されている。データは

<http://www.icare.univ-lille1.fr/projects/dardar>

からダウンロード可能であり、データの詳細も記述されている。本実習では、DARDAR_MASK および DARDAR_CLOUD プロダクトを用いて、雲の鉛直分布について解析を行う。それぞれのデータファイルには多変数のデータが格納されているが、本実習で使用する変数の一覧を表 3 と表 4 に示す。

表 3. 本実習で使用する DARDAR_MASK プロダクト

Name	Description	Unit	Type	Fill	Comments
CLOUDSAT_ Latitude	Latitude of the collocated product	degrees	float32		Height of range bin above reference surface (~mean sea level)
CLOUDSAT_ Longitude	Longitude of the collocated product	degrees	float32		
CS_Track_ Height	Height of range bin	km	float32		
CLOUDSAT_2B_ GEOPROF_Radar_ Reflectivity	Radar reflectivity at 94GHz	dBZ	int16	-8888	must be divided by 100*100 though the manual says by only 100
CLOUDSAT_2B_ GEOPROF_CPR_ Cloud_Mask	CloudSat cloud mask	-	int8	-9	0: No cloud 1: Likely bad data 5: Ground clutter 5-10: weak detection found using along track integration 20-40: Cloud detected. Increasing values represents clouds with lower chance of being a false detection

参考 : http://www.icare.univlille1.fr/projects/dardar/documentation_dardar_mask

表 4. 本実習で使用する DARDAR_CLOUD プロダクト

Name	Description	Unit	Type	Fill
longitude	Latitude	degree	float32	-999.0
latitude	Longitude	degree	float32	-999.0
height	Height	m	float32	-999.0
effective_radius	Retrieved effective radius	m	float32	-999.0
iwc	Retrieved effective radius	kg m ⁻³	float32	-999.0

参考 : http://www.icare.univlille1.fr/projects/dardar/documentation_dardar_cloud

1.4 実習の流れ

実習は以下の流れで行う。

- ① Python の基本操作を学ぶ。
- ② netCDF 形式データファイルの処理方法を学び，1 グラニュール（CloudSat/CALIPSO に同期したグラニュール）の ICAS データを用いて台風の雲の空間分布（水平分布）を解析する。
- ③ 上記の 1 グラニュールの解析に加え，5 日間の ICAS データを用いて台風の雲の空間分布および日変化を解析する。
- ④ HDF 形式データファイルの処理方法を学び，DARDAR 雲プロダクトを用いて，CloudSat/CALIPSO がアッサニーを通過した際の台風の雲の鉛直分布を解析する。

参考文献

- Delanoë, J., & R. J. Hogan. (2010). Combined CloudSat-CALIPSO-MODIS retrievals of the properties of ice clouds, *Journal of Geophysical Research: Atmospheres*, 115, D00H29. <https://dx.doi.org/10.1029/2009JD012346>.
- Iwabuchi, H., Saito, M., Tokoro, Y., Putri, N. S., & Sekiguchi, M. (2016). Retrieval of radiative and microphysical properties from of clouds from multispectral infrared measurements. *Progress in Earth and Planetary Science*, 3. <https://doi.org/10.1186/s40645-016-0108-3>.
- Iwabuchi, H., Putri, N. S., Saito, M., Tokoro, Y., Sekiguchi, M., Yang, P. & Baum, B. A. (2018). Cloud property retrieval from multiband infrared measurements by Himawari-8. *Journal of Meteorological Society of Japan Ser. II*, 2018-001, <https://doi.org/10.2151/jmsj.2018-001>.
- Khatri, P., Iwabuchi, H., and Saito, M. (2018), Vertical profiles of ice cloud microphysical properties and their impacts on cloud retrieval using thermal infrared measurements, *Journal of Geophysical Research: Atmospheres*, 123, 5301-5319, <https://doi.org/10.1029/2017JD028165>.

2. Python の使い方

2.1 Python の基本要素

2.1.1 変数

```
a = 20
b = 3
c = a*b
print(c)

c = b//a      # 割り算の整数部
print(c)

c = a%b       # 割り算の剰余
print(c)

c = a**b      # べき剰
print(c)
```

2.1.2 リスト

リストとは, []の中をコンマで区切られた要素で構成されている変数である。

変数名= [要素 1, 要素 2, ……]

```
var1 = [0,1,2,3,4]
var2 = [5,6,7,8]
var = var2 + var1
print(var)

var.append(100)      # リストの末尾に新しい要素を追加する
print(var)

n_elements = len(var)      # リストの要素数 (長さ)
print(n_elements)

var.sort()            # リストを小さい順から大きい順に並び替える
print(var[0],var[4],var[-1]) # 特定の要素 (1番目, 5番目, 最後) を抽出する

print(var[1:5])        # 特定範囲の要素 (2番目から4番目まで) を抽出する

print(var[::-1])       # 要素を逆順に並び替える
```

2.1.3 デictionary

Dictionaryとは, キーと値を対応させるような性質を持った変数である。

変数名={キー 1 : 値 1, キー 2 : 値 2, ……}

```
x = {"fruit":"apple", "country":"Japan", "university":"Tohoku"}
x.keys()      # 全てのキーをリストにする

x.values()    # 全ての値をリストにする

x['university'] # キーに対応する値を参照する
```

2.1.4 タプル

タプルは、()の中でコンマで区切られた定数で構成される。リストと同じように使うことができるが、要素の変更ができない。

変数名=(要素 1, 要素 2, ……)

要素が一つだけの時も、最後にコンマが必要。

```
var = (100)           # 数値を表す
var = (100,)          # タプルを表す
var1 = [1,2,3]        # リスト
var2 = (4,5,6)         # タプル
print (tuple(var1))   # リストをタプルに変更
(1, 2, 3)
print (list(var2))     # タプルをリストに変更
[4, 5, 6]
```

2.1.5 if, else

例：与えた西暦年が閏年か平年かを判定する。

```
year = 2000
if (year%4 == 0) and (year%100 != 0) or (year%400 == 0):
    print("Leap year")
    print("year-->",year)
else:
    print("Common year")
    print("Year:",year)
```

Python の場合に複合文 (Compound statement) の一行の終わりにコロン(:)が必須である。インデントには重要な意味があり、インデントはブロックを明示している。改行は文と文の区切りである。

2.1.6 for, while

例：1 から 100 までの奇数をプリントする。

```
#for ループ
for i in range(1,100):    # 任意の回数を繰り返す
    if i%2!=0:
        print(i)

#while ループ
i=1
while i<100:              # 特定の条件を満たすまで繰り返す
    if i%2!=0:
        print(i)
    i=i+1
```

2.1.7 関数

def 文を用いて関数を定義する。例：与えた西暦年が閏年か平年かを判定する。

```
def leap(x):              # 関数名
    if x%4 == 0 and x%100 != 0 or x%400 == 0:
        year = 'leap year'
    else:
        year = 'Normal year'
    return(year)          # returnで値を返す

print(leap(2000))
```

2.1.8 Numpy

Numpy は学術計算用ライブラリ（モジュール）であり，その中に多くの数学関数が用意されている。ここでは，今回の実習で使う関数の使用例を示す。Numpy の詳細は，<http://www.numpy.org> 等に記述されている。

Python でライブラリ・その中の関数・クラスを使う場合，初めにそれらをインポートする必要がある。例えば，

```
import numpy          # numpy をインポートする
import numpy as np    # numpy を np という別名でインポートする
from numpy import array # numpy から array という関数をインポートする
```

① arange() - 連番や等差の数列を生成する

```
import numpy as np
x = np.arange(10)          # 0から9(10-1)まで間隔1の数列
print(x)
x = np.arange(1,5,0.5)     # 1から4.5(5-0.5)まで間隔0.5の数列
print(x)
```

② linspace() - 等間隔な数列を作成する

```
x = np.linspace(10,50,5)  # 10～50の区間を5等分に分けた数列
```

③ array() - リストを配列に変換する

```
x_list = [[1,2,5.8],[4,5,8.1]]          # リスト
x_arr = np.array(x_list, dtype=np.float32) # データ型を指定した配列
x_arr.ndim                               # 配列の次元数
x_arr.shape                               # 配列の形状 (行と列の数)
x_arr.size                                # 配列の要素数
```

④ zeros(), zeros_like() - 全ての要素を0にした配列を作成する

```
size = (2,2)
x = np.zeros(size)
print(x)

y = np.zeros_like(x)  # 配列xと同じ形状で全ての要素を0にした新しい配列
print(y)
```

⑤ reshape() - 配列を変形させる

```
x = np.arange(10)
print(x)
y = x.reshape(2,5)          # 1次元配列xを2次元配列(行数2・列数5)に変換
print(y)
```

⑥ 三角関数

```
x = 3.14
np.sin(x)          # x (rad)のsin値
np.cos(x)          # x (rad)のcos値
```

⑦ ravel - 多次元配列を1次元配列に変換する

```
x = np.array([[1,2,3],[4,5,6]])
print(np.ravel(x))
```

⑧ rot90() - 配列を90度回転させる

```
x = np.arange(10)
y = x.reshape(2,5)
z = np.rot90(y)          # 2次元配列yを90度回転
print(z)
```

⑨ count_nonzero() - 0ではない要素を数える

```
x = np.array([[0,2,3,0,-7,-3],[2,3,0,0,-10,-20]])
cnt_nonzero = np.count_nonzero(x)
print(cnt_nonzero)
```

⑩ where() - 与えた条件を満たす要素のインデックスを返す

```
x = np.linspace(-4,4,8).reshape(4,2)
print(x)

indx = np.where(x<0)      # xが0より小さい要素のインデックスを取得
x[indx] = np.nan          # xが0より小さい要素を無効値(nan)にする
print(x)
```

⑪ isnan() - 要素がNaN (Not a Number)かどうかをbool値(True or False)で返す

```
x = [[1.,2.,-8.,np.nan], [np.nan,2.,-8.,0], [1.,2.,np.nan,np.nan]]
x = np.array(x)
print(x)

print(np.isnan(x))      # 要素がnanであるかをbool値(true or false)で返す

x[np.isnan(x)] = 0      # 要素がnanであれば, 0にする
print(x)
```

⑫ nanmin(), nanmax(), nanmean(), nansum()等 - NaNを無視した処理

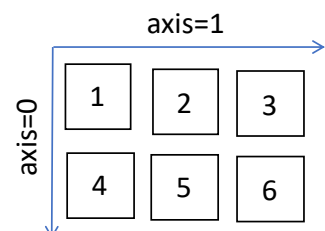
```
x = [np.nan,0,5,-6,10,np.nan,500,50,60,np.nan,3,4]
x = np.array(x)
y = x.reshape(4,3)          # 2次元配列(行数4・列数3)への変換
print(y)
print(np.nansum(y))          # nanを無視した場合の合計値
print(np.nanmean(y))         # nanを無視した場合の平均値
print(np.nanmin(y))          # nanを無視した場合の最小値
print(np.nanmax(y))          # nanを無視した場合の最大値
print(np.nanmin(y, axis=0))   # nanを無視した場合の行方向の最小値 (axisについて以下の説明を見る)
print(np.nanmin(y, axis=1))   # nanを無視した場合の列方向の最小値
```

例えば, 以下の2次元の配列を見よう。

```
a = np.array([[1,2,3],[4,5,6]])
```

a.shape は, (2,3) である。

axis=0 と axis=1 は右図のように行または列方向を示す。



3次元の配列の場合：

```
b = np.array([a,a+10])
```

`b.shape` は、`(2,2,3)`である。

新しく出来た最上位の `axis` が 0 番となる。

`np.sum(b,axis=0)` は、

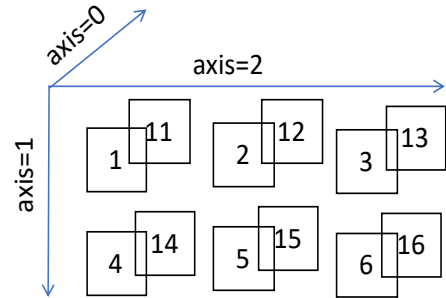
```
[[12, 14, 16],[18, 20, 22]]
```

`np.sum(b,axis=1)` は、

```
[[ 5,  7,  9],[25, 27, 29]]
```

`np.sum(b,axis=2)` は、

```
[[ 6, 15],[36, 45]]
```



⑬ histogram() - 一次元数列の要素を用いてヒストグラムを計算

```
x = np.linspace(-100,100,200)
print(x)
hist,edge = np.histogram(x, bins=5, range=(-80,80))
print(hist)
print(edge)
```

#-----Input-----

#x: 数列名

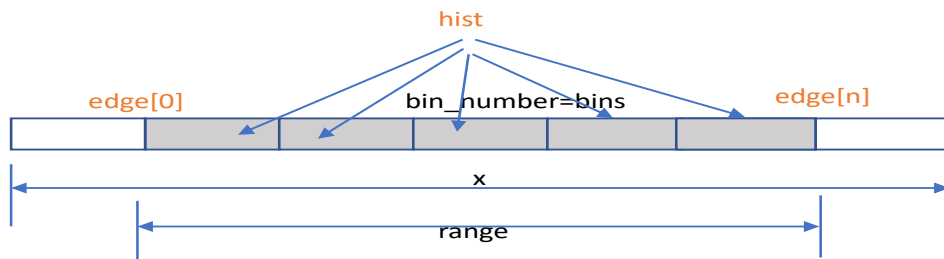
#bins: ヒストグラムを計算するためのビン数

#range: ヒストグラムを計算する値 (xの要素) の範囲

#-----Output-----

#hist: 各ビンに対する頻度分布 (1次元配列)

#edge: 各ビン境界値 (1次元配列)



histogram の概略図。赤字で示しているのは出力である。

⑭ histogram2d() - 二つのデータサンプルを用いて2次元ヒストグラムを計算

```
x = np.linspace(-100,100,200)
y = x*2
hist,yedge,xedge = np.histogram2d(y, x, bins=[10,10], range=[[-80,80],[-80,80]])
print(xedge)
print(yedge)
```

#-----Input-----

#y: 行方向の1次元配列

#x: 列方向の1次元配列

#bins=(ny,nx): y (行方向) と x (列方向) に対するヒストグラムのビン数

#range=[[ymin,ymax],[xmin,xmax]]: y (行方向) と x (列方向) に対するそれぞれの最小値と最大値

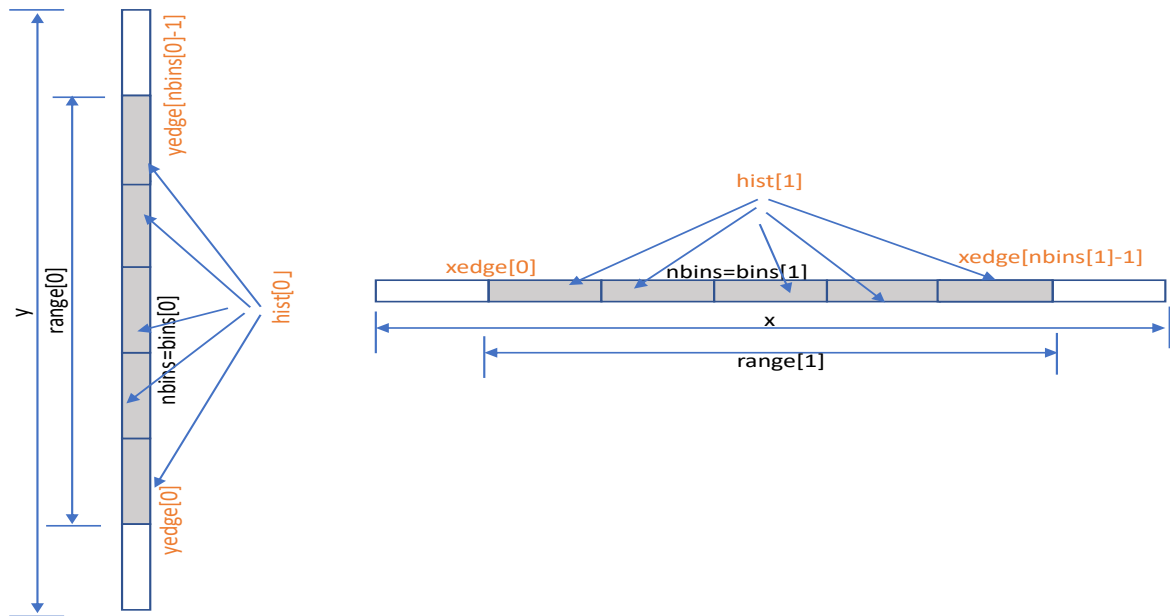
その範囲に当てはまるデータのヒストグラムが計算される。

#-----Output-----

#hist: yとxに対する2次元ヒストグラム (2次元配列)

#yedge: y (行方向) のビンの境界値 (タプル)

#xedge: x (列方向) のビンの境界値 (タプル)

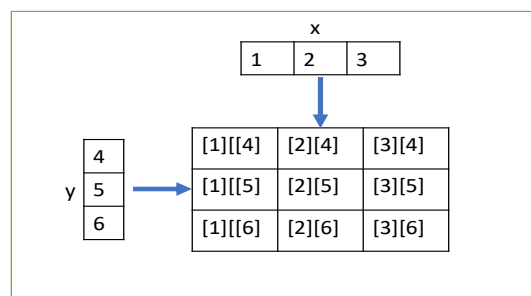


histogram2d の概略図。赤字で示しているのは出力である。

⑮ meshgrid() - 格子状の座標を表す配列を作成する

```
x = np.array([1,2,3])
y = np.array([4,5,6])

xx,yy = np.meshgrid(x,y)
xx
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
yy
array([[4, 4, 4],
       [5, 5, 5],
       [6, 6, 6]])
```



⑯ roll - 配列の要素を循環移動する

```
a = [1,2,3,4,5,6,7,8]
b = np.roll(a,-3)
print(b)
[4 5 6 7 8 1 2 3]
c = np.roll(a,3)
print(c)
[6 7 8 1 2 3 4 5]
```

⑪ savetxt(), loadtxt()

savetxt - dat, csv, txt 形式のファイルに書き換える

loadtxt - dat, csv, txt 形式のファイルを読み込む

```
x = np.array(np.arange(10))
x = x.reshape(5,2)
print(x)

filename = 'test.txt'
np.savetxt(filename, x*2, fmt='%.2f', header='This is a test file')

y = np.loadtxt(filename, skiprows=1)
print(y)
# filename : ファイル名
# x         : ファイルに書き込む変数
# fmt       : ファイルに書き込む形式
# header    : ヘッダー
# y         : 読み込んだデータを保存する変数
# skiprows  : 読み込む際、何行とばすかを指定
```

ここで、`fmt='%.2f'`は、小数点以下2桁の浮動小数点数値という意味である。ファイル `test.txt` ができていることを確認しよう。

2.2 データのプロット

Python でデータをプロットする際、`matplotlib.pyplot` というモジュールが必要。

```
import numpy as np
import matplotlib.pyplot as plt
```

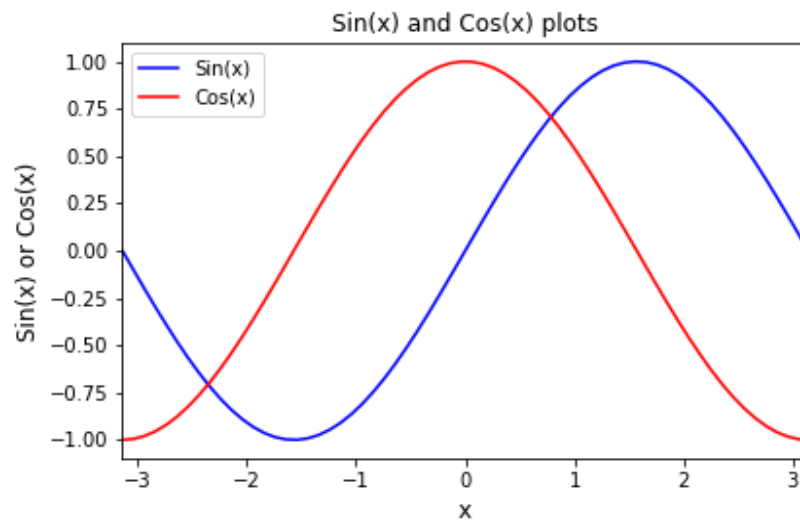
2.1.1 一次元データのプロット

本実習では `plot` を用いて一次元データをプロットする。以下 `plot` を用いた一次元データのプロット方法の例を示す。

例：sin 波と cos 波をサンプリング間隔 0.1、範囲 $[-\pi, \pi]$ について描画する。

```
delta = 0.1 # サンプリング間隔
x = np.arange(-np.pi, np.pi+delta, delta) # xの値
fig = plt.figure() # 何も描画されていない新しいウィンドウを開く
fig.add_subplot(1,1,1)
plt.xlabel('x', fontsize=12)
plt.ylabel('Sin(x) or Cos(x)', fontsize=12)
plt.plot(x, np.sin(x), color='blue', label='Sin(x)')
plt.plot(x, np.cos(x), color='red', label='Cos(x)')
plt.xlim(-np.pi, np.pi)
plt.title('Sin(x) and Cos(x) plots')
plt.legend(loc='best') # loc='best'にすると最適な場所にlegendが置かれる
plt.tight_layout() # グラフの位置やサイズを自動で調整する
plt.savefig('Sin_and_cos.png')
plt.show()
```


`add_subplot(a,b,c)` : aは行数, bは列数, cはどの位置にプロットするかを示す。
`plt.plot(x,y)` : x と y の 1次元プロットを行う。



2.1.2 二次元データのプロット

本実習では `pcolormesh` と `contourf` を用いて二次元データをプロットする。以下に `pcolormesh` と `contourf` を用いた 2次元データのプロット方法を示す。

例 : x と y の範囲が $-1 \leq x \leq 1$ と $-1 \leq y \leq 1$ である時の $z = x^2 + y^2$ の 2次元プロットを作成する。

```
x = np.arange(-1, 1+0.1, 0.1)
y = np.arange(-1, 1+0.1, 0.1)
xx,yy = np.meshgrid(x,y)          # Numpyのmeshgridの例を参照する
z = xx**2 + yy**2                 # z=x^2+y^2
fig = plt.figure(figsize=(5,4))   # サイズを指定し, ウィンドウを開く
ax = fig.add_subplot(1,1,1)
mesh = ax.pcolormesh(xx, yy, z, cmap=plt.get_cmap('jet'))
plt.xlim(-1,1)
plt.ylim(-1,1)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('z=$x^2+y^2$ (Grid plot)')
plt.xticks(np.arange(-1,1+0.1, 0.4)) # 表示するx軸の目盛り
plt.yticks(np.arange(-1,1+0.1, 0.4)) # 表示するy軸の目盛り
fig.colorbar(mesh)
plt.tight_layout()
plt.savefig('pcolormesh.png')
plt.show()
```

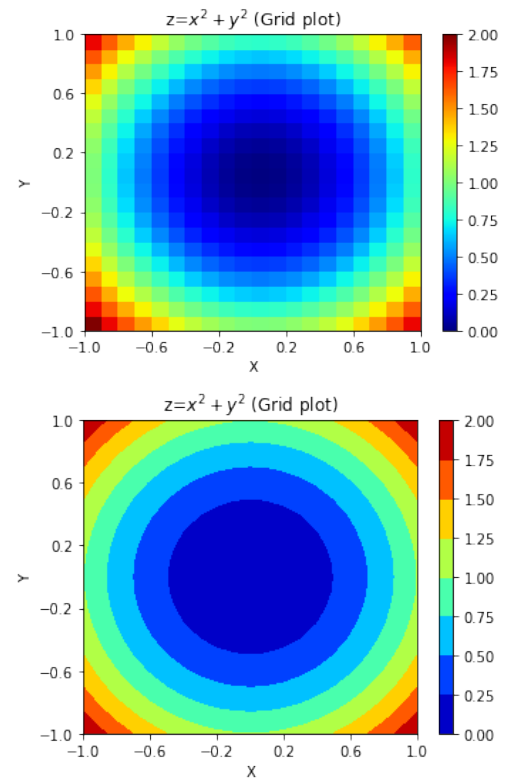
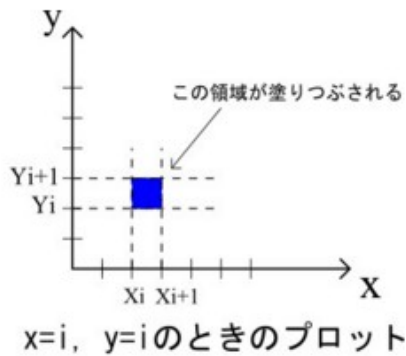
`pcolormesh(x, y, z)`: z の値をマッピングする。

$x \rightarrow$ グリッドの x 軸

$y \rightarrow$ グリッドの y 軸

$z \rightarrow$ グリッド (x, y) の値

`plt.get_cmap()`: カラーマップを取得する。



等高線図プロットの場合、`pcolormesh` の代わりに `contourf` 関数を用いる。

```
mesh = ax.contourf(x,y,z,cmap=plt.get_cmap('jet'))
```

2.3 データの補間

Python では、Scipy に様々な補間関数が用意されており、補間曲線を得るための幾つかの方法がある。ここでは Scipy を用いた線形補間とスプライン補間を紹介する。

例: $[-\pi, \pi]$ 範囲の 10 間隔でのサンプリングデータを 1000 等間隔の点に補間し、元のサンプリングデータと補間したデータに対する \cos 波をプロットする。

① 必要なモジュールをインポートする

```
import numpy as np
import scipy.interpolate as interp
import matplotlib.pyplot as plt
```

② 元のサンプリング間隔と補間後のサンプリング間隔を指定する。

```
x_org = np.linspace(-np.pi, np.pi, 10)
x_new = np.linspace(-np.pi+0.0001, np.pi-0.0001, 1000)
```

上記では、端の点が $[-\pi, \pi]$ の範囲より、少しだけ内側になるように細工をした。

③ データを補間する。以下に三つの補間方法を示す。

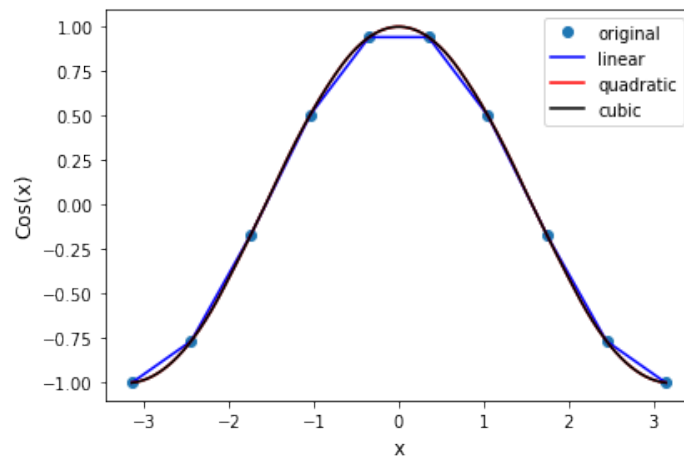
```
# 線形補間
f_lin = interp.interpld(x_org, np.cos(x_org), kind='linear')
# サンプリングデータより計算された関数
cos_lin = f_lin(x_new) # 得られた関数を用いた補間
print(cos_lin)

# 2 次スプライン補間
f_quad = interp.interpld(x_org, np.cos(x_org), kind='quadratic')
cos_quad = f_quad(x_new)

# 3 次スプライン補間
f_cub = interp.interpld(x_org, np.cos(x_org), kind='cubic')
cos_cub = f_cub(x_new)
```

④ 元データと補間後のデータをプロットする。(一次元データのプロットの例を参照)

```
fig = plt.figure()
fig.add_subplot(1,1,1)
plt.xlabel('x', fontsize=12)
plt.ylabel('Cos(x)', fontsize=12)
plt.plot(x_org, np.cos(x_org), 'o', label='original')
plt.plot(x_new, cos_lin, '-', color='blue', label='linear')
plt.plot(x_new, cos_quad, '-', color='red', label='quadratic')
plt.plot(x_new, cos_cub, '-', color='black', label='cubic')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```



3. 衛星データ解析

3.1 NetCDF ファイルの扱いとデータの可視化

本節では、netCDF ファイルの扱い方とデータの可視化方法を学び、台風の雲の空間分布を図示する。2015 年 8 月 19 日の 3:30 UTC における ICAS AHI プロダクト (CloudSat/CALISPO の通過に同期したグラニュール) を対象にして以下の実習を行う。

実習 3.1 2015 年 8 月 19 日の 3:30 UTC における ICAS プロダクト (AH18_L2STI.I2015231.033000.v210.nc) を用いて, high cloud (cloud top pressure; CTP < 300 hPa) の cloud optical thickness (COT) と cloud top height (CTH) の空間分布を図示する。

まず、取得済みデータがあるディレクトリに移動して、Jupyter Notebook で作業する。以下では、netCDF 形式のデータファイルから対象のデータセットを読み込む。

- ① netCDF4 モジュールの Dataset クラスをインポートし、ファイルの中身を任意変数 (ここでは f) に格納する。

```
import numpy as np
from netCDF4 import Dataset as dt
file = 'AH18_L2STI.I2015231.033000.v210.nc'
f = dt(file, 'r')
```

- ② f 内に格納されたデータの名前, サイズ, 属性 (add_offset, scale_factor 等) を表示する。

```
f.variables  
OrderedDict([('ctp', <class 'netCDF4._netCDF4.Variable'>  
float32 ctp(ydim, xdim)  
scale_factor: 1  
add_offset: 0  
units: hPa  
longname: cloud top pressure  
unlimited dimensions:  
current shape = (1861, 2052) ← 配列のサイズ  
filling on, default _FillValue of 9.969209968386869e+36 used),  
:  
:
```

データセット名

} 属性

- ③ 対象の変数（ここでは `cot`）のデータを取り出して、任意の別名の変数に格納する。

```
cot = f.variables['cot']
```

- ④ データは整数値 (Scaled Integer SI) または物理量 (PV) として格納される。データセットの属性を確認すれば格納されたデータが SI か PV か分かる。データセットの属性に `scale_factor` と `add_offset` があれば、それらを用いて SI を以下のように物理量 (PV) へ変換する。

$$PV = SI \times scale_factor + add_offset$$

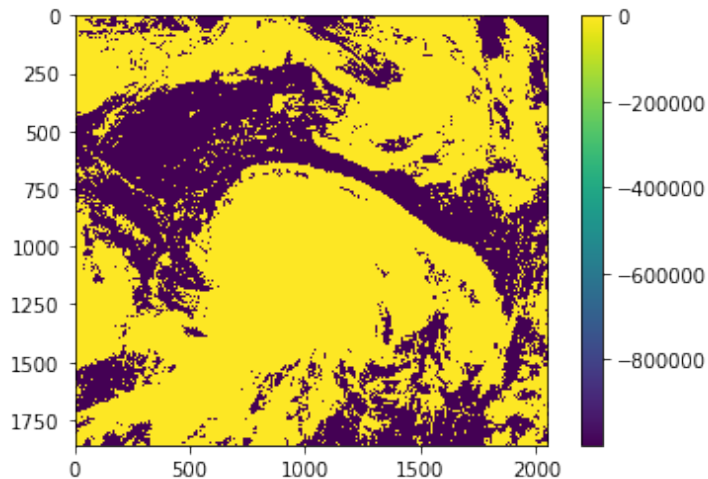
上記の変数 cot に対する scale_factor と add_offset を読み込み、SI を PV へ変換する。またその PV を配列にする（配列にすると後処理の際、便利）。

```
scale = getattr(cot, 'scale_factor') # getattrを用いてaの属性を読みこむ
offset = getattr(cot, 'add_offset')
cot = cot*scale + offset
cot = np.array(cot)
```

⑤ 読み込んだデータを画像にする場合、matplotlib.pyplot モジュールが必要。

画像を表示する関数 imshow を用いて 2 次元配列データを画像として表示する。

```
import matplotlib.pyplot as plt
plt.imshow(cot) #cotの値を画像として表示する
plt.colorbar() #カラーバーをつける
plt.show()
```

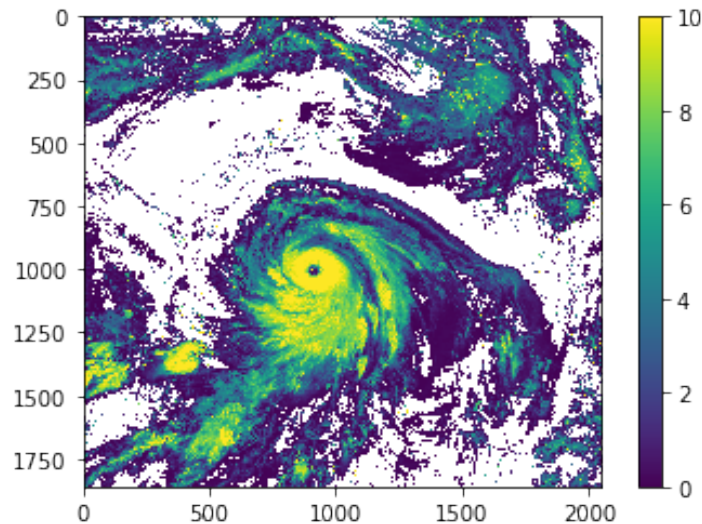


これまでのデータはデータ欠損を示す fill_value 等の非物理値を含めている。このような非物理値を NaN (Not a Number) にして再び画像化する。本実習のために用意したデータファイルでは、fill_value = -999999.0 である（表 2 を参照）。

```
plt.clf() # 上記の画像の表示を取り消す
fv = -999999.0
cot[cot==fv] = np.nan # COT = -999999.0を NaN にする
plt.imshow(cot)
plt.colorbar()
```

データの値の範囲を指定してプロットする場合

```
plt.imshow(cot, vmin=0, vmax=10) # 色付けする範囲を0(vmin)から10(vmax)に指定
plt.colorbar()
```



上記で述べた手順を以下のような関数 `read_netcdf` にまとめておくと、多数のデータセットを読み込む際に便利である。また、前述の2次元データのプロット方法を参考にし、読み込んだデータの空間分布を描くため、関数 `image_plot` を作っておく。以降の実習では、これらの関数を利用する。

function:read_netcdf

```
def read_netcdf(filename,varname):
    """
    Read netCDF file

    Input
        filename : netCDF file name
        varname   : variable name
    Output
        v, p      : variable data
    """
    f = dt(filename,'r')
    v = f.variables[varname]
    if (varname == 'longitude' or varname == 'latitude'):
        v = np.array(v[:], dtype=np.float32)
        return v
    else:
        sc,of=getattr(v,'scale_factor'),getattr(v,'add_offset')
        p = sc*v-of
        p = np.array(p, dtype=np.float32)
        return p
```

注) 本実習で使う netCDF ファイルの latitude と longitude には、scale_factor と add_offset の情報が格納されていないため、このコードでは場合分けして処理している。

function: image_plot

```
def image_plot(x,y,z,xlabel,ylabel,cbrng,cbtit,figtit,figsiz,ofile):
    """
    Make image of 2D array

    Input
        x      : 1D array for x-axis
        y      : 1D array for y-axis
        z      : 2D array data
        xlabel : x-axis label name
        ylabel : y-axis label name
        cbrng  : [min,max] for coloring
        cbtit  : color bar title
        figtit : title of this image
        figsiz : size of this image
        ofile  : output file name

    Output
        image and output file of image
    """
    fig = plt.figure(figsize = figsiz)
    ax = fig.add_subplot(1, 1, 1)
    CF = plt.pcolormesh(x, y, z, vmin=cbrng[0], vmax=cbrng[1],
cmap=plt.get_cmap('jet'))
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(figtit)
    plt.colorbar().ax.set_title(cbtit)
    plt.tight_layout( )
    plt.savefig(ofile)
    plt.show()
```

- ① まず、functions.py の中に含まれている関数群を import する。read_netcdf 関数を用いて、netCDF ファイルから cot, cth, lat と lon を読み込む。lat と lon は座標を指定するために使用する。

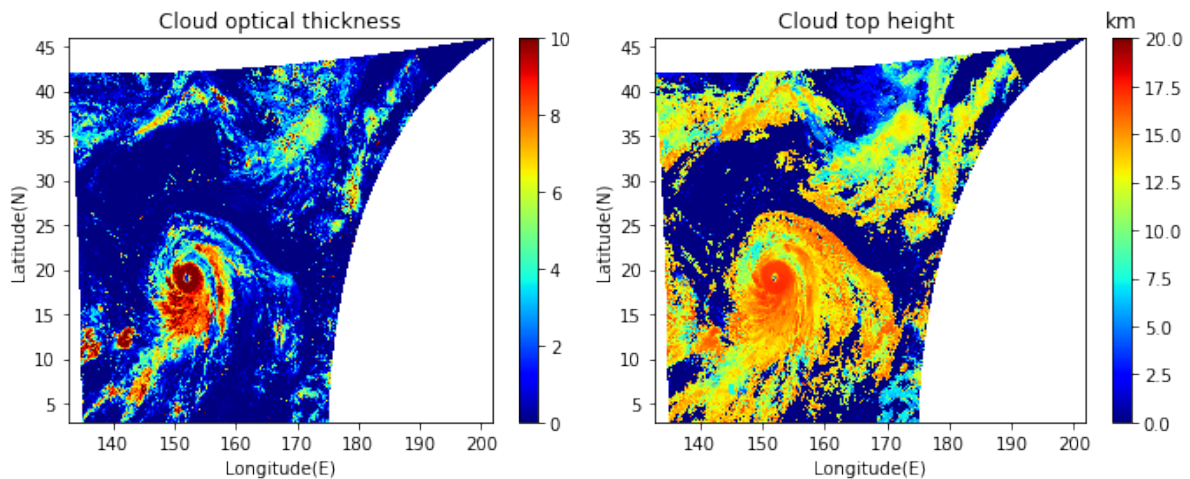
```
from functions import *
file = 'AHI8_L2STI.I2015231.033000.v210.nc'
cot = read_netcdf(file,'cot')
cth = read_netcdf(file,'cth')
lon = read_netcdf(file,'longitude')
lat = read_netcdf(file,'latitude')
```

- ② データをプロットする前に単位を確認するとよい。

```
from netCDF4 import Dataset as dt
f = dt(file,'r')
unit = getattr(f.variables['cth'],'units')
print (unit)
m # cthの単位はm であることがわかる
```


③ cot と cth をプロットする。この処理は時間がかかる(10 秒程度)。

```
xlabel = 'Longitude(E)'\nylabel = 'Latitude(N)'\nfigsiz = (5,4)\nlon[lon<0] += 360.\n\n#cot\nfigtit = 'Cloud optical thickness'\ncbrng = [0,10]\ncbtit = ''\nofile = 'cot.png'\np = image_plot(lon,lat,cot,xlabel,ylabel,cbrng,cbtit,figtit,figsiz,ofile)\n\n#cth\nfigtit = 'Cloud top height '\ncbrng = [0,20]\ncbtit = 'km'\nofile = 'cth.png'\np = image_plot(lon,lat,cth/1000.,xlabel,ylabel,cbrng,cbtit,figtit,figsiz,ofile)
```



3.2 台風の雲の空間分布と日変化の解析

前節では 1 グラニュールのデータを用いて雲の光学的厚さ (COT) と高さ (CTH) の空間分布を描画したが、この節では以下の手順で台風の雲の空間分布と日変化の解析を行う。

- A) 台風の雲の 6 時間毎のベストトラックデータから、1 時間毎の台風の中心位置を時間的に内挿して求める。
- B) 1 つのグラニュールの ICAS データを用いて、雲タイプ識別を行い、台風の中心からの距離とタイプ別雲量の関係を解析する。
- C) 5 日間(120 時間)のデータを用いて、以下の解析を行う。
 - (ア) 1 時間毎 (合計 120 時間) のタイプ別雲量を計算し、雲量の時空間コンポジット図を作る。横軸は台風の中心からの距離、縦軸は現地時間 (日) とする。
 - (イ) 上記(ア)の結果を現地時間毎に平均し、横軸は台風の中心からの距離、縦軸は現地時間 0-24 時としたコンポジット図を作る。

- (ウ) 上記(イ)の結果をさらに平均し、5日間の平均として台風の中心からの距離対する雲タイプ別雲量の分布を求める（横軸は台風の中心からの距離、縦軸はタイプ別雲量）。

実習 3.2

- A) 台風の雲の6時間毎のベストトラックデータから、1時間毎の台風の中心位置を時間的に内挿して求める

- ① 必要なモジュールをインポートし、データファイルの中身を読み込む。

```
import numpy as np
import scipy.interpolate as interp # データを補間するクラス（節2.3を参照する）
import matplotlib.pyplot as plt
from datetime import date
from functions import *

data = np.loadtxt('bt_original.txt', skiprows=1) # txtファイルを読み込む
year = data[:,0]
month = data[:,1]
day = data[:,2]
hour = data[:,3]
lat = data[:,4]
lon = data[:,5]
slp = data[:,6]
```

- ② 補間前後の日時データのための配列を作る。

```
nsample = lat.size # 元データの要素数
ndays = int(nsample / 4) # 合計日数の計算
nhour = 24
sample_org = np.arange(nsample)
sample_new = np.linspace(0+0.0001, nsample-1-0.0001, nhour*ndays)
daynum_org = np.zeros(nsample)
daynum_new = np.zeros(nhour*ndays)
```

- ③ 1月1日からの日数の計算を行う。

```
for i in range(nsample):
    daynum = date(int(year[i]), int(month[i]), int(day[i]))
    daynum = daynum.timetuple().tm_yday
    daynum = daynum + hour[i] / 24.
    daynum_org[i] = daynum
```

- ④ 3次スプライン補間を用いて経度と緯度を1時間ごとに補間する。（節2.3を参照）

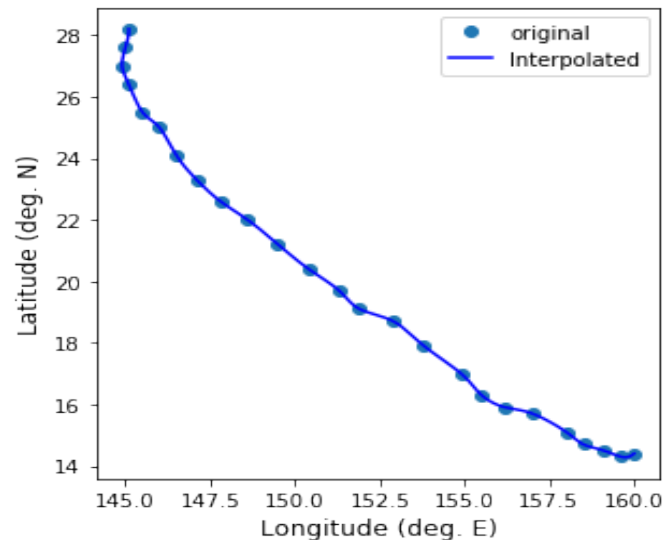
```
f_daynum = interp.interp1d(sample_org, daynum_org, kind='cubic')
daynum_new = f_daynum(sample_new)

f_lon = interp.interp1d(daynum_org, lon, kind='cubic')
lon_new = f_lon(daynum_new)

f_lat = interp.interp1d(daynum_org, lat, kind='cubic')
lat_new = f_lat(daynum_new)
```

⑤ 元のデータと補間したデータをプロットする（節 2.3 を参照）。

```
Fig = plt.figure(figsize=(4.5,4.5))
fig.add_subplot(1,1,1)
plt.xlabel('Longitude (deg. E)', fontsize=12)
plt.ylabel('Latitude (deg. N)', fontsize=12)
plt.plot(lon, lat, 'o', label='original')
plt.plot(lon_new, lat_new, '-', color='blue', label='Interpolated')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```



B) 1つのグラニュールの ICAS データを用いて、雲タイプ識別を行い、台風の中心からの距離とタイプ別雲量の関係を解析する

International Satellite Cloud Climatology Project (ISCCP)による雲の分類は図 3.2 の通りである。

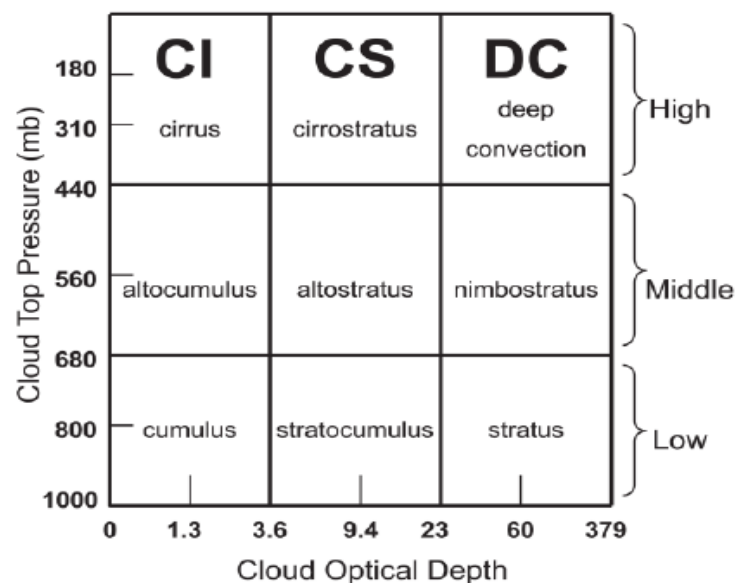


図 3.2 ISCCP 雲分類 (<https://isccp.giss.nasa.gov/cloudtypes.html>)

ICAS では赤外波長のみを用いて COT を推定しているため、COT = 8 以上の深い雲の下部には感度がなく、雲の上部の光学的厚さを求めていると考えられる。推定される COT は 8 以上になることは少なく、最大でもおよそ 30 以下となっている。また、雲が多層に重なっている場合は、ICAS では最上層の雲の光学的厚さを推定している。よって ICAS で推定された COT を用いて雲分類を行う場合は ISCCP とは異なる閾値を用いる。本実習では、Ci・Cs・DC 雲の分類は以下の通りとする。

- Deep Convective cloud (DC) : CTP < 300 hPa & 6 < COT
- Cirrostratus cloud (Cs) : CTP < 300 hPa & 1 < COT ≤ 6
- Cirrus cloud (Ci) : CTP < 300 hPa & 0 < COT ≤ 1

前節の「台風の雲の空間分布の解析(Part I)」の read_netcdf 関数を基に、以下の通り必要なデータセット (cot, ctp, mstat, lat, lon) を一括で読み取れる関数を作っておくと、今後のデータ解析において便利である。この関数も functions.py に含まれている。

function: read_ahi

```
def read_ahi(filename):
    """
    Read ICAS-AHI data

    Input
        filename : ICAS-AHI netCDF data file name
    Output
        cot      : COT data
        ctp      : CTP data
        mstat    : MSTAT data
        lat      : latitude data
        lon      : longitude data
    """
    nt = dt(filename, 'r')
    cot = np.array(nt.variables['cot'][:, :], dtype=np.float32)
    ctp = np.array(nt.variables['ctp'][:, :], dtype=np.float32)
    mstat = np.array(nt.variables['mstat'][:, :], dtype=np.float32)
    lat = np.array(nt.variables['latitude'][:, :], dtype=np.float32)
    lon = np.array(nt.variables['longitude'][:, :], dtype=np.float32)
    cot[np.where(cot<0)] = np.nan
    ctp[np.where(ctp<0)] = np.nan
    return cot, ctp, mstat, lat, lon
```

① functions.py 内の read_ahi 関数を用いてデータを読み込み、雲タイプの判別を行う。

```
file = 'AHI8_L2STI.I2015231.033000.v210.nc'
cot, ctp, mstat, lat, lon = read_ahi(file)

ctype = np.zeros(cot.shape) # cloud typeのインデックスを保存するための配列

dc_indx = (abs(mstat) >= 1) & (ctp < 300.) & (cot > 6.) # DC
cs_indx = (abs(mstat) >= 1) & (ctp < 300.) & (cot > 1.) & (cot <= 6.) # Cs
ci_indx = (abs(mstat) >= 1) & (ctp < 300.) & (cot > 0.) & (cot <= 1.) # Ci

ctype[np.where(dc_indx)] = 3 #DC のpixelに3を代入
ctype[np.where(cs_indx)] = 2 #Cs のpixelに2を代入
ctype[np.where(ci_indx)] = 1 #Ci のpixelに1を代入
```

- ② 雲タイプを3色の離散的なカラーバーを用いて描く。そのため、上記の「2次元データのプロット方法」に基づいて以下の関数を作る。

function: image_plot_discrete

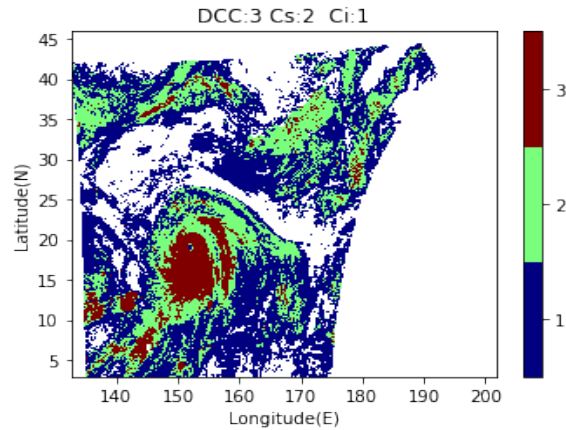
```
def image_plot_discrete(x,y,z,xlabel,ylabel,cbrng,nbin,title,figsize,ofile):
    """
    Make image of 2D array (discrete coloring)

    Input
    x      : 1D array for x-axis
    y      : 1D array for y-axis
    z      : 2D array data
    xlabel : x-axis label name
    ylabel : y-axis label name
    cbrng  : [min,max] for coloring
    nbin   : bin number for coloring
    figtit : title of this image
    figsiz : size of this image
    ofile  : output file name

    Output
    image and output file of image
    """
    fig = plt.figure(figsize=figsize)
    ax = fig.add_subplot(1, 1, 1)
    cticks = np.arange(nbin)+1
    bounds = np.linspace(cbrng[0],cbrng[1],nbin)
    norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
    CF = plt.pcolormesh(x, y, z, cmap=plt.get_cmap('jet'), norm=norm)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.colorbar(CF, ticks=cticks)
    plt.tight_layout()
    plt.savefig(ofile)
    plt.show()
```

- ③ functions.py 内の image_plot_discrete 関数を用いて、雲タイプの空間分布図を示す。

```
ctype[np.where(ctype <=0) ] = np.nan # 非物理値をNaNに置き換える
lon[lon<0]+=360                      # 経度の値を[-180,180]から[0,360]に変更する
title = 'DCC:3 Cs:2 Ci:1'
ofile = 'cloud_type.png'
xlabel = 'Longitude(E)'
ylabel = 'Latitude(N)'
vrange = [0.5,3.5]
nbin = 4
figsize = (4,3)
p = image_plot_discrete(lon,lat,ctype,xlabel,ylabel,vrange,nbin,title,figsize,ofile)
```



次に、台風を中心からの距離に対してタイプ別雲量を計算する。そのため以下の関数を作っておくと便利である。これらの関数も `functions.py` 内に含まれている。

- 関数 `best_track`: 対象データファイルの観測時刻に対する台風の中心位置の緯度・経度を計算する（上記のベストトラックデータの内挿方法を以下に整理したもの）
- 関数 `calc_dist_ahipxl`: 台風の中心位置と各ピクセル間の距離を計算する

function: `best_track`

```
def best_track(i_dayno):
    """
    Calculate the center of the typhoon corresponding to day number

    Input
        i_dayno : day number (JST)
    Output
        lat_new : latitude corresponding to day number
        lon_new : longitude corresponding to day number
    """
    data = np.loadtxt('bt_original.txt', skiprows=1)
    year = data[:,0]
    month = data[:,1]
    day = data[:,2]
    hour = data[:,3]
    lat = data[:,4]
    lon = data[:,5]
    slp = data[:,6]

    nsample = lat.size
    ndays,nhour = int(nsample / 4), 24
    sample_org = np.arange(nsample)
    sample_new = np.linspace(0,nsample-1,nhour*ndays)
    daynum_org = np.zeros(nsample)
    daynum_new = np.zeros(nhour*ndays)

    for i in range(nsample):
        daynum_org[i] =
(date(int(year[i]),int(month[i]),int(day[i]))).timetuple( ).tm_yday \
    + hour[i] / 24.

    f_daynum = interp.interp1d(sample_org,daynum_org,kind='cubic')
    daynum_new = f_daynum(sample_new)
    f_lon = interp.interp1d(daynum_org,lon,kind='cubic')
    f_lat = interp.interp1d(daynum_org,lat,kind='cubic')
    lon_new,lat_new = f_lon(i_dayno),f_lat(i_dayno)
    return lat_new,lon_new
```

地点 A (経度 λ_1 , 緯度 ϕ_1) と地点 B (経度 λ_2 , 緯度 ϕ_2) 間の距離 (d) の計算は以下の式を用いて行う。

$$a = \sin^2((\phi_2 - \phi_1)/2) + \cos \phi_1 \cos \phi_2 \sin^2((\lambda_2 - \lambda_1)/2)$$

$$c = 2 \operatorname{atan2}(\sqrt{a}, \sqrt{1-a})$$

$$d = R \cdot c$$

ここで, R は地球の半径 (6371 km) である。

function: calc_dist_ahipxl

```
def calc_dist_ahipxl(tlon, tlat, alon, alat):
    """
    Calculate the distance between the center of the typhoon and pixel

    Input
        tlon : longitude (the center of the typhoon)
        tlat : latitude (the center of the typhoon)
        alon : longitude (pixel)
        alat : latitude (pixel)
    Output
        d : distance between the center of the typhoon and pixel
    """
    re = 6371
    tlon_rad = np.radians(tlon)
    tlat_rad = np.radians(tlat)
    alon_rad = np.radians(alon)
    alat_rad = np.radians(alat)
    dd = np.sin((alat_rad-tlat_rad)/2)*np.sin((alat_rad-tlat_rad)/2) \
        + np.cos(tlat_rad)*np.cos(alat_rad)*np.sin((alon_rad-tlon_rad)/2) \
        * np.sin((alon_rad-tlon_rad)/2)
    d = re * 2.*np.arctan2(np.sqrt(dd), np.sqrt(1-dd))
    return d
```

④ 台風を中心位置を元め, 中心から各ピクセルまでの距離を計算する。

```
dayno = 231 + (3 + 30/60)/24 + 9/24
alat, alon = best_track(dayno) # daynoに対応する台風の中心の緯度・経度
dis = calc_dist_ahipxl(alon, alat, lon, lat) # 中心から各ピクセルまでの距離 (km)
```

⑤ 台風を中心から 1500 km までの領域を 10 km 毎のビンに分け, 各ビンについて雲タイプ毎のピクセル数を計数する。計数には 2 次元ヒストグラムを求める numpy の histogram2d 関数を用いる。最後に各ビン毎の全ピクセル数で割って, 雲タイプ別の雲量を求める。

```
ntype = 3 # 雲タイプの個数
disrng = [0, 1500] # 距離の範囲 (km)
dis_delta = 10 # ビンの大きさ (km)
ndbin = int((disrng[1]-disrng[0])/dis_delta) # ビンの数
typrng = [0.5, 3.5] # 雲タイプを表す値が1-3なので, ヒストグラムの範囲を0.5-3.5に設定

c_count = np.zeros((ndbin, ntype)) # 各ビンにおける各雲タイプの画素数を保存するための配列
p_count = np.zeros((ndbin)) # 各ビンにおける全ての画素数を保存するための配列
```



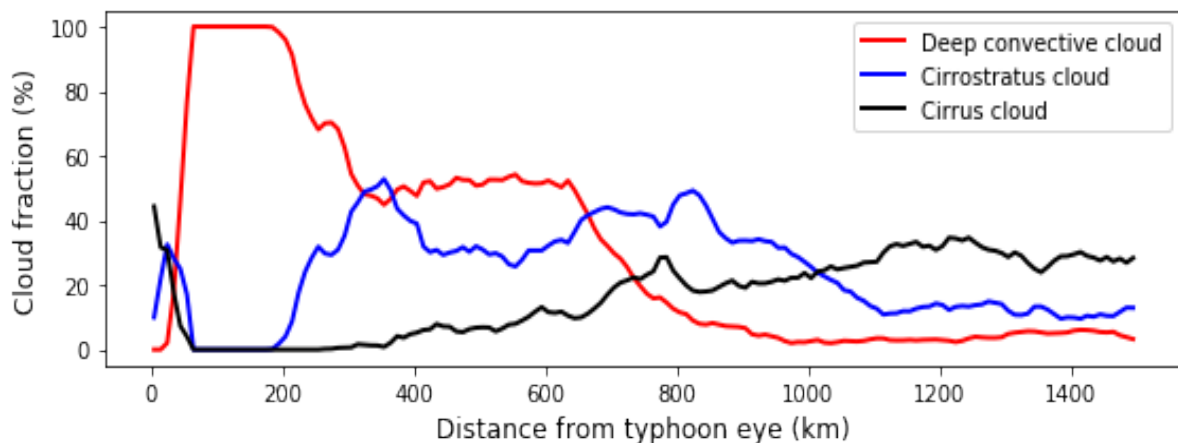
```
#numpyのhistogram2dを用いて指定した範囲内の2次元頻度分布を計算する(2章を参照)
c_count, edge1, edge2 = np.histogram2d(np.ravel(dis),\
                                         np.ravel(ctype), bins = (ndbin, ntype), \
                                         range=(disrng, typrng))

#numpyのhistogramを用いて指定した範囲内の1次元頻度分布を計算する
p_count, edge = np.histogram(np.ravel(dis), bins=ndbin, range=disrng)

#タイプ別の雲量の計算
dc = c_count[:,2] / p_count[:] * 100
cs = c_count[:,1] / p_count[:] * 100
ci = c_count[:,0] / p_count[:] * 100
```

⑥ 上記で求めた雲量をプロットする。

```
x_val = dis_delta/2 + np.arange(ndbin) * dis_delta # xの値
fig = plt.figure(figsize=(8,3))
ax = fig.add_subplot(1,1,1)
plt.xlabel('Distance from typhoon eye (km)', fontsize=12)
plt.ylabel('Cloud fraction (%)', fontsize=12)
plt.plot(x_val, dc, color='red', label='Deep convective cloud', \
         linestyle='solid', linewidth=2)
plt.plot(x_val, cs, color='blue', label='Cirrostratus cloud', \
         linestyle='solid', linewidth=2)
plt.plot(x_val, ci, color='black', label='Cirrus cloud', \
         linestyle='solid', linewidth=2)
plt.legend(loc='best')
plt.tight_layout()
plt.savefig('one_granule_cloudfraction.png')
plt.show()
```



C) 5日間(120時間)のデータを用いて解析を行う

(ア) 2015年8月17日から21日(5日)までの1時間毎(合計120時間)のデータを用いて、横軸を台風の中心からの距離、縦軸を5日間の現地時間としてコンポジット図を作成する。

上記(B)にて1グラニューールのみのデータを解析したが、ここでは連続した5日間の1時間毎のデータを読み込んで処理を行う。

① 解析に用いるデータファイルを準備する。

```
import glob
files = sorted(glob.glob('.*daily.v210.nc'))
```

ここでは、glob モジュールの glob クラスを利用して、指定のデータフォルダにある *daily.v210.nc データファイルのリストを files という変数に格納した。

② 解析条件などを定義する。

```
nday,nhour = 5,24 # 合計日数と1日の時間数
day_st,day_ed = 229,233 # 始日と終日の day number
ntype = 3 # (0) Ci, (1) Cs, (2) DC
disrng = [0,1500] # 距離の範囲 (km)
typrng = [0.5,3.5] # 雲タイプの値に対するヒストグラムの範囲
dis_delta = 10 # ビンの幅 (km)
ndbin = int( (disrng[1]-disrng[0])/dis_delta ) # ビンの数
print(ndbin)
```

③ 毎時における雲タイプ別の雲量を計算する。この処理は時間がかかる（1～数分程度）。

```
cf_temporal = np.zeros((nday,nhour,ndbin,ntype)) # 雲タイプ別の雲量を保存するための配列
for idy in range(nday):
    cot,ctp,mstat,lat,lon = read_ahi(files[idy])
    ctype = np.zeros(cot.shape)

    ctype[np.where((abs(mstat) >= 1) & (ctp < 300) & (cot>0) & (cot<=1))] = 1 # Ci
    ctype[np.where((abs(mstat) >= 1) & (ctp < 300) & (cot>1) & (cot<=6))] = 2 # Cs
    ctype[np.where((abs(mstat) >= 1) & (ctp < 300) & (cot>6))] = 3 # DC

    for ihr in range(nhour):
        dayno = day_st + idy + ihr/24 + 9/24
        lat_a,lon_a = best_track(dayno) # 台風中心の緯度・経度
        dis = calc_dist_ahipxl(lon_a, lat_a, lon, lat)
        c_count, edge1, edge2 = np.histogram2d(np.ravel(dis), \
            np.ravel(ctype[ihr,:,:]), bins=(ndbin,ntype), range=(disrng,typrng))
        p_count,edge = np.histogram(np.ravel(dis), bins=ndbin, range=disrng)

        for ityp in range(ntype):
            cf_temporal[idy,ihr,:,ityp] = c_count[:,ityp] / p_count *100
```

④ 結果をプロットする。

```
typname = ['Ci', 'Cs', 'DC']
x_val = dis_delta/2 + np.arange(ndbin) * dis_delta
y_val = np.arange(nday*nhour)/ nhour
fig = plt.figure(figsize = (12,7))

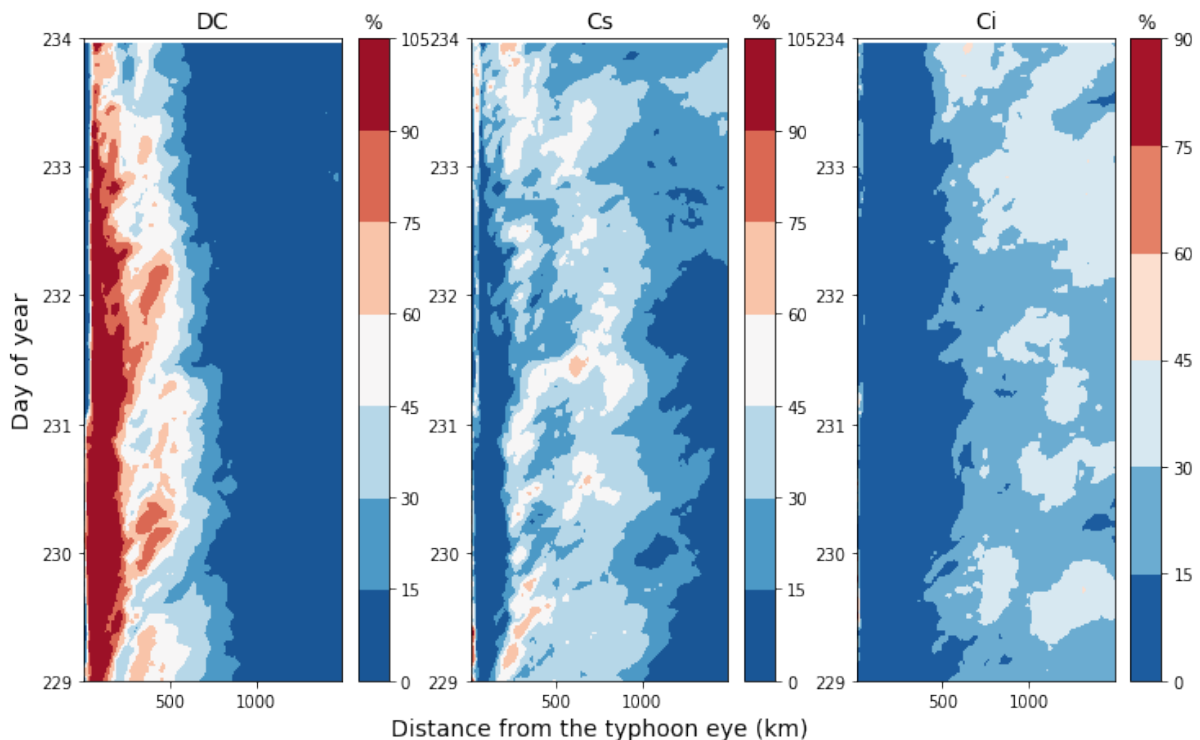
for ityp in range(3):
    ax = fig.add_subplot(1,3,ityp+1)
    xt = np.linspace(0,disrng[1],ndbin)
    plt.contourf(x_val, y_val, \
        cf_temporal[:, :, :, 2-ityp].reshape(nday*nhour,ndbin), cmap='RdBu_r')
    plt.title(typname[2-ityp], fontsize=14)
    plt.yticks(np.arange(6), ['229', '230', '231', '232', '233', '234'])
    plt.colorbar().ax.set_title('%')
    if ityp==0:
        plt.ylabel('Day of year', fontsize=14)
```

```

if ityp==1:
    plt.xlabel('Distance from the typhoon eye (km)', fontsize=14)

plt.savefig('Atsani_229-233_cf_cldtyp_variation.png')
plt.show()

```



(イ) 上記の結果を現地時間 0-24 時について 1 時間毎に平均し、横軸を台風の中心からの距離、縦軸を現地時間としたコンポジット図を作る。

データファイルに格納されている時間は UTC である。5 日間の平均的な日変化を見るため、UTC 時間を現地時間に変換し、現地時間毎の平均雲量を計算する。UTC と現地時間 (loc_time) の関係は以下の通りである。

$$\text{loc_time} = \text{UTC} + \text{lon} / 15$$

ただし、24 の剰余をとって、0-24 の値の範囲に収める。Python を用いた場合、UTC から loc_time を、numpy の mod 関数を用いて下記の通り計算することができる。

$$\text{loc_time} = \text{numpy.mod}(\text{utc} + \text{lon}/15, 24)$$

台風の中心位置の経度は約 145-160 度の範囲内にあるので、ここでは、lon=150 として UTC を現地時間へ変換する（より正確には各ピクセルの経度を考慮して、ピクセル毎に現地時間を計算する必要がある）。

⑤ 台風中心位置の経度を用いて5日間のUTC時間を現地時間に変換する。

```
ltc = np.zeros((nday,nhour)) # local timeを保存するための配列
for idy in range(nday):
    for ihr in range(nhour):
        lon_a = 150 # 代表的な経度
        ltc[idy,ihr] = int( np.mod(ihr+lon_a/15,24) ) # 現地時間の計算 (整数化した)
```

⑥ 現地時間0時に対応するUTCを求め(0-23の何番目に対応するか), 現地時間0時の値が先頭に来るよう, cf_temporalの配列要素をrollで循環移動させる。

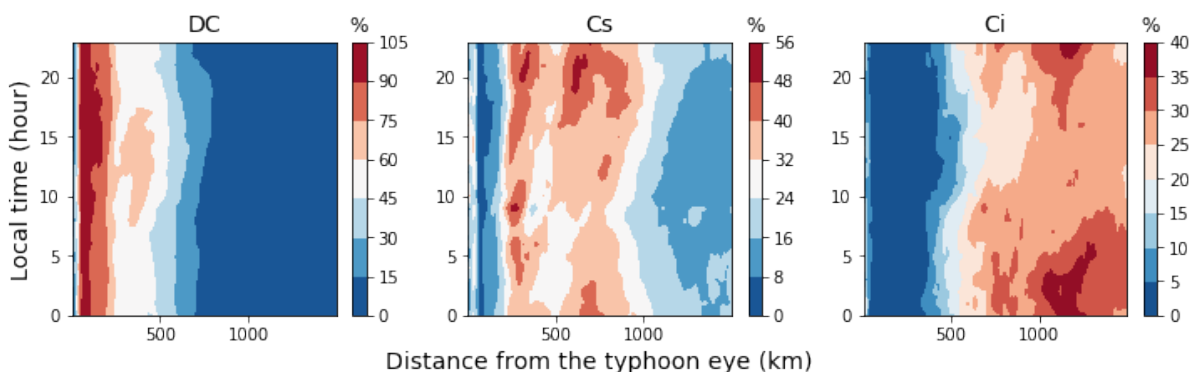
```
print(cf_temporal.shape)
cf_ltc = np.zeros_like(cf_temporal) # 時間ずれを格納する配列
for idy in range(nday):
    pos_ltc0 = 24-(np.where(ltc[idy,:]==0))[0] # 現地時間0時に対応する0-23UTCの番号
    cf_ltc[idy,:,:,:] = np.roll(cf_temporal[idy,:,:,:], pos_ltc0[0], axis=0)
```

⑦ 現地時間毎に雲量の平均を計算する。

```
print(cf_ltc.shape)
cf_daily = np.nanmean(cf_ltc, axis=0)
print(cf_daily.shape)
```

⑧ 結果をプロットする。

```
typname = ['Ci', 'Cs', 'DC']
x_val = dis_delta/2 + np.arange(ndbin) * dis_delta
y_val = np.arange(nhour)
fig = plt.figure(figsize = (12,3))
for ityp in range(3):
    ax = fig.add_subplot(1,3,ityp+1)
    plt.contourf(x_val, y_val, cf_daily[:, :, 2-ityp], cmap='RdBu_r')
    plt.title(typname[2-ityp], fontsize=14)
    plt.colorbar().ax.set_title('%')
    if ityp==0:
        plt.ylabel('Local time (hour)', fontsize=14)
    if ityp==1:
        plt.xlabel('Distance from the typhoon eye (km)', fontsize=14)
plt.show()
plt.savefig('Atsani_229-233_cf_cldtyp_dailyaverage.png')
```



(ウ) 上記(イ)の結果をさらに平均し、5日間の平均としての台風の中心からの距離と雲タイプ別の雲量の関係を求める（横軸は台風の中心からの距離、縦軸はタイプ別の雲量としてプロット）

⑨ 上記の現地時間毎の平均値をさらに平均する。

```
print(cf_daily.shape)
cf_total = np.nanmean(cf_daily[:,:,:],axis=0) # 平均
print(cf_total.shape)
```

⑩ 結果をプロットする。

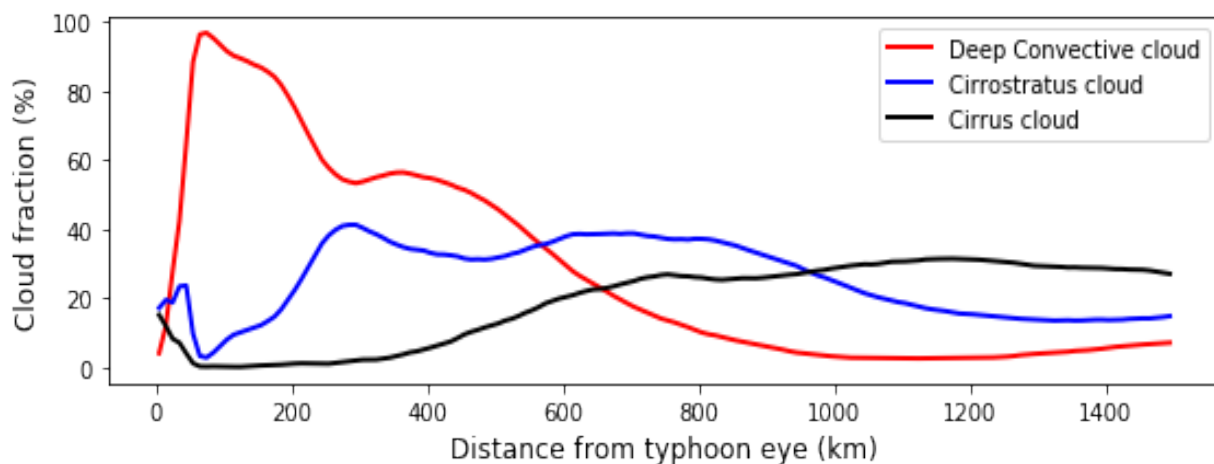
```
fig = plt.figure(figsize=(8,3))
ax = fig.add_subplot(1,1,1)

plt.xlabel('Distance from typhoon eye (km)',fontsize=12)
plt.ylabel('Cloud fraction (%)',fontsize=12)

plt.plot(x_val,cf_total[:,2],color='red',label='Deep Convective
cloud',linestyle='solid',linewidth=2)
plt.plot(x_val,cf_total[:,1],color='blue',label='Cirrostratus
cloud',linestyle='solid',linewidth=2)
plt.plot(x_val,cf_total[:,0],color='black',label='Cirrus
cloud',linestyle='solid',linewidth=2)

plt.legend(loc='best')
plt.tight_layout()

plt.savefig('average_all_cf.png')
plt.show()
```



3.3 台風の雲の鉛直分布の解析

CloudSat と CALIPSO がアッサニー台風を通過したときのデータを用いて、台風の雲の鉛直分布の解析を行う。具体的に以下のような解析を行う。

- A) DARDAR_MASK データを用いて東経 140 度から 170 度 および 北緯 10 度から 26 度の範囲内における cloud mask (cmsk) および Radar reflectivity (Zeff)の鉛直分布と Zeff に対する CFAD (Countoured Frequency by Altitude Diagram)を描く。
- B) DARDAR_MASK データにほぼ同期した 03:30 UTC における ICAS データを用いて、上記範囲内の上層雲 (雲頂高度 CTH > 10 km) の collocated pixels に対する雲光学的厚さ (COT)を読み取り、(i) $1 \leq COT < 13$ と (ii) $COT \geq 13$ の範囲に当てはまる CFAD を比較する。
- C) DARDAR_CLOUD データを用いて Ice water content (IWC) と Cloud particle effective radius (CER) の鉛直分布を描く。

実習 3.3

- A) DARDAR-MASK_2015231023311_49514_V2-11.hdf データファイルを用いて、東経 140 度から 170 度 および 北緯 10 度から 26 度の範囲内における cloud mask (cmsk) および Radar reflectivity (Zeff)の鉛直分布を描く

まず HDF ファイルのデータセットの読み込む方法を学ぶ。

- ① pyhdf.SD モジュールの SD と SDC クラスをインポートし、データファイルの中身を任意変数に格納する。

```
from pyhdf.SD import SD,SDC
file = 'DARDAR-MASK_2015231023311_49514_V2-11.hdf'
f = SD(file, SDC.READ)
```

- ② データファイルに格納されているデータセットの情報 (データセットの名前, サイズ, 属性など)を確認する。

```
for i in f.datasets().keys():
    print(i, f.select(i)[:].shape, f.select(i).attributes() )
    print(' ')
.
.
CLOUDSAT_2B_GEOPROF_CPR_Cloud_Mask (20678, 436) {'long_name': 'CloudSat cloud
mask', 'comments': 'Each CPR resolution volume is assigned 1 bit mask
value\\n0=No cloud detected\\n1=likely bad data\\n5=likely ground clutter\\n5-
10=week detection found using along track integration\\n20 to 40=Cloud
detected ..increasing values represents clouds with lower chance of a being a
false detection', 'missing_value': -9, '_FillValue': -9, 'valid_range': [40, 0],
'scale_factor': 1.0, 'scale_factor_err': 0.0, 'add_offset': 0.0,
'add_offset_err': 0.0, 'calibrated_nt': 5}
.
.
```

- `f.datasets().keys()` : `f` に含まれた変数(データセット)のキー (名前) (2.1.3 を参照)
- `f.select(i)` : `f` に含まれた `i` という変数
- `f.select(i).attributes()` : `f` に含まれた `i` という変数の属性

③ 変数 `CLOUDSAT_2B_GEOPROF_CPR_Cloud_Mask` を抽出する。

```
v = f.select('CLOUDSAT_2B_GEOPROF_CPR_Cloud_Mask')
```

④ 属性の `fillvalue`, `add_offset`, `scale_factor` を読みこむ。

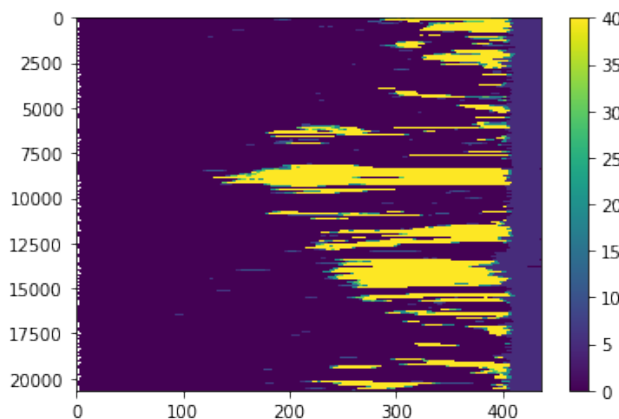
```
v_attr = v.attributes()
fv, offset, scale = v_attr['_FillValue'], \
    v_attr['add_offset'], v_attr['scale_factor']
```

⑤ 変数 `v` の `fillvalue` を `NaN` に置き換え, 整数値 (SI) を物理量 (PV) へ変換する。

```
v = np.array(v[:], dtype=np.float32)
v[np.where(v==fv)] = np.nan
v = scale*(v-offset)
```

⑦ データをプロットする。

```
plt.imshow(v, aspect='auto')
plt.colorbar()
plt.show()
```

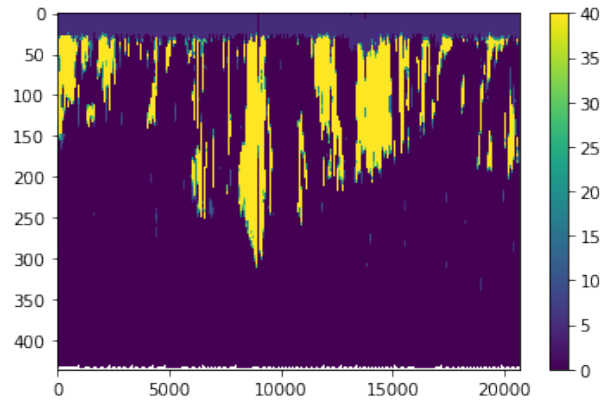


変数 `v` の配列のサイズを確認する。

```
v.shape
(20678, 436)
```

よって, 上記の画像 (配列データ) が 20678 行, 436 列の配列であることが分かる。データ構造の説明を確認すると, 行が `latitude`, 列が `height` であることが分かる。ここで, 上記の図の横軸と縦軸を入れ替える。

```
plt.clf() # 前の画像表示を取り消す
v_rot = np.rot90(v) # 配列を90度回転させる (Numpyを参照する)
plt.imshow(v_rot, aspect='auto')
plt.colorbar()
```

上記のデータセットの読みこみ処理は、以下の関数にまとめることができる。

function: read_hdf

```
def read_hdf(filename,varname):
    """
    Read HDF file

    Input
        filename : HDF file name
        varname   : variable name
    Output
        var       : variable data
    """
    f = SD(filename, SDC.READ)
    v = f.select(varname)
    v_attr = v.attributes()
    fv, offset, scale = v_attr['_FillValue'], v_attr['add_offset'],
    v_attr['scale_factor']
    var = np.array(v[:],dtype=np.float32)
    var[np.where(var==fv)] = np.nan
    var=scale*(var-offset)
    return var
```

ここから実際に DARDAR-MASK データファイルを扱っていく。

- ① functions.py 内に含まれている上記の関数を用いて、必要なデータを読み込む。

```
from functions import *
filename = 'DARDAR-MASK_2015231023311_49514_V2-11.hdf'
lat=read_hdf(filename,'CLOUDSAT_Latitude')
lon=read_hdf(filename,'CLOUDSAT_Longitude')
hgt=read_hdf(filename,'CS_TRACK_Height')
cmsk=read_hdf(filename,'CLOUDSAT_2B_GEOPROF_CPR_Cloud_Mask')
zeff=read_hdf(filename,'CLOUDSAT_2B_GEOPROF_Radar_Reflectivity')
```

- ② 表 3 に記述されている通り、読み込んだ zeff を 100*100 で割って単位を変換する。また、下端から上端になっている高度方向のデータの並びを逆順に修正する。

```
hgt = hgt[::-1] # 高度方向に反転
zeff /= 10000. # 表3を参照
```

③ 指定領域のデータを取り出す。

```
lonrng = [140,170]
latrng = [10,26]
llidx = np.where((lon>=lonrng[0]) & (lon<=lonrng[1]) & \
                 (lat>=latrng[0]) & (lat<=latrng[1]) )
lat = lat[llidx]
lon = lon[llidx]
cmsk = cmsk[llidx[0],:]
zeff = zeff[llidx[0],:]
```

④ 本解析では、雲出現領域のみの zeff を抽出するため、cmsk >= 20 のデータのみを用いる（表 3 を参照する）。そのため、それ以外の領域を無効値に置き換える。

```
zeff[np.where(cmsk<20)] = np.nan
cmsk[np.where(cmsk<20)] = np.nan
```

⑤ 行が height となるように、np.rot90 を用いて 90 度回転する。

```
cmsk = np.rot90(cmsk)
zeff = np.rot90(zeff)
```

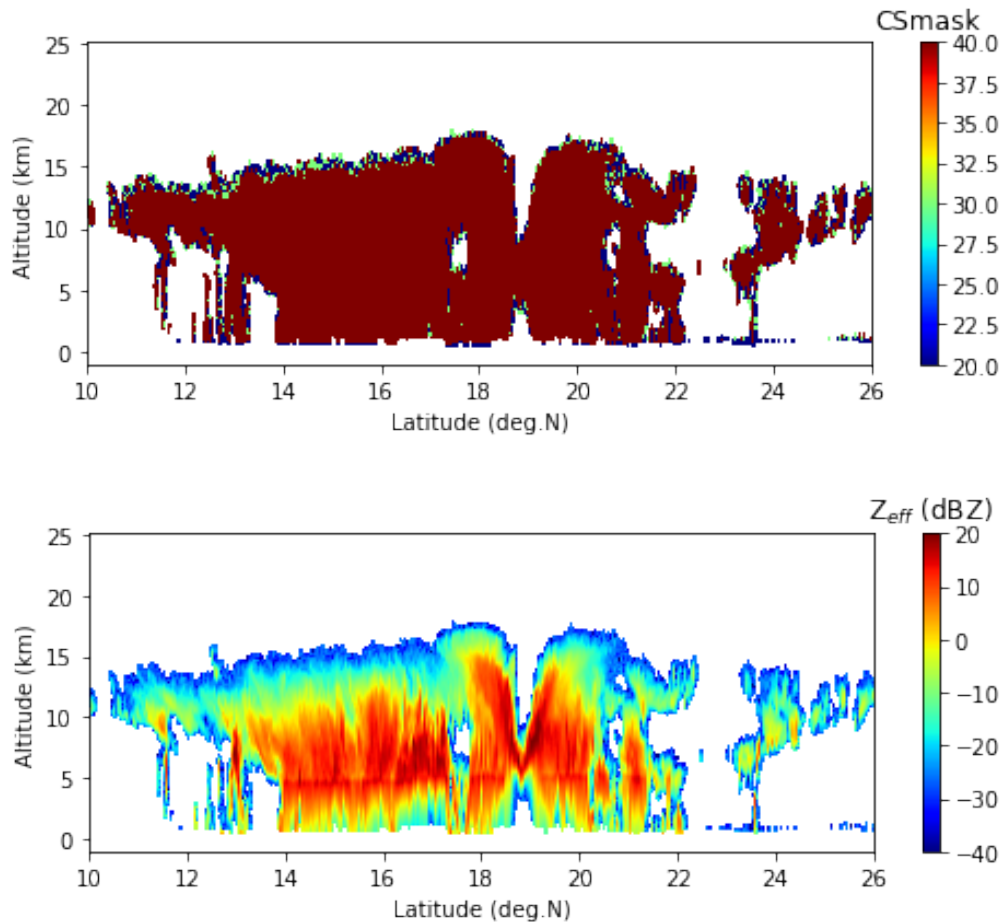
⑥ image_plot 関数を用いて図を作る。numpy の meshgrid を使い、緯度・経度の配列を 1 次元から 2 次元に変換する。

```
X,Y = np.meshgrid(lat, hgt)

xlabel = 'Latitude (deg.N)'
ylabel = 'Altitude (km)'
figsize = (7,3)

#mask
cbrng = [20,40]
cbtit = 'CSmask'
title = ' '
outfile = './csmask.png'
p = image_plot(X, Y, cmsk, xlabel, ylabel, cbrng, cbtit, title, figsize, outfile)

#zeff
cbrng = [-40,20]
cbtit = 'Z$_{eff}$ (dBZ)'
title = ' '
outfile = './radref.png'
p = image_plot(X, Y, zeff, xlabel, ylabel, cbrng, cbtit, title, figsize, outfile)
```



以降、Zeff のデータを用いて CFAD を描く。

- ⑦ zeff (横軸) と height (縦軸) の 2 次元ヒストグラムを計算するため、それぞれの範囲と間隔を指定し、ビンの数を求める。

```
hgt_rng = [0,20]          # heightの範囲 [km]
zeff_rng = [-30,30]       # zeffの範囲 [dBZ]
hgt_del = 0.24            # heightの間隔 [km]
zeff_del = 1              # zeffの間隔 [dBZ]
ny = int ( (hgt_rng[1]-hgt_rng[0])/hgt_del ) # x軸のビンの数
nx = int ( (zeff_rng[1]-zeff_rng[0])/zeff_del ) # y軸のビンの数
```

- ⑧ zeff 変数と同じサイズの新しい height 変数 (ここでは hgt_2d) を作成する。まず新しい配列を用意し、そこに height 変数の値を代入する。

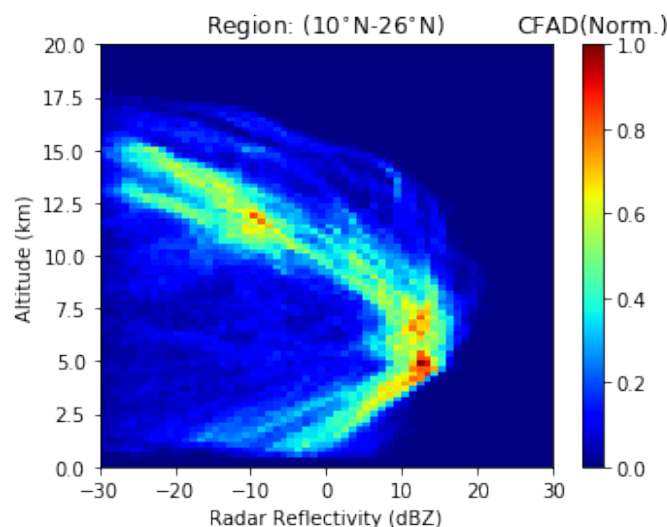
```
hgt_2d = np.zeros_like(zeff) # zeffと同じサイズの配列を用意
for ix in range(zeff.shape[1]): # hgt変数を各列に代入
    hgt_2d[:,ix] = hgt
```

- ⑨ numpy の histogram2d を用いて 2 次元頻度分布を計算する。

```
cfad, hgt_edge, zeff_edge = np.histogram2d(np.ravel(hgt_2d), \
                                           np.ravel(zeff), bins=(ny,nx), \
                                           range=(hgt_rng,zeff_rng))
```

⑩ 結果をプロットする。

```
xlabel = 'Radar Reflectivity (dBZ)'
ylabel = 'Altitude (km)'
cbrng = [0,1]
cbtit = 'CFAD(Norm.)'
title = 'Region: (10 $^{\circ}$ N-30 $^{\circ}$ N) '
outfile = 'cfad_all.png'
figsize = (5,4)
p = image_plot(zeff_edge, hgt_edge, cfad/np.nanmax(cfad), \
               xlabel,ylabel,cbrng,cbtit,title, figsize, outfile)
```



B) DARDAR_MASK データにほぼ同期した 03:30 UTC における ICAS AHI データファイルを用いて、上記範囲内の上層雲（雲頂高度 CTH > 10 km）の collocated pixels に対する雲光学的厚さ(COT)を読み取り、(i) $1 \leq COT < 13$ と(ii) $COT \geq 13$ の範囲に当てはまる CFAD を比較する

① ICAS AHI データファイルから必要なデータを読み込む。

```
file='AHI8_L2STI.I2015231.033000.v210.nc'
from netCDF4 import Dataset as dt
cot_ahi = np.array(dt(file,'r').variables['cot'][:,], dtype = np.float32)
cth_ahi = np.array(dt(file,'r').variables['cth'][:,], dtype = np.float32)
lat_ahi = np.array(dt(file,'r').variables['latitude'][:,], dtype = np.float32)
lon_ahi = np.array(dt(file,'r').variables['longitude'][:,], dtype = np.float32)
```

② 上記の calc_dist_ahipxl 関数を用いて DARDAR データファイルの各 pixel から一番距離が近く、かつ CTH > 10 km の条件を満たす AHI pixel を抽出する。計算時間の短縮のため DARDAR pixel の中心から ± 0.1 度までの AHI pixel を取り出し、その AHI pixel 群の中から DARDAR pixel に最も距離の近い pixel を抽出する。

```

lon_ahi = np.ravel(lon_ahi)          # AHIの2次元配列データを1次元配列へ変換
lat_ahi = np.ravel(lat_ahi)
cot_ahi = np.ravel(cot_ahi)
cth_ahi = np.ravel(cth_ahi)

cot_collpix = np.zeros_like((lat)) # collocated pixel のAHI COT
cth_collpix = np.zeros_like((lat)) # collocated pixel のAHI CTH
dis_collpix = np.zeros_like((lat)) # AHI-DARDAR collocated pixel間の距離 (km)
lat_collpix = np.zeros_like((lat)) # collocated pixel のAHI latitude
lon_collpix = np.zeros_like((lat)) # collocated pixel のAHI longitude

for i in range(lat.size):
    small = np.where((lon_ahi >= (lon[i]-0.1) ) & (lon_ahi <= (lon[i] + 0.1)) & \
                      (lat_ahi >= (lat[i]-0.1) ) & (lat_ahi <= (lat[i] + 0.1)) )

    #DARDAR pixelの周りの+/- 0.1度範囲におけるAHI pixelを取り出す
    cot_small = cot_ahi[small]      # 取り出したAHI pixels のCOT
    cth_small = cth_ahi[small]      # 取り出したAHI pixels のCTH
    lat_small = lat_ahi[small]      # 取り出したAHI pixels のlatitude
    lon_small = lon_ahi[small]      # 取り出したAHI pixels の longitude
    dis_small = calc_dist_ahipxl(lon[i],lat[i],lon_small,lat_small)

    # DARDAR pixel からの距離
    indx_min = np.where(dis_small == np.nanmin(dis_small)) # 一番近いpixelのindex
    cot_collpix[i] = cot_small[indx_min]
    cth_collpix[i] = cth_small[indx_min]
    dis_collpix[i] = dis_small[indx_min]
    lat_collpix[i] = lat_small[indx_min]
    lon_collpix[i] = lon_small[indx_min]

```

- ③ DARDAR と AHI の collocated pixels 間の距離をプロットする。この処理は時間がかかる（10-60 秒程度）。

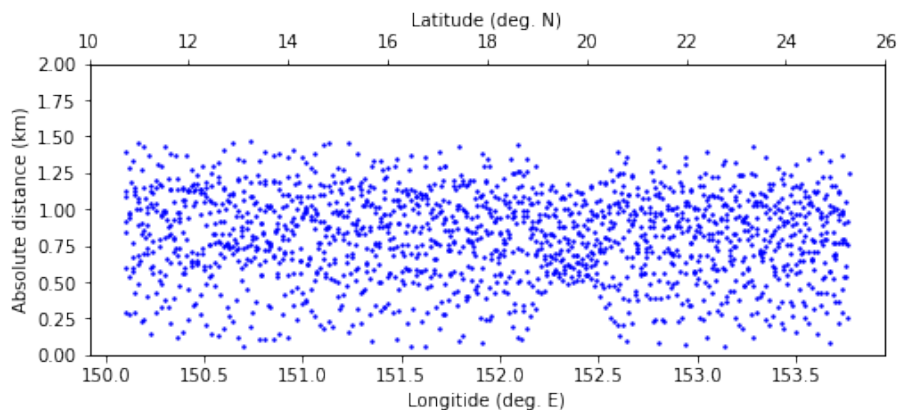
```

fig = plt.figure(figsize = (18,6) )

ax1 = fig.add_subplot(1,1,1)
ax1.set_xlabel("Longitude (deg. E)")
ax1.set_ylabel("Absolute distance (km)")
ax1.scatter(lon_collpix,dis_collpix,s=2,color='blue')
ax1.set_ylim(0,2)

ax2 = ax1.twinx()
ax2.set_xlabel("Latitude (deg. N)")
ax2.set_xlim(np.min(lat_collpix),np.max(lat_collpix))
plt.show()

```



- ④ 次は、 $CTH > 10$ km の $1 \leq COT < 13$ および $13 \leq COT$ に当てはまる Radar reflectivity (zeff) の index をそれぞれ抽出する。

```
iidx_0 = np.where( (cot_collpix >= 13) & (cth_collpix/1000. > 13.) )
iidx_1 = np.where( (cot_collpix >= 1) & (cot_collpix < 13) & \
                   (cth_collpix/1000. > 10.) )
```

- ⑤ 上記の index に当てはまる CFAD をそれぞれ計算する。

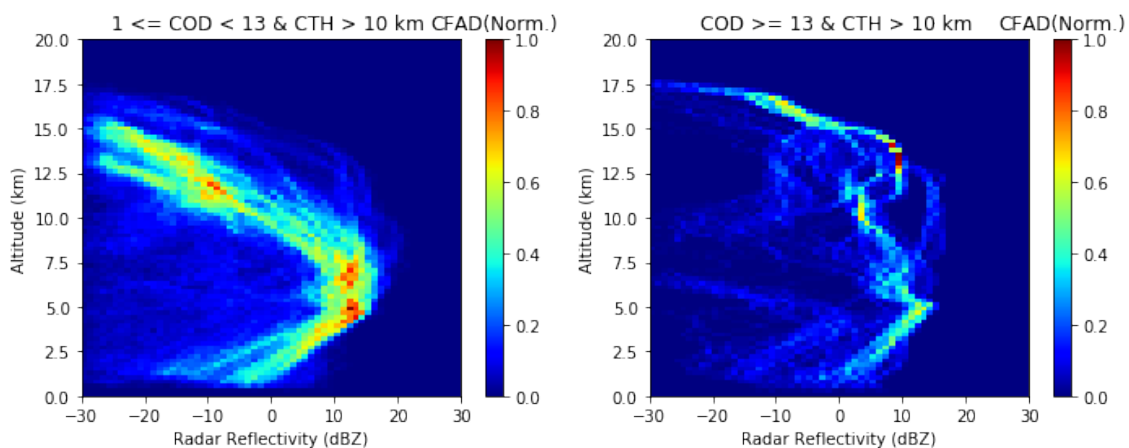
```
cfad_0, hgt_0, zeff_0 = \
    np.histogram2d(np.ravel(hgt_2d[:,iidx_0[0]]), \
                   np.ravel(zeff[:,iidx_0[0]]),bins=(ny,nx),range=(hgt_rng,zeff_rng))
cfad_1, hgt_1, zeff_1 = \
    np.histogram2d(np.ravel(hgt_2d[:,iidx_1[0]]), \
                   np.ravel(zeff[:,iidx_1[0]]),bins=(ny,nx),range=(hgt_rng,zeff_rng))
```

- ⑥ 結果をプロットする。

```
xlabel='Radar Reflectivity (dBZ)'
ylabel='Altitude (km)'
cbrng = [0,1]
cbtit='CFAD(Norm.)'
figsize = (5,4)

title='1 <= COD < 13 & CTH > 10 km '
outfile='cfad_COTlt13_CTHgt10km.png'
p=image_plot(zeff_1, hgt_1, cfad_1/np.nanmax(cfad_1), \
             xlabel,ylabel,cbrng,cbtit,title, figsize, outfile)

title='COD >= 13 & CTH > 10 km '
outfile='cfad_COTge13_CTHgt10km.png'
p=image_plot(zeff_0, hgt_0, cfad_0/np.nanmax(cfad_0), \
             xlabel,ylabel,cbrng,cbtit,title, figsize, outfile)
```



C) DARDAR_CLOUD のデータを用いて Ice water content (IWC) と Cloud particle effective radius (CER) の鉛直分布を描く

この手順は上記の (A) と一致する部分が多いので重要な手順のみ記述する。

① DARDAR_CLOUD のデータファイルを開き、データセットを読み込む。

```
from pyhdf.SD import SD,SDC
import numpy as np
filename = 'DARDAR-CLOUD_v2.1.1_2015231023311_49514.hdf'
iwc = SD(filename,SDC.READ).select('iwc')[:]
re = SD(filename,SDC.READ).select('effective_radius')[:]
lat = SD(filename,SDC.READ).select('latitude')[:]
lon = SD(filename,SDC.READ).select('longitude')[:]
hgt = SD(filename,SDC.READ).select('height')[:]
```

② iwc と re の配列を 90 度回転する。

```
iwc = np.rot90(iwc)
re = np.rot90(re)
```

③ 単位を修正する。

```
hgt = hgt[:, :-1]/1000. # m --> km
re *= 1000000. # [m] → [μm]
iwc *= 1000. # [kg/m-3] → [g/m-3]
```

④ 特定領域を切り出す。

```
lonrng,latrng=[140,170],[5,30]
llidx = np.where((lon>=lonrng[0]) & (lon<=lonrng[1]) & \
                 (lat>=latrng[0]) & (lat<=latrng[1]) )
lon,lat = lon[llidx[0]], lat[llidx[0]]
iwc = iwc[:,llidx[0]]
re = re[:,llidx[0]]
iwc[np.where(iwc <= 0)] = np.nan
re[np.where(re <= 0)] = np.nan
```

⑤ 図を作成する。

```
X,Y = np.meshgrid(lat,hgt)

xlabel = 'Latitude (deg.N)'
ylabel = 'Altitude (km)'
figsize = (7,3)

#iwc
cbrng = [-4,0]
cbtit = 'log$_{10}$ ( IWC ) (g/m$^3$)'
title = ''
outfile = 'iwc.png'
p = image_plot(X, Y, np.log10(iwc), xlabel,
              ylabel,cbrng,cbtit,title,figsize,outfile)

#re
cbrng = [0,120]
cbtit = 'r$_e$ ($\mu$m)'
title = ''
outfile = 're.png'
p = image_plot(X, Y, re, xlabel, ylabel,cbrng,cbtit, title, figsize, outfile)
```

