

6. Basemap の利用

matplotlib には等高線や陰影を描く機能がありますが、気象データの作図では海岸線と一緒に等高線や陰影を描きたい場合が多く、また様々な地図投影法で作図したい場合があります。ここでは、様々な地図投影法で作図することができる Basemap を用いて、地図上に等高線や陰影を描く方法を紹介します。

6.1 Basemap の基本

Basemap は matplotlib 上で地図を表示するためのパッケージです。メルカトル図法、ランベルト図法など様々な図法に対応しています。モジュールは `mpl_toolkits.basemap` で、`from mpl_toolkits.basemap import Basemap` のように Basemap をインポートします。

これまで紹介した matplotlib では、プロット領域を作成し、その上で作図を行いました。Basemap では matplotlib のプロット領域を使うので、作図の際には matplotlib の通常の方法と同じく、先にプロット領域を作成します。通常の方法と異なる点は、サブプロットを生成する所で `Basemap()` を使うことです (`basemap_sample.py`)。`m=Basemap()` でインスタンスを生成します。インスタンスを生成しただけでは何も表示されませんが、既に Basemap パッケージに含まれている様々な作図メソッドが使える状態になっています。作図の際には、`m.メソッド` のようなインスタンスメソッドを使います。

まずは、`m.drawcoastlines` で海岸線を描いてみます（図 6-1-1）。

```
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
fig = plt.figure() # プロット領域の作成
m = Basemap() # Basemap を呼び出し、インスタンスを生成
m.drawcoastlines() # 海岸線を描く
```

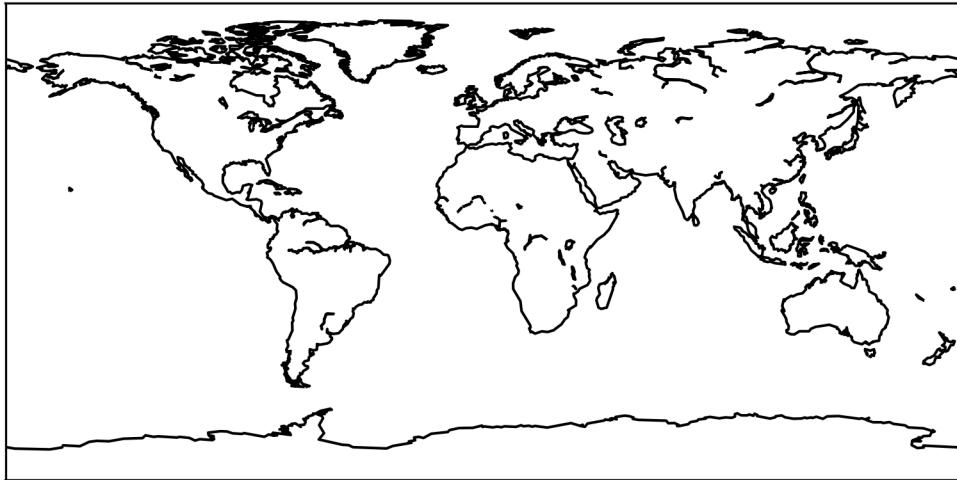


図 6-1-1 Basemap で海岸線を描く

描いた図に経度線と緯度線を加えてみます（図 6-1-2）。作図には basemap_sample2.py を使いました。経度線は `m.drawmeridians`、緯度線は `m.drawparallels` を使って引きます。いずれも 1 番目の引数として、線を引く緯度の配列を与えます。ここでは `np.arange(開始点, 終了点, 間隔)` を使い、緯度方向、経度方向ともに 30 度刻みの線を描きました。1 番目の引数には、このように生成された Numpy の ndarray の他に、手動で作成したリストやタプルを与えることも可能になっています。オプションの `color` は線の色、`fontsize` は目盛線ラベルの文字サイズを表します。ここでは黒色 (`color="k"`) のように、色の名前（かその省略形）で記述していますが、`color="0.8"` のように黒 ("0.0") ～白 ("1.0") までの値で記述することも可能です。この場合は、数字の 0.8 ではなく文字列の "0.8" と記述しないとエラーになります。

```
m.drawmeridians(np.arange(0, 360, 30), color="k", fontsize='small',
                  labels=[False, False, False, True]) # 経度線を引く
m.drawparallels(np.arange(-90, 90, 30), color="k", fontsize='small',
                  labels=[True, False, False, False]) # 緯度線を引く
```

`labels` オプションでは、ラベルを付けるかどうかを真偽値のリストで与えます。少し複雑ですが、経度線・緯度線ともに 4 つの要素で構成されたリストを与える必要があります、その内の 2 つの要素だけが作図に反映されます。まず経度線の

場合には、4つの要素は labels=[False, False, 上側, 下側]です。3番目と4番目の要素が上側と下側のラベルを付けるかどうか決めるのに使われます(図6-1-3)。ここでは False、True としたので下側に目盛線ラベルが付きました。最初の2つの要素は使われないので False にしておきます。緯度線の場合には、4つの要素は labels=[左側, 右側, False, False]です。最初の2つの要素が左側と右側の目盛線ラベルを付けるかどうかを決めるのに使われ、True、False にしたので左側だけに目盛線ラベルが付きました。こちらでは、3番目と4番目の要素は使われないので False にします。

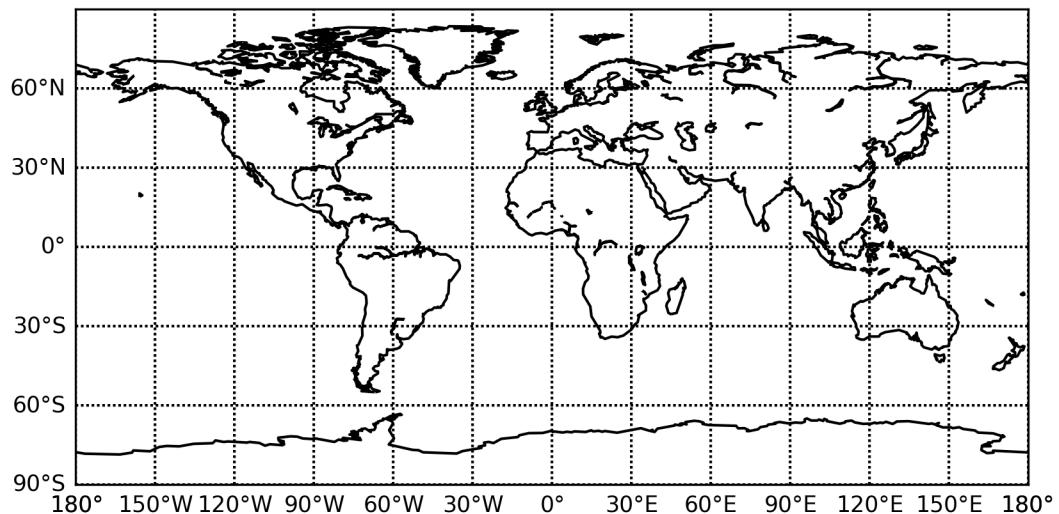


図6-1-2 経度線、緯度線を引く

経度線 (m.drawmeridians)



下側の目盛り

緯度線 (m.drawparallels)

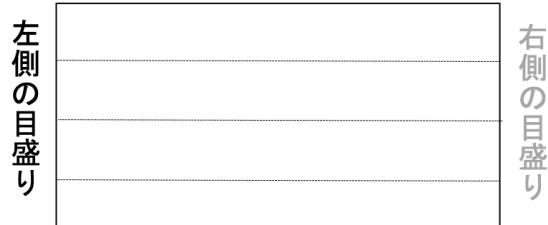


図6-1-3 経度線、緯度線と目盛り

なお m.drawcoastlines でも m.drawparallels や m.drawmeridians 同様、線

の色を指定する `color` が利用できます。また、これらのインスタンスメソッド全てに共通して、線の幅を指定する `linewidth` が利用可能で、デフォルト値は 1 です。`m.drawcoastlines` では、線種を指定する `linestyle` も利用でき、デフォルトの実線から変更可能です。例えば、海岸線を幅 1.5 で緑色の点線、経度線・緯度線を幅 2 で灰色の点線として描いてみます（図 6-1-4）。作図に使ったプログラムは `basemap_sample3.py` です。

```
m.drawcoastlines(color='g', linestyle='--', linewidth=1.5) # 海岸線を描く
m.drawmeridians(np.arange(0, 360, 30), color="gray", fontsize='small', \
    linewidth=2, labels=[False, False, False, True]) # 経度線を引く
m.drawparallels(np.arange(-90, 90, 30), color="gray", fontsize='small', \
    linewidth=2, labels=[True, False, False, False]) # 緯度線を引く
```

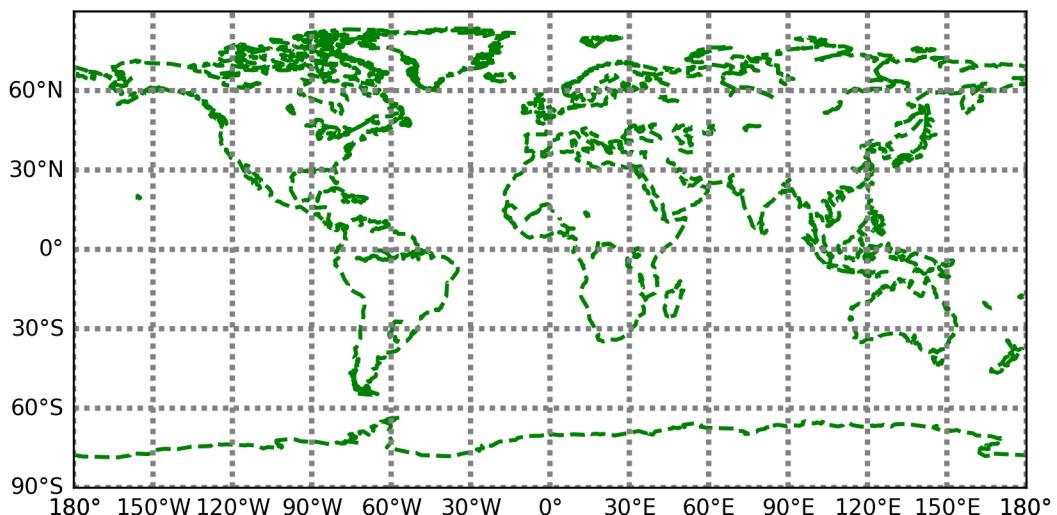


図 6-1-4 経度線、緯度線と海岸線の描画スタイルを変更

地図に国境線を加えることも可能で、`m.drawcountries` で行います（図 6-1-5）。オプションとして、海岸線を描く場合と同様に、線の色を指定する `color`、線の幅を指定する `linewidth`、線種を指定する `linestyle` が利用可能です。作図には `basemap_sample4.py` を使いました。

```
m.drawcountries() # 国境線を描く
```

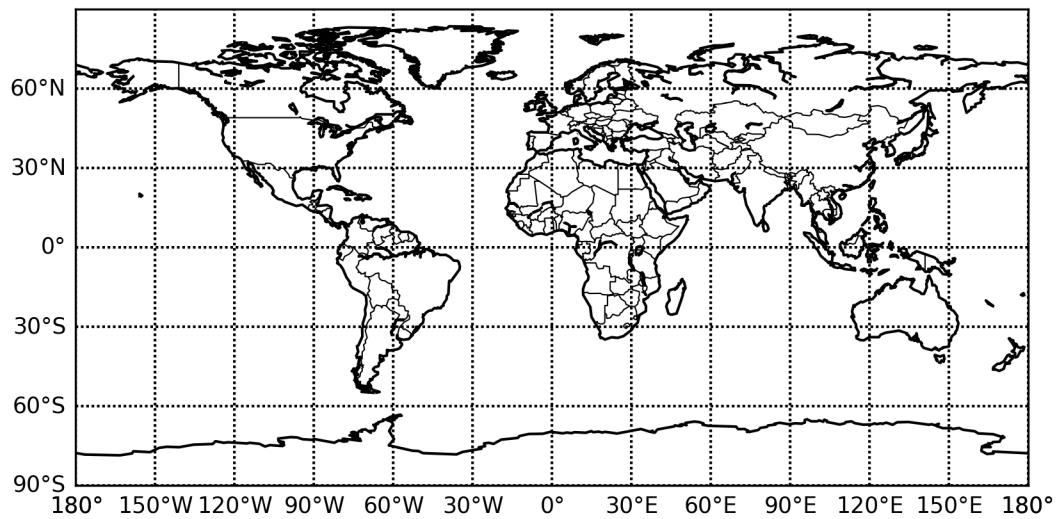


図 6-1-5 図 6-1-3 に国境線を加えた

なお、南北アメリカとオーストラリア限定ですが、`m.drawstates` で州の境界線を描くこともできるようになっています。違いが分かりやすいように、`m.drawstates` に `color='r'` オプションを与えて赤色で境界線を描いてみます（図 6-1-6）。作図には `basemap_sample5.py` を使いました。

`m.drawstates(color='r')` # 州の境界線を描く

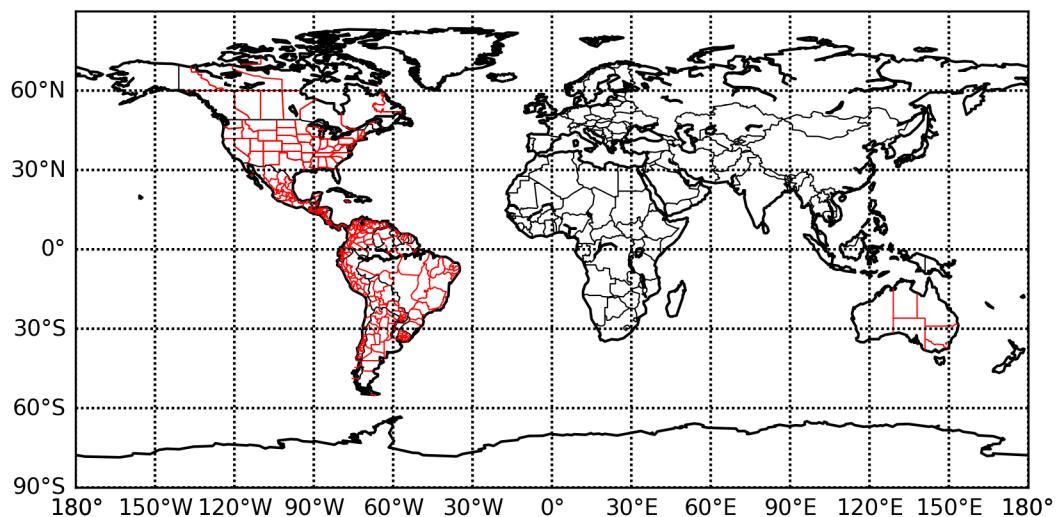


図 6-1-6 図 6-1-5 に赤色で州の境界線を加えた

地図に河川を描くことも可能になっており、`m.drawrivers` を使います。ここでは、`m.drawrivers` に `color='b'` オプションを与えて青色で河川を描きます（図 6-1-7）。作図には `basemap_sample6.py` を使いました。

```
m.drawrivers(color='b') # 河川を描く
```

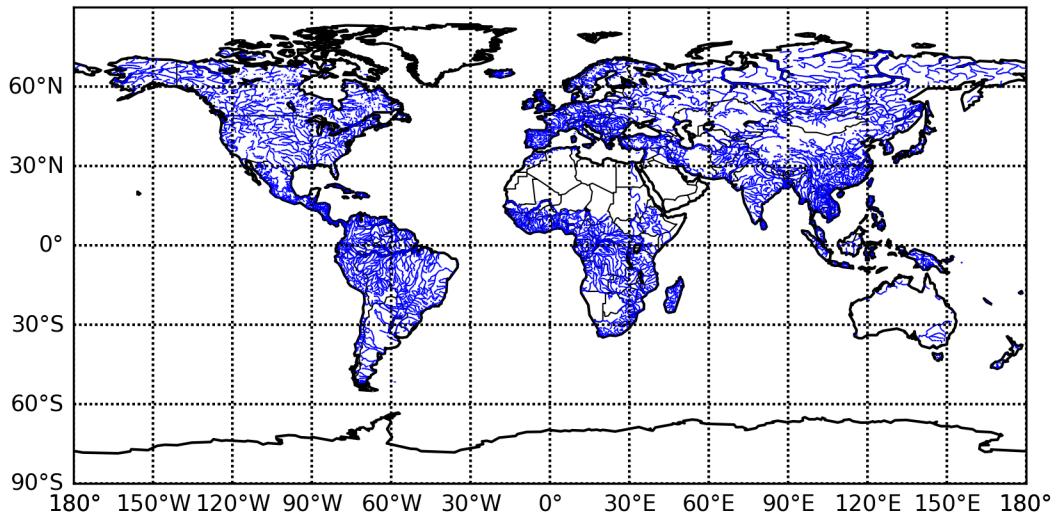


図 6-1-7 図 6-1-5 に青色で河川の分布を描いた

なお `m.drawstates`、`m.drawrivers` でも、線の幅を指定する `linewidth`、線種を指定する `linestyle` を利用可能です。

図の背景に色を付けることも可能で、`m.drawmapboundary` を使います（`basemap_sample7.py`）。ここでは、図の背景を水色で塗り潰すために、`fill_color='aqua'` のオプションを与えました。図 6-1-8 のように全体が水色に変わります。

```
m.drawmapboundary(fill_color='aqua') # 背景を塗り潰す
```

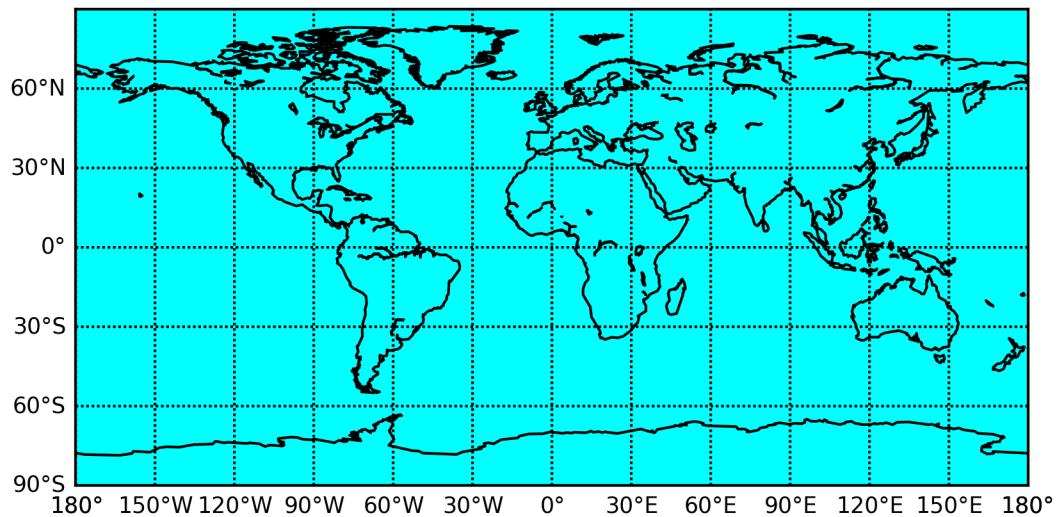


図 6-1-8 図の背景を塗り潰す

次に大陸部分だけ別の色で塗り潰します。こうすることで、図 6-1-9 のように海洋と大陸を別々の色で塗り分ける事が可能になります。大陸部分は `m.fillcontinents` で塗り潰します (`basemap_sample8.py`)。

```
m.fillcontinents(color='g') # 大陸を塗り潰す
```

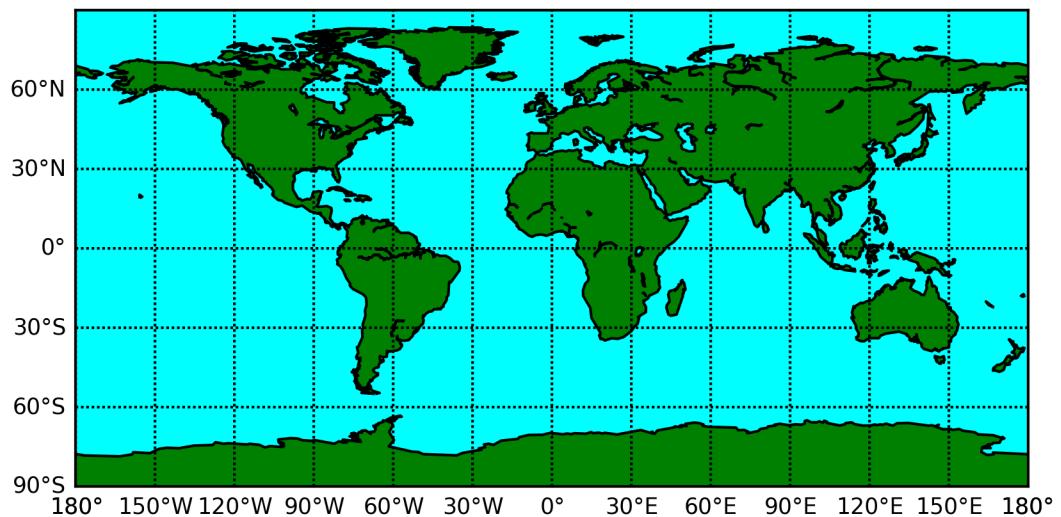


図 6-1-9 大陸部分を緑色で塗り潰す

図 6-1-10 のように、海洋を水色にして大陸は白抜きにしたいこともあるかと思います。basemap_sample9.py のように、大陸を塗り潰す色を color='w' とすることで白抜きにすることが可能です。

```
m.fillcontinents(color='w') # 大陸を白抜き
```

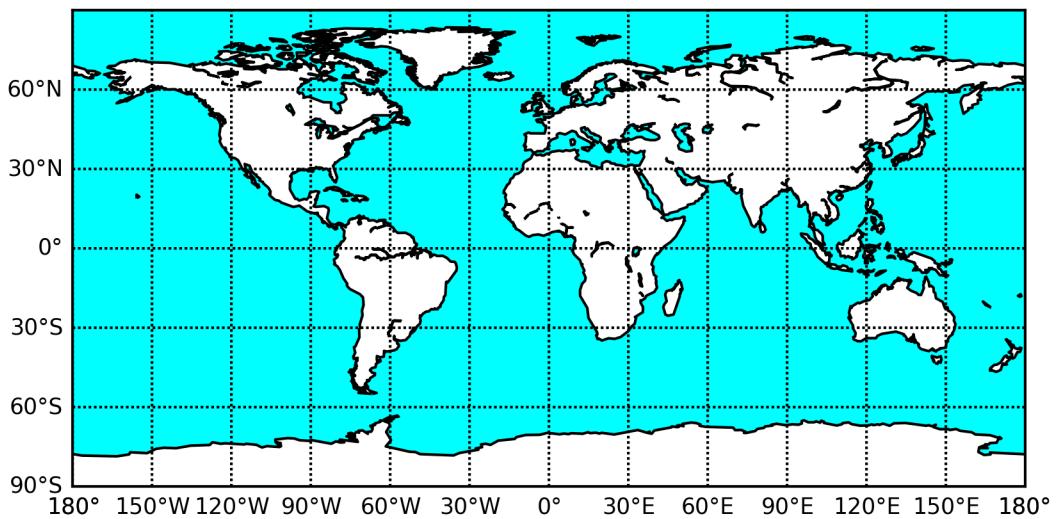


図 6-1-10 大陸部分を白抜きにする

図 6-1-10 で、カスピ海などの湖が水色のままになっているのに気が付いたでしょうか。m.fillcontinents は、デフォルトでは湖に色を付けない設定になっているので、背景色の水色が現れています。湖を別の色にすることも可能で、m.fillcontinents に **lake_color** オプションを与えます。ここでは湖を青色で塗り潰しました（図 6-1-11）。basemap_sample10.py で作図しました。

```
m.fillcontinents(color='w', lake_color='b') # 湖を青
```

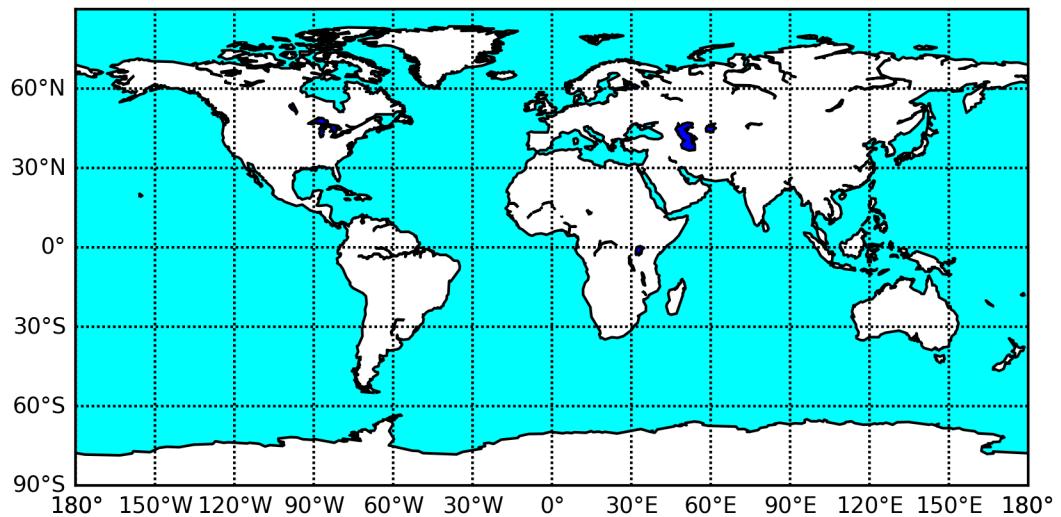


図 6-1-11 湖を青色で塗り潰した

他には、大陸や海洋を衛星画像風に表示する `m.bluemarble` や高度分布図風に表示される `m.etopo` があるので、簡単に紹介しておきます。図 6-1-12 は、`m.bluemarble()` を使い背景を変更したもので (`basemap_sample11.py`)。このように背景にイメージを表示するような場合には、同時に背景を塗り潰す `m.drawmapboundary` や大陸を塗り潰す `m.fillcontinents` を行っているとイメージが隠されてしまいます。プログラムではコメントアウトしているので、どのようになるか試してみましょう。

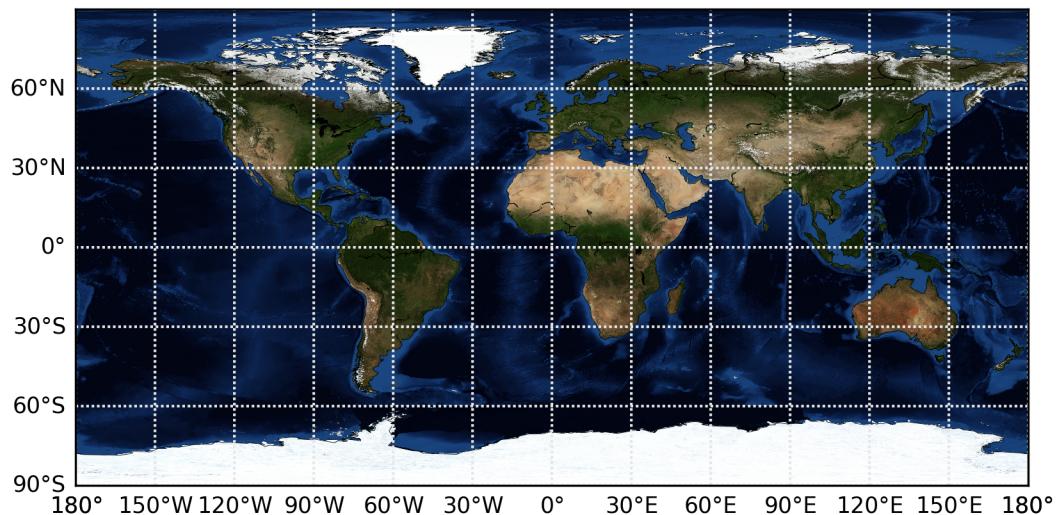


図 6-1-12 背景を `m.bluemarble` で衛星画像風に変えた

もう一つの `m.etopo()` を使って高度分布図風の作図を行ったものが、図 6-1-13 です。作図は `basemap_sample12.py` で行いました。

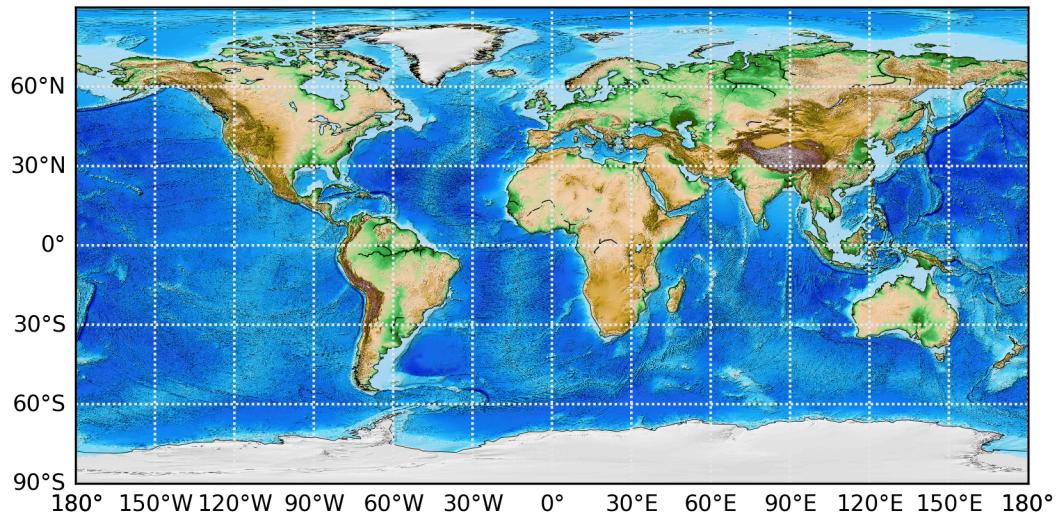


図 6-1-13 背景を `m.etopo` で高度分布図風に変えた

`m.bluemarble()` や `m.etopo()` では、作図にかかる時間を短くするために解像度を下げた作図も可能になっていて、`scale` オプションの数値として与えることが可能です。デフォルトでは、`scale=0.5` で、数値を下げるほど解像度が下がります。どのように変わっていくのかを図 6-1-14 に示しています。解像度を下げていっても、世界地図では `scale=0.05` くらいまでは許容範囲のように思えます。`scale=0.01` になると、地形が平均化された四角形が判別できるほどになっています。`basemap_sample13.py` で作図しました。

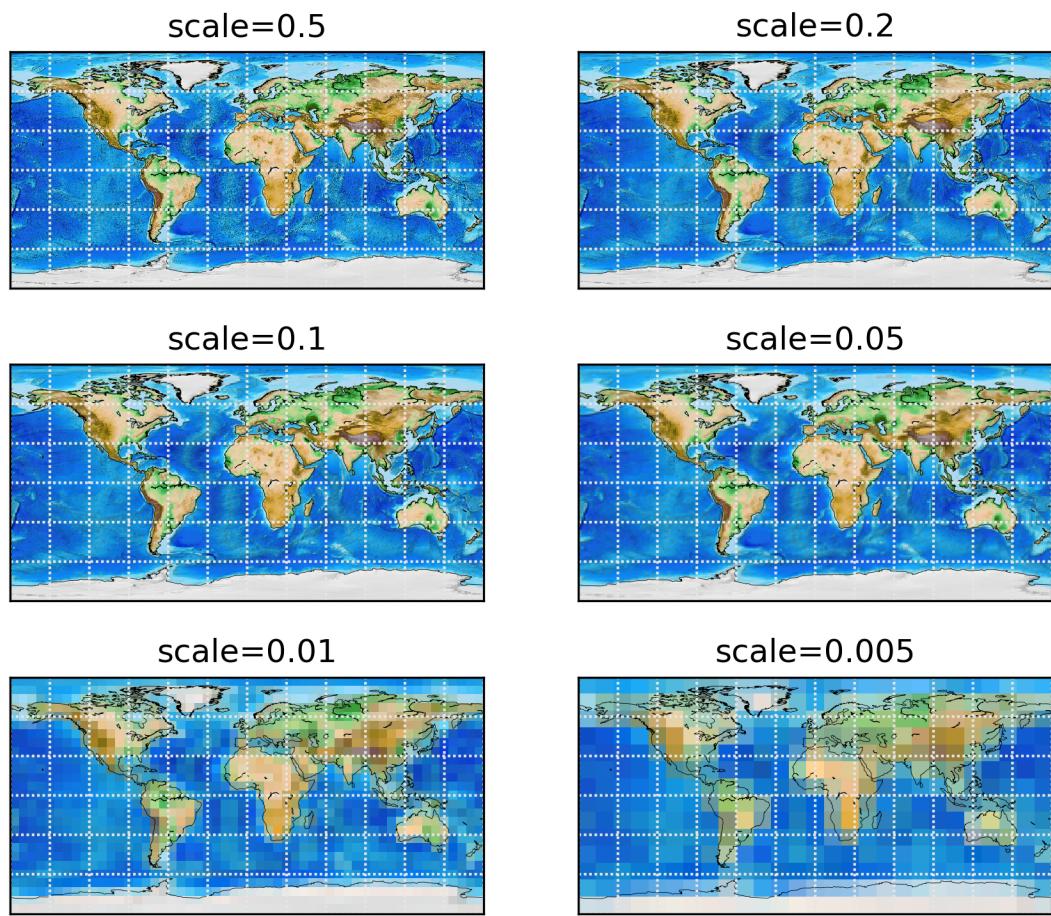


図 6-1-14 `scale` オプションを変えていった場合

6.2 様々な図法

6.2.1 正距円筒図法

前節で使っていたのは Basemap のデフォルトの図法で、正距円筒図法と呼ばれるものです。これまで GrADS を使っていた人にとっては、latlon として馴染みがあるかもしれません。正距円筒図法は、地球表面の球形を円筒に投影した図法の 1 つで、経度線と緯度線が直角かつ等間隔に交差します。経度線・緯度線が等間隔なので、極周辺の面積と赤道周辺の面積が地図上で同じ大きさになつており、極域ほど実際の面積より拡大して表示されます。

`m = Basemap(projection='cyl')` のように `projection` オプションで図法を明示しても、同じ図が生成されます。図 6-2-1 の左上はそのように作成したのですが、`lon_0=180, lat_0=0` オプションで中心となる経度を 180 度、緯度を 0 度に設定したので、図の中心に太平洋が来ています。

図 6-2-1 では、他に正射投影図法（右上）、メルカトル図法（左中）、ランベルト正角円錐図法（右中）、極投影図法（左下が北極、右下が南極中心）を並べています。作図には、`basemap_proj.py` を使いました。ここでは、サブプロットを生成して、それぞれのサブプロットに Basemap の地図を配置するため、前章までと同様にサブプロットを生成しています。`n=0` から 5 までのループを回していく、`n=0` の場合には正距円筒図法を描いています。タイトルを付ける場合には、Basemap のインスタンスではなく `ax.set_title()` を使います。

```
fig = plt.figure(figsize=(6,9)) # プロット領域の作成
for n in np.arange(6):
    ax=fig.add_subplot(3,2,n+1) # サブプロット生成
    if n == 0:
        # デフォルト(latlon, 正距円筒図法)
        m = Basemap(projection='cyl', lon_0=180, lat_0=0)
        ax.set_title("projection='cyl'")
```

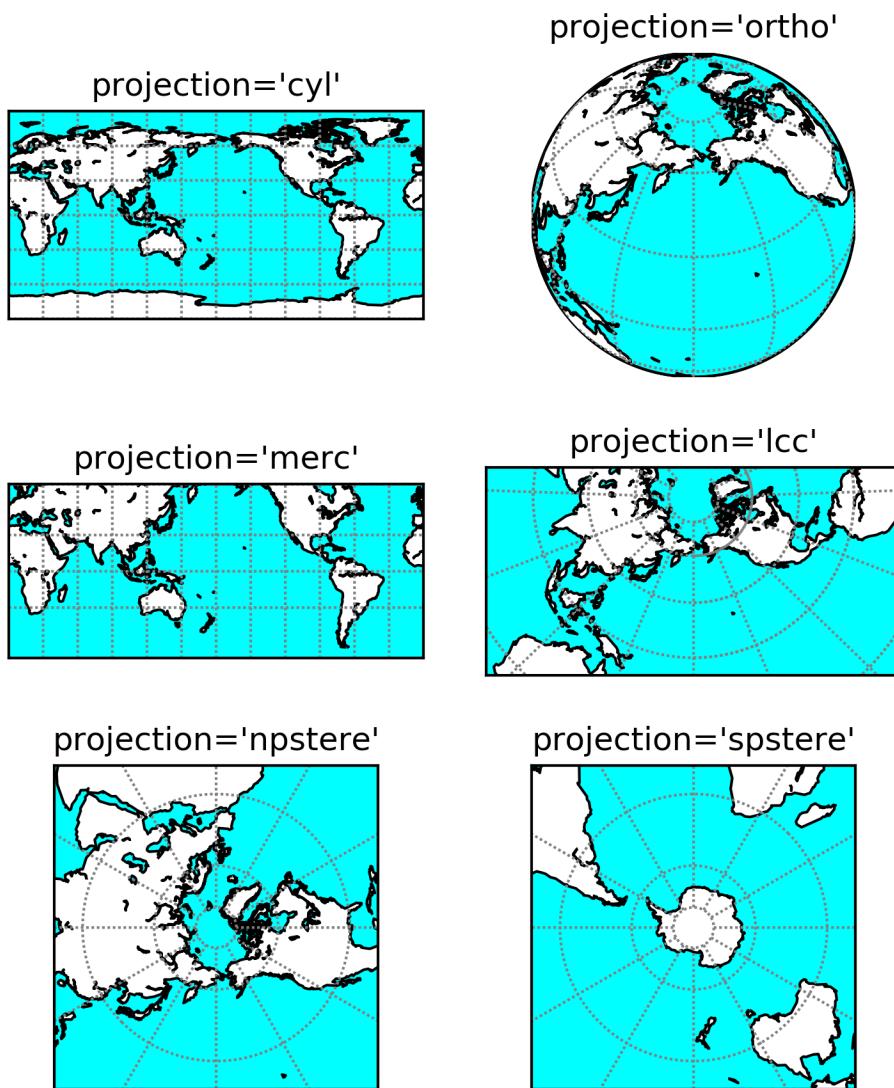


図 6-2-1 様々な図法。左上：正距円筒図法、右上：正射投影図法、左中：メルカトル図法、右中：ランベルト正角円錐図法、下：極投影図法（左が北極、右が南極中心）

6.2.2 正射投影図法

正射投影図法（あるいは平射図法）（orthographic projection）は、地球表面を平面に対して正射影した図で、無限遠点から地球を見た時の見え方に相当します。静止気象衛星など人工衛星から地球を撮影した時の画像に近い図法です（厳密には、静止気象衛星の軌道は無限遠点とみなすには近すぎますが）。半球より広い範囲を表すことはできず、図の端では歪みが大きくなります。正射投影図法で描く場合、projection='ortho'を用います。正射投影図法では、視点の中心となる経度・緯度を指定する必要があり、それぞれlon_0、lat_0のオプションに対応します。ここでは東経180度、北緯45度を中心に描いたので、北太平洋が手前に来て北極域全体を含むような図になっています。

```
elif n == 1:  
    # 正射投影図法  
    m = Basemap(projection='ortho', lon_0=180, lat_0=45)  
    ax.set_title("projection='ortho'")
```

6.2.3 メルカトル図法

メルカトル図法（Mercator projection）は正角円筒図法に属しており、地球表面を円筒に投影した図法で経度線と緯度線が直交します。16世紀にフランドル（現ベルギー）のメルカトルによって考案されました。地球表面のすべての部分で角度（方向）が正しく表され、出発地と目的地を結んだ直線と子午線が交わる角度を求めることができます。この特徴から、航海用の地図（海図）の図法として使われてきました。メルカトル図法で描く場合、projection='merc'を用います。この図法では、経度線の間隔は一定であるのに対して緯度線は高緯度に向かうほど間隔が広くなり、高緯度域の面積は低緯度域に対して大幅に拡大されます。また、南北両極が無限遠点になるため、全世界を1つの地図で描くことはできません。ここでは、南北60度の範囲に制限して描くようにしています。

```
elif n == 2:  
    # メルカトル図法  
    m = Basemap(projection='merc', llcrnrlon=0, urcrnrlon=360, ¶  
                llcrnrlat=-60, urcrnrlat=60)  
    ax.set_title("projection='merc'")
```

6.2.4 ランベルト正角円錐図法

ランベルト正角円錐図法 (Lambert conformal conic projection) は、地球表面を円錐に投影した図法の1つで、18世紀にドイツのランベルトによって考案されました。正角図法であるため風向などの方向が地図上で正しく表示され、中緯度で歪みが少ないという特徴もあるため、中緯度の地形図や気象庁の天気図に使われています。極が円錐の頂点になり、反対側の極は緯線半径が無限大になつて表示できないため、全世界を1つの地図で描くことはできません。ランベルト図法と略してしまうこともありますが、ランベルトの名前は他にも6.2.7節のランベルト正積円筒図法など計6種の図法に使われているため、混同される可能性がある場合は正式名称を使った方が良いでしょう。ランベルト正角円錐図法で描く場合、projection='lcc'を使います。この図法では中心となる経度・緯度を指定する必要があり、それぞれlon_0、lat_0のオプションを使います。中心となる経度・緯度からの図の幅と高さをwidthとheightで指定します。

```
elif n == 3:  
    # ランベルト正角円錐図法  
    m = Basemap(projection='lcc', lon_0=180, lat_0=50,   
                width=3e7, height=1.5e7)  
    ax.set_title("projection='lcc'")
```

6.2.5 極投影図法

極投影図法 (stereographic projection) は地球表面を平面に対して正射影した平射図法の一つです。正角図法であることが特徴で、北極、南極を中心とする場合には、経度線が極を中心とした直線、緯度線は極を中心とした円になります。この特徴から、極域に着目した図によく用いられます。全世界を1つの地図で描くことはできません。極投影図法で描く場合、北極中心の図ではprojection='npstere'、南極中心の図ではprojection='spstere'を使います。GrADSを使っていた人は、nps、spsとして作図したことがあるかもしれません。図の端となる緯度をboundinglatで与えます。北極を中心とした極投影図法でboundinglat=20とした場合、Basemapの場合には、ちょうど北緯20度で円になるのではなく、北緯20度が上下左右の枠に収まるような範囲の四角形

になります。

```
elif n == 4:  
    # 極投影図法（北極）  
    m = Basemap(projection='npstere', lon_0=180, boundinglat=20)  
    ax.set_title("projection='npstere'")  
elif n == 5:  
    # 極投影図法（南極）  
    m = Basemap(projection='spstere', lon_0=180, boundinglat=-20)  
    ax.set_title("projection='spstere'")
```

6.2.6 モルワイデ図法

他にも様々な図法があるので、いくつか紹介しておきます（図 6-2-2）。作図には、`basemap_proj2.py` を使いました。左上のモルワイデ図法（Mollweide projection）は、経度線を橍円弧、緯度線を赤道に平行な直線群で表し、地球全体を橍円形で表示することで、極域に近い場所の歪みを小さくした図法です。緯線は等間隔ではありません。緯度線は円筒図法の条件を満たしますが、経度線は満たさないため、擬円筒図法に分類されています。19世紀はじめにドイツのモルワイデが考案したもので、分布図等で使われます。モルワイデ図法で描くには `projection='moll'` を使います。中心の経度・緯度を指定することができ、赤道太平洋を中心にするため `lon_0=180`、`lat_0=0` としました。

```
if n == 0:  
    # モルワイデ図法  
    m = Basemap(projection='moll', lon_0=180, lat_0=0)  
    ax.set_title("projection='moll'")
```

6.2.7 ロビンソン図法

ロビンソン図法（Robinson projection）は、1960年代にアメリカのロビンソンが世界地図の表現に適した投影法として開発した図法です。正積図法でも正角図法でもないため、面積も角度も正しくはありませんが、ひずみが過大になる箇所がないように全体的なバランスを考慮し世界全体の特徴を捉えられるよ

うにしたため、分布図等に用いられています。ロビンソン図法で描くには、`projection='robin'`を使います。中心の位置を `lon_0` で指定することができ、太平洋を中心とする `lon_0=180` を指定しています。

```
elif n == 1:  
    # ロビンソン図法  
    m = Basemap(projection='robin', lon_0=180)  
    ax.set_title("projection='robin'")
```

6.2.8 ランベルト正積円筒図法

ランベルト正積円筒図法 (Lambert cylindrical equal-area projection) は、ランベルトによって考案された 6 種の図法の 1 つです。地球表面を円筒に投影した図法で経度線と緯度線が直交します。実際の面積との比が地球上のどこでも等しく表示される正積図法です。正積を保つために高緯度ほど緯度線の間隔が狭くなっています。南北には拡大され南北には圧縮され形の歪みは大きくなります。ランベルト正積円筒図法で描くには、`projection='cea'`を使います。

```
elif n == 2:  
    # ランベルト正積円筒図法  
    m = Basemap(projection='cea')  
    ax.set_title("projection='cea'")
```

6.2.9 ミラー図法

ミラー図法 (Miller cylindrical projection) は 1940 年代にアメリカのミラーが発表した図法です。メルカトル図法で南北両極が無限遠点になってしまいう問題を改善し、極域まで描くことができるようになります。全世界を 1 つの地図で表現できます。ただしメルカトル図法のように正角図法ではないので、角度は正しくありません。ミラー図法で描くには `projection='mill'`を使います。ここでは太平洋域を中心にするため、`lon_0=180` としています。

```
elif n == 3:  
    # ミラー図法  
    m = Basemap(projection='mill', lon_0=180)  
    ax.set_title("projection='mill'")
```

6.2.10 正距方位図法

正距方位図法 (azimuthal equidistant projection) は、図の中心点から見て距離と方向が地球上の全ての点について正しくなるようにした図法です。図の中心点からの距離を正確に表すので、飛行機の最短経路や方位を見積もる際に使われます。正距方位図法で描くには、projection='aeqd'を使います。また中心点の経度・緯度を lon_0、lat_0 で与えます。ここでは、東京付近の東経 140 度、北緯 35 度を中心にするため、lon_0=140、lat_0=35 としました。

```
elif n == 4:  
    # 正距方位図法  
    m = Basemap(projection='aeqd', lon_0=140, lat_0=35)  
    ax.set_title("projection='aeqd'")
```

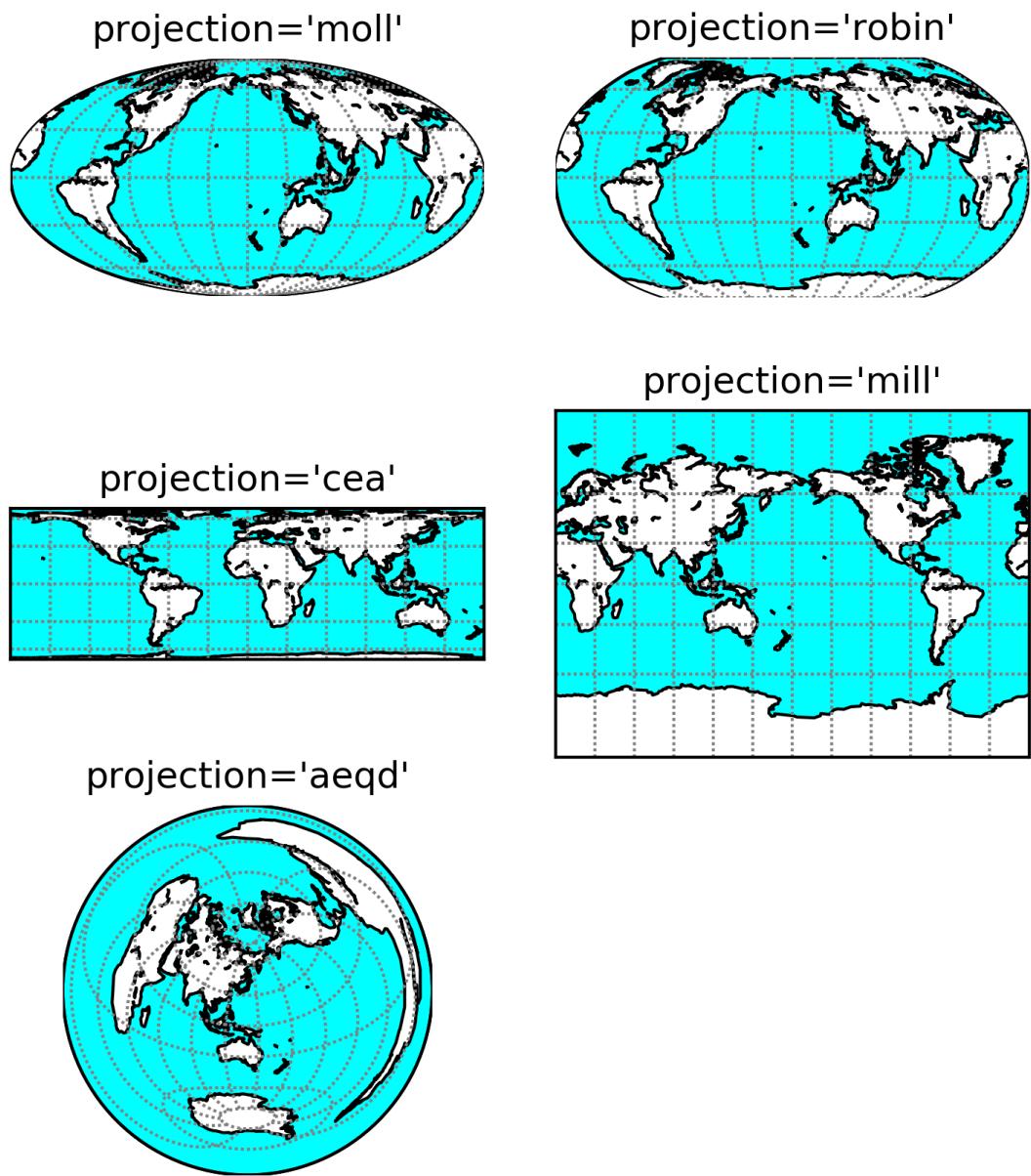


図 6-2-2 様々な図法。左上：モルワイデ図法、右上：ロビンソン図法、左中：ランベルト正積円筒図法、右中：ミラー図法、左下：正距方位図法

6.3 地域を限定した図

これまで世界地図を作成してきましたが、日本付近などに限定した図を作りたいこともあるかと思います。ここでは天気図に使われるランベルト正角円錐図法を例に、日本付近に限定した図を作成します。領域を限定する方法は2つあり、中心の位置と図の縦横の長さを指定する方法、及び図の経度・緯度範囲を指定する方法があります。

6.3.1 中心の位置と図の縦横の長さを指定する

中心の位置と図の縦横の長さを指定する場合、`lat_0=中心の緯度`、`lon_0=中心の経度`、`height=縦（南北）方向の長さ`、`width=横（東西）方向の長さ`のようなオプションを指定します（`basemap_jp.py`）。縦横方向は、図の端から端までの長さを表しており単位はmです。図6-3-1のような図が作成されます。

```
m = Basemap(projection='lcc', lat_0=35, lon_0=135, width=8000000, height=6000000)
```

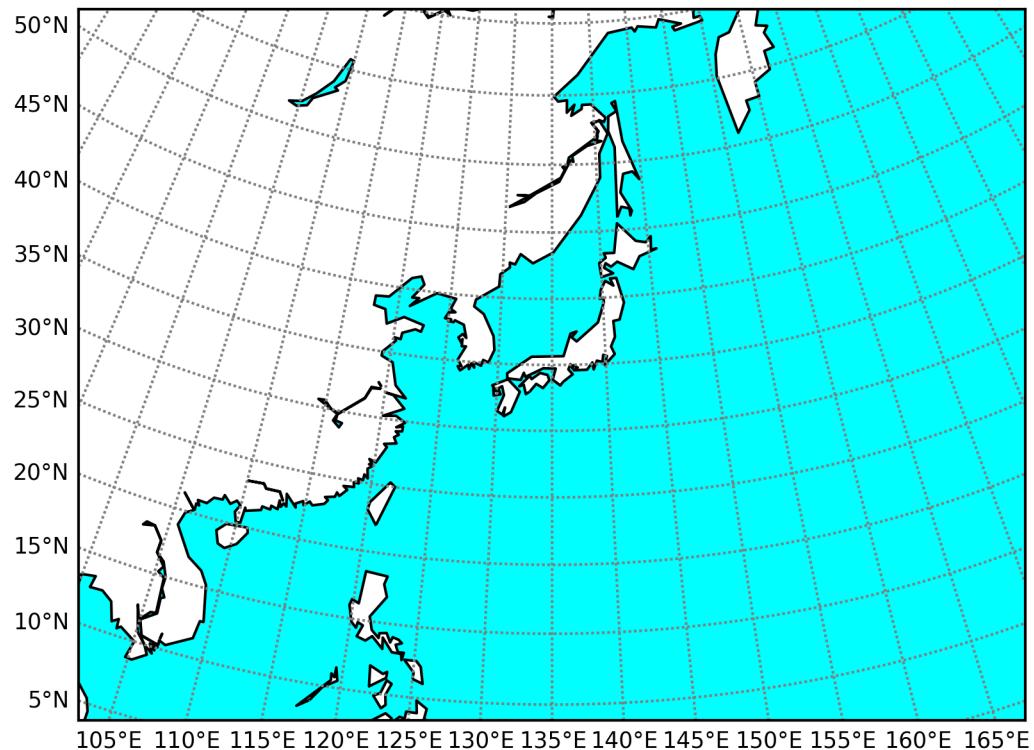


図6-3-1 日本周辺に限定した図をランベルト正角円錐図法で描いた

6.3.2 図の経度・緯度範囲を指定する

図の経度・緯度範囲を指定する場合には、llcrnrlat=緯度範囲下限、urcrnrlat=緯度範囲上限、llcrnrlon=経度範囲下限、urcrnrlon=経度範囲上限のようなオプションを指定します。指定した範囲が図の端に来るよう图の範囲が調整されます（図 6-3-2）。

```
m = Basemap(projection='lcc', lat_0=35, lon_0=135, ¥  
      llcrnrlat=10, urcrnrlat=60, llcrnrlon=100, urcrnrlon=180)
```

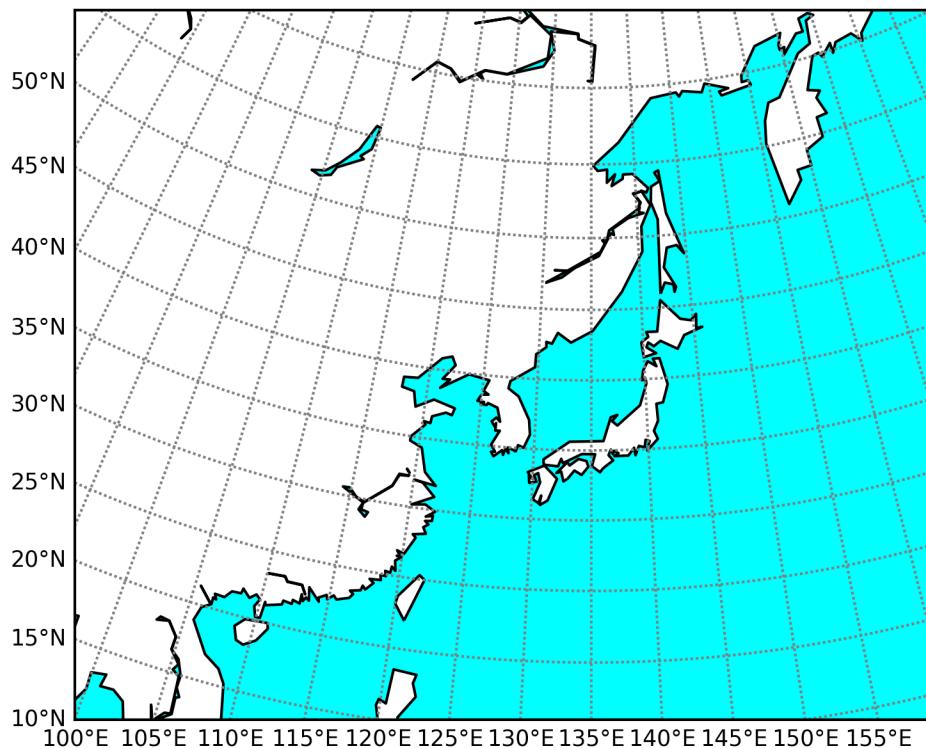


図 6-3-2 図の経度範囲を 100~180E、緯度範囲を 10~60N に指定した場合

6.3.3 地図の解像度を変更する

作成された図 6-3-1、図 6-3-2 では、淡路島が消えているなど、海岸線が荒く描かれています。海岸線の描き方はオプションで変更することが可能なので、どれくらい変わるのが試してみましょう（basemap_reso.py）。図 6-3-3 では、地図の解像度を指定する resolution オプションに、左上は resolution='c' (coarse、粗い解像度)、右上は resolution='l' (low、低解像度)、

resolution = 'i' (intermediate、中間解像度)、resolution='h' (high、高解像度)、resolution='f' (full、最高解像度) を与えました。右下はデフォルトの設定でresolution='c'と同じです。粗い解像度では四国や紀伊半島、能登半島の形がかろうじて分かるくらいで、低解像度で海岸線の大まかな輪郭が現れます。中間解像度になると、ようやく淡路島、琵琶湖、小豆島などの大きさまで解像されるようになります。高解像度では瀬戸内海の島々まで解像できるようになります。最高解像度では描かれる島の数が増えたのが分かるかと思います。作図にかかる時間は解像度を上げていくほど長くなり、また広い範囲の地図で細かい部分まで表現しようとすると海岸線が潰れてしまうので、作図範囲や作図の目的に合わせて適切な解像度を選ぶようにしましょう。

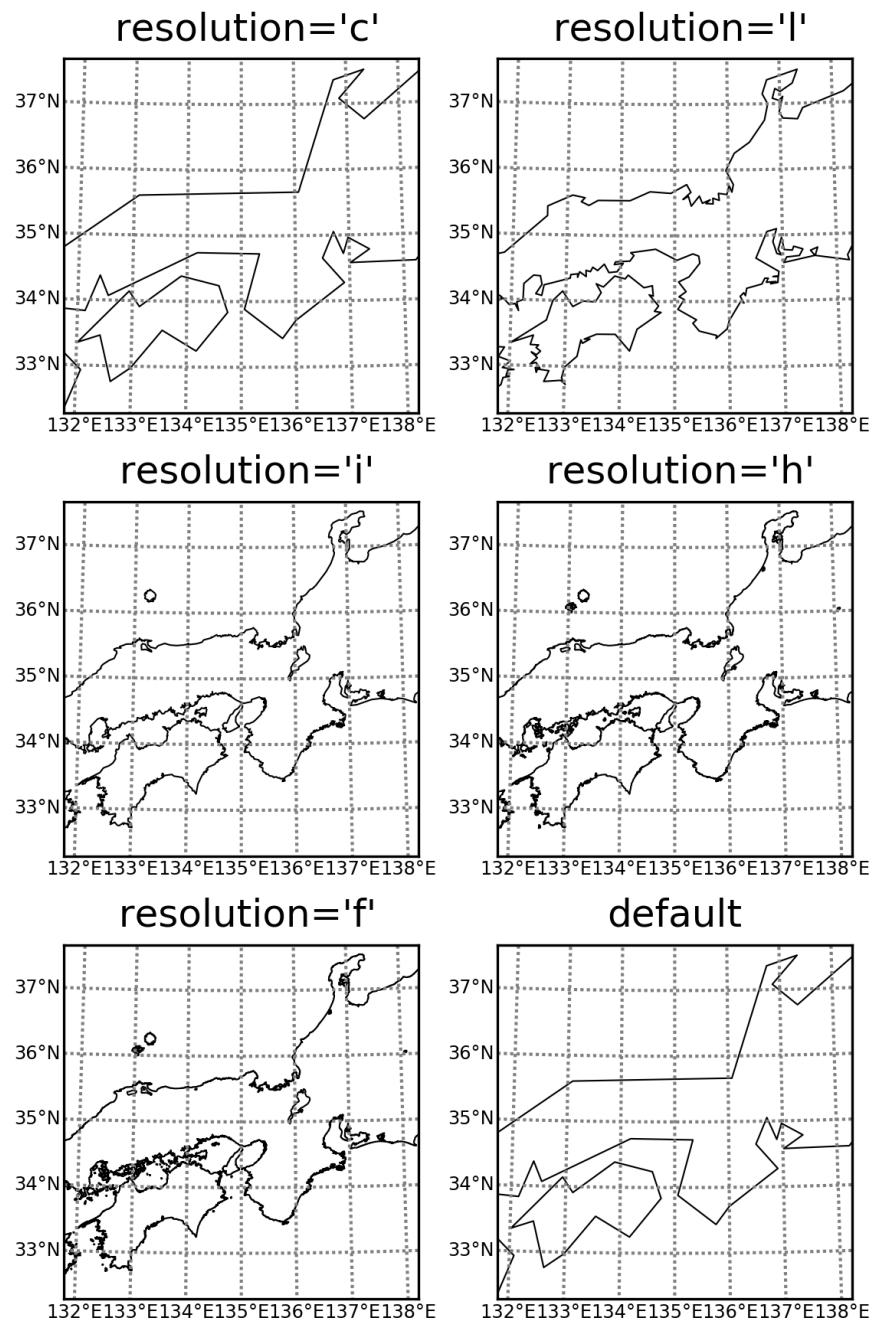


図 6-3-3 地図の解像度を指定する `resolution` オプションを変えた

`resolution` オプションは、河川を描く際の解像度も同時に変更しています。図 6-3-4 は図 6-3-3 に河川マップを重ねたもので、解像度を上げていくほど細かい流路まで表現されているのが分かります。しかし描かれる河川は解像度にはよらず、最高解像度でも吉野川などは描かれていません。なお、作図には

basemap_reso2.py を用いました。

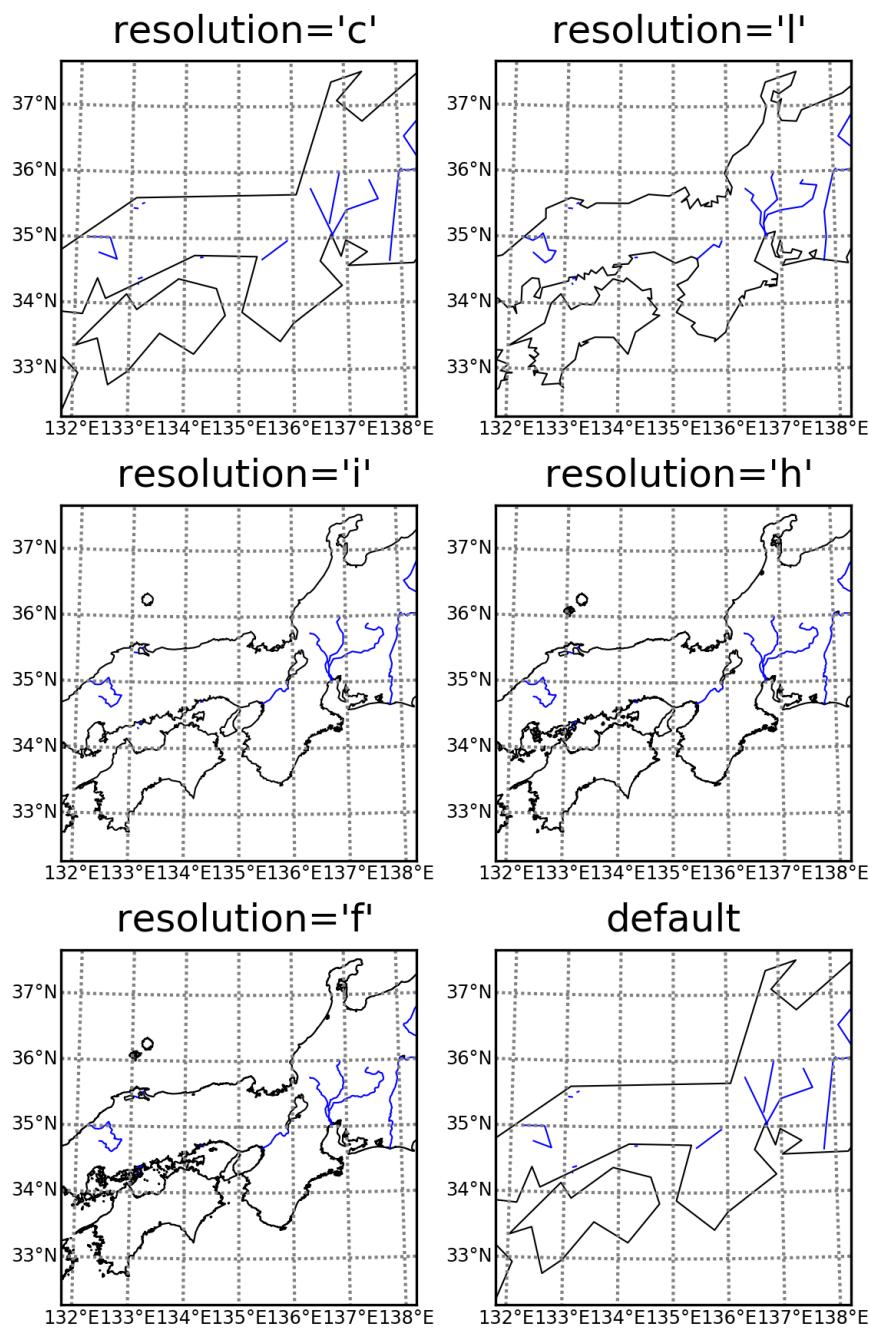


図 6-3-4 地図の解像度を指定する `resolution` オプションを変えた

6.3.4 地図上にマーカーやテキストをプロット

地図上に場所を示すためのマーカーや地名などをプロットしたいことがあると思います。ここでは、図 6-3-2 にマーカーと都市名を追加してみます（図 6-3-5）。作図には basemap_marker.py を使いました。使用可能なマーカーは表 3-3-1 に、色は図 3-2-3 に載せています。

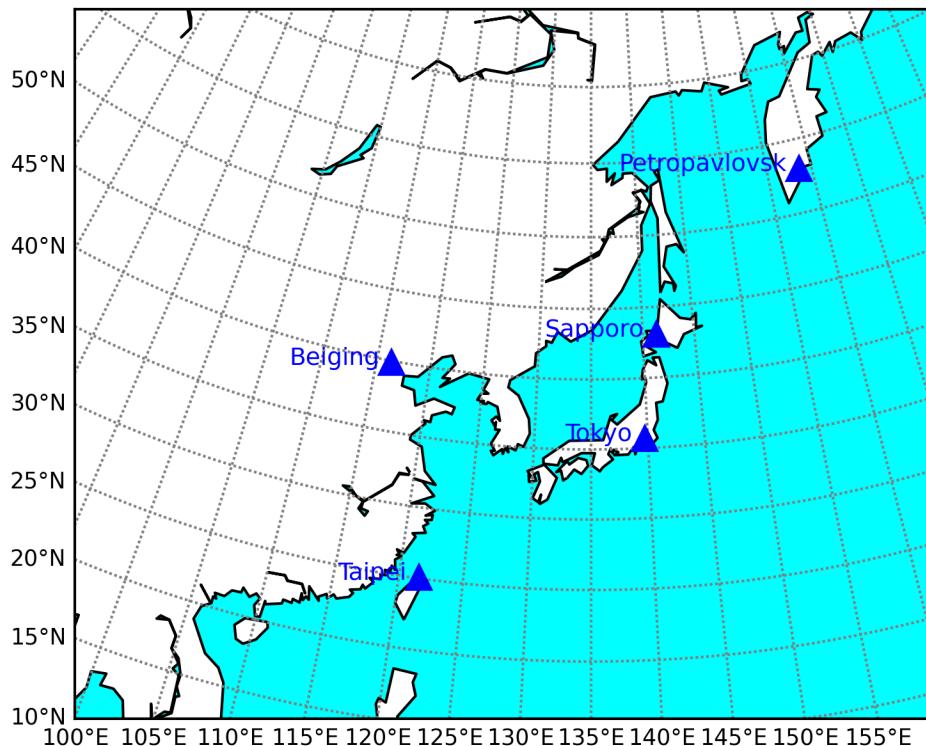


図 6-3-5 図 6-3-2 にマーカーと都市名を追加した

まずは準備として、マーカーを描く経度・緯度と都市名を、それぞれリストに格納しておきます。

```
# マーカーを描く経度・緯度・都市名
lats = [43.06, 35.69, 25.08, 39.9, 53.02]
lons = [141.328, 139.75, 121.55, 116.38, 158.65]
cities = ['Sapporo', 'Tokyo', 'Taipei', 'Beijing', 'Petropavlovsk']
```

格納したリストを元にマーカーを描き、マーカーの周辺に都市名をテキストでプロットしていきます。マーカーをプロットするには `m.plot`、都市名をプロットするには `plt.text` を使います。`m.plot` の 1 番目と 2 番目の引数には、図法の座標に変換した後の経度・緯度を与えます。三角形のマーカーを描くため `marker='^'`、青色にするため `color='b'` とし、マーカーサイズは 9 としました (`markersize=9`)。テキストはマーカーの左上に少し座標をずらして描くために、 x 方向にマイナス 100000 m、 y 方向にプラス 30000 m を加え、水平方向は右端揃え (`ha='right'`) としました。

```
x, y = m(lons, lats) # 図の座標へ変換
# マーカーをプロット
for xc, yc in zip(x, y):
    m.plot(xc, yc, marker='^', color='b', markersize=9)
# 都市名をプロット
for name, xpt, ypt in zip(cities, x, y):
    plt.text(xpt - 100000, ypt + 30000, name, fontsize=9, color='b',
              ha='right', va='center')
```

なお、マーカーのプロットを行う際には、次のようにマーカーと色をまとめて記述することも可能です。但し、マーカー、色ともに 1 文字しか使うことができないため、マーカーは表 3-3-1 の左側のみ、色は図 3-2-2 の 8 色のみが使用可能です。

```
m.plot(xc, yc, "b^", markersize=9)
```

6.4 気象データの読み込み

地図の作成方法が分かったので、次は作成した地図上にデータをプロットすることを考えます。その準備として、まずは気象データにどのようなフォーマットが使われているか紹介したいと思います。

6.4.1 気象データのフォーマット

気象データは web 上からダウンロードしたり、ハードディスクや DVD 等の物理デバイス上に保存して送付したりといった方法で共有されています。現在ではネットワークの転送速度が向上したため、web 上からのダウンロードが主流になりつつあります。いずれのケースでも、転送にかかる時間の短縮やディスク容量の制約のために、大容量データを保存する際にはバイナリ形式が用いられ圧縮等の変換も行われています。一方で地点観測データなど容量の少ないデータの配信等では、人間が直接読むことができるテキスト形式も用いられます。

気象データに用いられているバイナリ形式のフォーマットとしては、大容量の客観解析データや衛星観測データ等の提供に用いられている NetCDF (Network Common Data Form) 形式、GRIB 形式、GRIB2 形式、HDF (Hierarchical Data Format) 形式があります。テキスト形式のものとしては、地点観測データの提供時に用いられる CSV (comma-separated values) 形式 (コンマ区切りのテキストデータ) や、タブ区切りの TSV (tab-separated values) 形式、半角スペース区切りの SSV (space-separated values) 形式等があります。また、区切り文字で分割する代わりに固定長で分割したテキストデータもあります。人間が読むことを前提として、自由形式のテキストデータ (例えば文章等) で気象情報が配信されることもあります。表 6-4-1 に気象データに用いられるフォーマットをまとめておきました。python には NetCDF データ等を取り扱うためのライブラリや、csv 形式等のデータを読み込むためのライブラリがあるので、それらを利用して気象データを読み込み、データをプロットしていきます。

表6-4-1 気象データに用いられる主要なフォーマット

種類	テキスト/バイナリ	Pythonのツール	説明
CSV形式	テキスト	pandas.read_csv()	コンマ区切りのテキストデータ
TSV形式	テキスト	pandas.read_table()	タブ区切りのテキストデータ
SSV形式	テキスト	pandas.read_csv(sep='¥s+')	半角スペース区切りのテキストデータ
固定長分割	テキスト	pandas.read_fwf()	固定長で分割したテキストデータ
単純なバイナリ形式	バイナリ	numpy.fromfile()	改行記号等を含まないバイナリデータ、機種依存
NetCDF形式	バイナリ	netCDF4.Dataset()	データ、格子点情報、データの説明を格納、機種非依存
GRIB1形式	バイナリ		WMOが定めたバイナリデータ交換形式、機種非依存
GRIB2形式	バイナリ	pygrib.open()	WMOが定めたバイナリデータ交換形式、機種非依存
HDF4形式	バイナリ		データ情報も格納したバイナリ形式、機種非依存
HDF5形式	バイナリ	pandas.read_hdf() h5py.File()	データセットの階層構造を持ったバイナリ形式、機種非依存

6.4.2 CSV 形式（アメダス）

現在、CSV 形式で提供されている気象データとしてアメダスのデータがあります。こうしたデータを扱う際には、Pandas の read_csv を使うのが便利です (basemap_readcsv.py)。スペース区切りのテキストデータの場合は、5.5.4 節で紹介した read_fwf を使います。サンプルプログラムでは、2019 年 8 月 26 日～28 日の佐賀県、福岡県、長崎県のアメダス降水量データを読み込むようにしています (20190826-20190828-amedas_prep.csv)。この時期には前線が日本付近に停滞して九州北部で大雨となり、28 日には佐賀県、福岡県、長崎県に大雨特別警報が発表されました。

(<https://www.jma.go.jp/jma/kishou/know/jirei/sokuhou/R010828.pdf>)
サンプルデータでは、1 行目にダウンロード時刻、2 行目にデータの種類、3 行目が地点名を表すヘッダが入っています。元のデータには経度・緯度情報が含まれていなかつたため、アメダス地点の経度・緯度を 4 ～ 5 行目に追加しました。6 行目以降が降水量データです。pd.read_csv で 3 行目以降を読み込み、DataFrame (df) に格納します。

```
input_file = "20190826-20190828-amedas_prep.csv"
df = pd.read_csv(input_file, parse_dates=[0], index_col=[0], skiprows=[0, 1])
```

read_csv は、時系列データのような時刻情報と観測値の情報を同時に含むようなデータの処理に力を発揮します。例えば、データの 1 列目に ISO 形式の時刻 (2019-07-24T17:22:13.465855)、2 列目にその時刻の観測値が入っているようなデータであれば、時刻とデータが組みになった Pandas の DataFrame が

read_csv の戻り値となります。5.5.4 節で紹介した read_fwf と同様に、read_csv には parse_dates オプションがあり、時刻データとして処理する列を指定できるようになっています。このオプションでは、1 つの列だけ指定する場合に parse_dates=[0] (1 列目)、複数の列を指定する場合には、parse_dates=[[0, 1]] (1 列目と 2 列目) のような記法を用います。例えば 1 列目が年、2 列目が月のデータの場合です。2 列目が月、1 列目が年のように逆順の場合には、parse_dates=[[1, 0]] となります。ここでは 1 列目が ISO 形式の時刻データになっているデータを用いたので、単に parse_dates=[0] としました。

時間軸を index にしておきたい場合、index_col=[0] とします。こうしておけば、後で df.index のような記法で時刻データを取り出すことができ、また plt.plot(df) で折れ線グラフが作図できるなど、データを扱いやすくなります。なお parse_dates を使った場合、元の年、月などのデータは保持されないので、後から年などのデータも使う場合には、keep_date_col=True も与えておきます。

csv データは 1 行目がヘッダ (この値が DataFrame の列の名前となる)、2 行目からデータとなっていることを想定しているので、この形式に従っていない場合は、skiprows オプションで明示しておく必要があります。例えば、1 行目に説明書きなどがされている場合、skiprows=[0] で 1 行目を読み飛ばします。skiprows=[0, 1] のような複数行指定も可能で、サンプルデータは 3 行目がヘッダなので、このようにしています。5.5.4 節で扱った ssw_info.txt のように、ヘッダがなく 1 行目からデータが始まっている場合、header=None を指定します。この場合には列の名前が番号になるので、もし列に名前をつけておきたい場合には、names=[名前 1, 名前 2, , , 名前 n] のようなオプションを使います。

basemap_readcsv.py では、経度、緯度、降水量データを df に格納しました。それぞれのデータを pandas の機能を使って取り出してみます。元データの 3 行目がヘッダで、4 行目が経度データ、5 行目が緯度データ、6 行目から降水量データでした。まず経度データを取り出すため、df.iloc[0, :] のように行番号をスライス記法で指定します。4 行目のデータが df の最初にあるのは、3 行目がヘッダとして扱われたためです。同様に緯度データも取り出します。降水量データは df.iloc[2:, :] のような記法で 3 行目から最後まで取り出します。

```
lons = df.iloc[0,:] # 経度データ取り出し  
lats = df.iloc[1,:] # 緯度データ取り出し  
prep = df.iloc[2:,:] # 降水量(mm)データ取り出し
```

降水量データとその配列サイズを表示させてみます。降水量データの最後に72行、43列のデータであることが表示されます。配列サイズを表示すると(72, 43)のように行が0番目の軸、列が1番目の軸に入っていることが分かります。

```
print(prep)  
print(prep.shape)
```

出力：

```
...  
[72 rows x 43 columns]  
(72, 43)
```

経度・緯度情報も表示させます。df.iloc[0:2,:]で、1～2行目を取り出しています。この状態では経度や緯度が行、地点名が列になっていますが、地点名を行、経度や緯度を列にして表示するため、df.iloc[0:2,:].T（行列の転置操作）を行いました。元の表示が気になる場合は、print(df.iloc[0:2, :])を試してみましょう。

```
print(df.iloc[0:2,:].T)
```

出力：

年月日時	経度	緯度
佐賀	130.305	33.265
...		
前原	130.190	33.560

6.4.3 NetCDF 形式

気象データの配布形式として NetCDF 形式がよく用いられます。この形式は、緯度・経度・高度等の格子点に対応する 3 次元データのように、データ容量の大きなものに用いられており、データと一緒に格子点の情報やデータの説明も一緒に格納されているのが特徴です。データの説明も格納できるため、データ作成者や作成に用いたモデルの名前、変数の導出方法、参照元の文献や URL などを記述しておけば、利用者がデータの中身を理解しやすくなります。また、格納されたデータを機種に依存することなく取り出せるため、データの配布に適しています。GrADS 等の作図アプリケーションで直接表示できるという利点もあります。NetCDF 形式では、データ本体と一緒にデータに掛けるファクターやデータに足すオフセット値も格納しておくことができます。そのため、元は 4 バイトの浮動小数点データ（単精度）であったとしても、データ格納時にファクターで割りオフセットを引いてから 2 バイトの整数にすることで、データサイズを圧縮して配布することも可能となっています。データ取り出し時には、ファクターを掛けオフセットを足して元のデータを復元します。この処理では、有効桁数は保たれませんが、それが問題とならないようなデータの配布には使われます。なお、気温や風速のようにデータ範囲が数桁の範囲内に限られているものには適用できますが、物質分布のように大きく桁が異なるデータが一緒に入っている場合、この方法で圧縮すると元のデータを復元することが難しくなります。

python で NetCDF 形式のデータを読み込むには、netCDF4 モジュールを import しておく必要があります。ここでは、京大生存圏データベースで公開されている NetCDF 形式の NCEP 再解析データを利用してデータ読み込みを試してみます (basemap_readnetcdf.py)。このデータを利用して出版物を作成する際には、<http://database.rish.kyoto-u.ac.jp/arch/ncep/> の規約に従って下さい。

まずは urllib ライブライアリを利用して NCEP 再解析データをダウンロードします。5.5.1 節では urllib.request.urlopen('URL') で URL を開いてから操作を行いましたが、ここでは **urllib.request.urlretrieve('URL', 'ファイル名')** を使い、データを直接ダウンロードします。

```

import urllib.request
import netCDF4 # NetCDF ライブライ
url="http://database.rish.kyoto-
u.ac.jp/arch/ncep/data/ncep.reanalysis.derived/surface/slp.mon.mean.nc"
file_name = "slp.mon.mean.nc"
urllib.request.urlretrieve(url, file_name) # データダウンロード

```

ダウンロードが成功すると、slp.mon.mean.nc というファイルが作成されます。月平均した海面更正気圧 (Sea Level Pressure : SLP) データです。ターミナルを開き、データをダウンロードしたディレクトリに移動して、ncdump -h slp.mon.mean.nc を行うと、NetCDF ファイルのヘッダ情報が表示されます。

出力：

```

netcdf slp.mon.mean {
dimensions:
    lat = 73 ;
    lon = 144 ;
    time = UNLIMITED ; // (860 currently)
variables:
    float lat(lat) ;
    ...
    float lon(lon) ;
    ...
    double time(time) ;
    ...
    float slp(time, lat, lon) ;
    ...

```

最初の dimensions の部分は、データの個数を表していて、緯度方向に 73、経度方向に 144、時間方向に 860 個のデータが存在していることが分かります。次の variables の部分は、格納されているデータの名前と型を表していて、変数

名 lat、lon は 4 バイト浮動小数点数、time は 8 バイト浮動小数点数です。括弧内はデータの軸を表していて、これらのデータは軸のデータ（格子点の座標や時刻の情報を表すデータ）なので、変数名と軸の名前は同じになっています。変数名 slp は 4 バイト浮動小数点数で、time、lat、lon の 3 つの軸を参照していて、経度、緯度、時間を軸に持つ 3 次元（空間 2 次元と時間 1 次元）データであることが分かります。

データの個数は netCDF4 モジュールを使って調べることも可能です。プログラム中では、その処理も行なっていて、まずは `netCDF4.Dataset("ファイル名", "r")` を使い、ファイルを開いてデータを読み込みます。プログラム中では、戻り値を nc に格納して `print(nc)` で表示させています。最後に `nc.close()` でファイルを閉じます。

```
nc = netCDF4.Dataset(file_name, 'r') # データ読み込み
print(nc) # ファイルの情報を表示
nc.close() # ファイルを閉じる
```

変数 nc の型が最初に表示された後、ファイルの情報が並んで、最後にデータの個数、格納されている変数名と型、軸の名前が表示されます。

出力：

```
<class 'netCDF4._netCDF4.Dataset'>
... ファイルの情報を表示
dataset_title: NCEP-NCAR Reanalysis 1
dimensions(sizes): lat(73), lon(144), time(860)
variables(dimensions): float32 lat(lat), float32 lon(lon),
float64 time(time), float32 slp(time,lat,lon)
```

データを処理する際には、経度、緯度、時間方向のデータの個数を使いたいので、個別に取得する方法を考えます（`basemap_readnetcdf2.py`）。ファイルを開いた際、戻り値を nc に格納しているので、`netCDF4.Dataset` に含まれるメソッドを利用できる状態です。そのうちの `nc.dimensions()` がデータ数を取得するメソッドですが、このメソッドのみでは余計な情報が含まれます。データ数の

みが取り出されるように、例えば経度方向の場合には、`len(nc.dimensions['lon'])` のようにします。

```
nc = netCDF4.Dataset(file_name, 'r') # データ読み込み
idim = len(nc.dimensions['lon']) # 経度方向のデータ数
jdim = len(nc.dimensions['lat']) # 緯度方向のデータ数
ndim = len(nc.dimensions['time']) # 時間方向のデータ数
datasize = idim * jdim # 水平方向のデータサイズ
```

経度、緯度、時間方向のデータ数 `idim`、`jdim`、`ndim` が個別に取得できたことが分かります。整数値に変換されたので、`idim * jdim` のような演算も可能で、水平方向の `datasize` が正確に表示されます。

```
print("lon =", idim, ", lat =", jdim, ", time =", ndim)
print("datasize = ", datasize)
```

出力：

```
lon = 144 , lat = 73 , time = 860
datasize = 10512
```

それでは、NetCDF ファイルから `nc.variables["変数名"][:]` を使いデータを取り出してみます（`basemap_readnetcdf3.py`）。

```
lat = nc.variables["lat"][:]
lon = nc.variables["lon"][:]
time = nc.variables["time"][:]
var = nc.variables["slp"][:]
```

取り出した `lon`、`lat`、`time`、`var` を表示します。

```
print("lon = ")
print(lon)
print("lat = ")
print(lat)
print("time = ")
print(time)
print("slp data = ")
print(var)
```

出力：

```
lon =
[ 0.    2.5   5.    7.5   10.   12.5  15.   17.5  20.   22.5  25.   27.5
...
330. 332.5 335. 337.5 340. 342.5 345. 347.5 350. 352.5 355. 357.5]
lat =
[ 90.   87.5  85.   82.5  80.   77.5  75.   72.5  70.   67.5  65.   62.5
...
-90. ]
time =
[1297320. 1298064. 1298760. 1299504. 1300224. 1300968. 1301688. 1302432.
...
1922592. 1923336. 1924056. 1924800.]
slp data =
[[1014.25446 1014.25446 1014.25446 ... 1014.25446 1014.25446 1014.25446]
...
[1034.3832 1034.3832 1034.3832 ... 1034.3832
1034.3832 1034.3832 ]]]
```

6.4.4 単純なバイナリ形式 (GrADS 形式)

気象データの配布には用いられていませんが、NetCDF 形式など他の形式で配布されたデータを手元のサーバで変換した際に、単純なバイナリ形式で保存しておくことがあります。この形式は、4 バイト浮動小数点数（単精度）のデータや 8 バイト浮動小数点数（倍精度）のデータを改行コード無しで並べたもので、単精度のものは GrADS の入力データとしてよく使われていることから、GrADS 形式のように呼ばれていることもあります。この形式の欠点としては、経度・緯度・高度などの情報がどこにも含まれておらず、データのレコード毎に区切られてもいないので、それらの情報を別個に持つておく必要があるということが挙げられます。GrADS を使っていた場合には、コントロールファイルを作っていたかと思いますが、そうしたファイルに相当します。

表 6-4-2 は、経度、緯度方向に 10 度の等間隔でデータが存在するバイナリファイルを模式的に表したもので、左側は時刻（ここでは $t=0$ 、 $t=1$ のみ）を表しています。色の付いた部分はデータを表していて、便宜上 | で区切っていますが、0E の 90N、10E の 90N の順に 350E の 90N までデータが連続して並んでいます。その後に連続して 0E の 80N のデータが来るようになっていて、 $t=0$ のデータが全て並んだ後で $t=1$ のデータが同様に並びます。

表 6-4-2 バイナリファイルの形式（色の付いた部分がデータを表す）

$t=0$		0E, 90N		10E, 90N		...		350E, 90N	
$t=0$		0E, 80N		10E, 80N		...		350E, 80N	
...									
$t=1$		0E, 90S		10E, 80N		...		350E, 80N	
$t=1$		0E, 90N		10E, 90N		...		350E, 90N	
$t=1$		0E, 80N		10E, 80N		...		350E, 80N	
...									
$t=1$		0E, 90S		10E, 80N		...		350E, 80N	

バイナリデータを扱う際に知っておく必要があることとして、コンピュータにデータを格納する形式にビックエンディアン (Big Endian) とリトルエンデ

ィアン (Little endian) の 2 種類存在することが挙げられます。ビックエンディアンが用いられるかリトルエンディアンが用いられるかは、CPU に依存しますが、データ共有の目的で保存する場合にはどちらかに合わせておくことが多いでしょう（過去のワークステーションで使われることが多かった SPARC や、スーパーコンピュータで使われることの多いビックエンディアンに合わせることが多いかもしれません）。コンピュータ上のデータは、電子回路上での電圧の高低や信号のオンオフに対応するような 0 か 1 のどちらかの状態で保持されており、それを 1 ビットと呼んでいます。データを扱う際には、最小単位の 1 ビットを 8 個まとめた 1 バイト (8 ビット) 単位で保存されており、2 バイト以上のデータを保存する際にどのような順番で保存するか、をエンディアンと呼んでいます。データの最初から保存するか、データの最後から保存するかで 2 つの流儀があり、ビックエンディアンとリトルエンディアンという名前が付けられています。例えば 4 バイト浮動小数点数であれば、1 バイトの集まりを 4 つ保存する必要があります。データの中身が AABBCCDD (1 バイトの集まり 4 つ、コンピュータ上では 0 と 1 の 32 個の並び) であれば、ビックエンディアンでは AABBCCDD のまま上位から保存されますが、リトルエンディアンでは DDCCBBA のように下位から保存されます。浮動小数点数データは、上位に符号や指数が入ることになっているので、もしエンディアンを間違えて読んでしまうと、明らかに桁のおかしな値に変換されるので、データを扱う時にそのような場面に出くわしたら、エンディアンを確認してみましょう。

先ほどの NCEP 再解析データを一旦バイナリ形式に変換し、そのデータを読み込むプログラムが basemap_nc2bin.py です。まずは、`np.array(var).tofile("ファイル名")` でバイナリファイルを書き出します。書き出される形式がビックエンディアンかリトルエンディアンのどちらになるのかは、プログラムを実行したマシンに依存します。ここで `np.array(var)` は Numpy の ndarray になっていて、`ndarray` にはファイルに書き出す `tofile` というメソッドがあるため、このような操作が可能になっています（表 6-4-3）。なお古いバージョンでは、`var = nc.variables["slp"][:]` で取り出した `var` で `var.tofile` が利用できましたが、netCDF4 モジュールのバージョン 1.5 以降ではエラーが出るようになりました。

```
np.array(var).tofile("output.bin") # バイナリファイル書き出し  
print(len(var))
```

表 6-4-3 Numpy の主なファイル読み書き (arr は Numpy の ndarray)

用途	書き出し	読み込み
バイナリ形式、非圧縮の読み書き	arr.tofile("ファイル名")	np.fromfile("ファイル名", dtype='形式')
テキスト形式、非圧縮の読み書き	np.savetxt("ファイル名", arr)	np.loadtxt("ファイル名")

バイナリファイルを読み込む際には、`np.fromfile("ファイル名", dtype='<f4')` を用います。戻り値は ndarray になっていて、それを `din` に入力しています。オプションとしてバイナリデータの形式を指定することができ、リトルエンディアンの 4 バイト浮動小数点数であれば、`dtype='<f4'` です。先ほどと同じ値が表示されたと思います。

```
din = np.fromfile("output.bin", dtype='<f4') # バイナリファイル読み込み
slp = din.reshape(ndim, jdim, idim) # データサイズを合わせる
print("slp data (nc) = ")
print(slp)
```

出力：

```
slp data (nc) =
[[[1014.25446 1014.25446 1014.25446 ... 1014.25446 1014.25446 1014.25446]
...
[1034.3832 1034.3832 1034.3832 ... 1034.3832
1034.3832 1034.3832 ]]]
```

ビックエンディアンの場合を試してみます。オプションは `dtype='>f4'` です。明らかに桁のおかしな値が表示されました。

```
din = np.fromfile("output.bin", dtype='>f4') # バイナリファイル読み込み
```

出力：

```
slp data (nc) =  
[[ 1.18365650e+06  1.18365650e+06  1.18365650e+06 ...  1.18365650e+06  
  1.18365650e+06  1.18365650e+06]  
[ 1.91327550e+31  1.83933320e-34  1.77614273e+19 ... -7.25709291e+28  
  -1.04009102e+15  3.48502827e-09]  
...  
[ 2.04504944e+02  2.04504944e+02  2.04504944e+02 ...  2.04504944e+02  
  2.04504944e+02  2.04504944e+02]]]
```

np.fromfile で読み込んだデータは 1 次元の配列になっています。作図の際には経度、緯度、時間方向に整理された空間 2 次元、時間 1 次元の配列の方が使いやすいため、`data.reshape(データの形状)`を使ってデータ形状を変更しました。

最後に Numpy で利用可能なデータ型について表 6-4-4 に、バイナリデータ読み書きの書式一覧を表 6-4-5 にまとめておきます。データを読み込む際には、`d = np.fromfile("ファイル名", dtype='書式')`のようにデータに対応するコードを設定します。データを書き出す際にも Numpy を使うことができ、`np.array(d).astype('書式').tofile("ファイル名")`のように ndarray の astype メソッドを使って指定したデータ型に変換した上で、tofile メソッドでファイルに書き出します。

表 6-4-4 Numpy で利用可能なデータ型

データ型	型コード	説明
int8	i1	符号あり8ビット整数型
int16	i2	符号あり16ビット整数型
int32	i4	符号あり32ビット整数型
int64	i8	符号あり64ビット整数型
uint8	u1	符号なし8ビット整数型
uint16	u2	符号なし16ビット整数型
uint32	u4	符号なし32ビット整数型
uint64	u8	符号なし64ビット整数型
float16	f2	半精度浮動小数点型（符号部1ビット、指数部5ビット、仮数部10ビット）
float32	f4	単精度浮動小数点型（符号部1ビット、指数部8ビット、仮数部23ビット）
float64	f8	倍精度浮動小数点型（符号部1ビット、指数部11ビット、仮数部52ビット）
float128	f16	四倍精度浮動小数点型（符号部1ビット、指数部15ビット、仮数部112ビット）
complex64	c8	複素数（実部・虚部がそれぞれfloat32）
complex128	c16	複素数（実部・虚部がそれぞれfloat64）
complex256	c32	複素数（実部・虚部がそれぞれfloat128）
bool	?	ブール型（True or False）
unicode	U	Unicode文字列
object	O	Pythonオブジェクト型

Numpy で `dtype=np.'データ型'` のように用いる

表 6-4-5 Numpy バイナリデータ読み書きの書式一覧

書式	データ型	エンディアン	説明
>f2	float16	big endian	半精度浮動小数点型 (2バイト)
<f2	float16	little endian	半精度浮動小数点型 (2バイト)
>f4	float32	big endian	单精度浮動小数点型 (4バイト)
<f4	float32	little endian	单精度浮動小数点型 (4バイト)
>f8	float64	big endian	倍精度浮動小数点型 (8バイト)
<f8	float64	little endian	倍精度浮動小数点型 (8バイト)
>i2	int16	big endian	符号あり16ビット整数型 (2バイト)
<i2	int16	little endian	符号あり16ビット整数型 (2バイト)
>i4	int32	big endian	符号あり32ビット整数型 (4バイト)
<i4	int32	little endian	符号あり32ビット整数型 (4バイト)
>i8	int64	big endian	符号あり64ビット整数型 (8バイト)
<i8	int64	little endian	符号あり64ビット整数型 (8バイト)
>u2	uint16	big endian	符号なし16ビット整数型 (2バイト)
<u2	uint16	little endian	符号なし16ビット整数型 (2バイト)
>u4	uint32	big endian	符号なし32ビット整数型 (4バイト)
<u4	uint32	little endian	符号なし32ビット整数型 (4バイト)
>u8	uint64	big endian	符号なし64ビット整数型 (8バイト)
<u8	uint64	little endian	符号なし64ビット整数型 (8バイト)

読み込む場合 : d = np.fromfile("ファイル名", dtype='書式')

書き出す場合 : np.array(d).astype('書式').tofile("ファイル名")

单精度浮動小数点型のコードは f4 を f と省略可能

6.4.5 GRIB 形式

GRIB 形式は WMO が定めたバイナリデータの交換形式で、データを圧縮して格納することができ、格納されたデータを機種に依存することなく取り出せます。GRIB 形式でも、データと一緒に格子点の情報や変数の簡単な説明等も格納されています。1989 年に制定された第 1 版 (GRIB1) と 2001 年に制定された第 2 版 (GRIB2) があり、両者に互換性はありません。現在では GRIB2 が主流になっていますが、過去の配信データや客観解析データ等には GRIB1 で提供されているものもあります。なお、GRIB 形式について詳細が知りたい場合には、次の Qiita にある解説や気象庁の資料が参考になります。

https://qiita.com/e_toyoda/items/ce7497e1a633b16f1ff1

<https://www.jma.go.jp/jma/kishou/books/nwpreport/63/chapter4.pdf>

Python では、バージョン 3.6 までは pygrib というモジュールがあり、GRIB2 形式のファイルを直接読むことができましたが、バージョン 3.7 以降では廃止されました。しかし 2 章で導入した wgrib2 を使えば、GRIB2 形式のファイルを NetCDF 形式のファイルに変換することができるため、6.4.3 節の方法で読むことが可能です。GRIB2 形式のファイルを NetCDF 形式のファイルに変換する際には、次のように行います。具体的な例は 6.6.3 節に載せました。

% wgrib2 入力 GRIB2 ファイル.grb -netcdf 出力 NetCDF ファイル.nc

なお GRIB1 形式の場合には、python で直接読むことはできませんが、GrADS に付属している wgrib コマンドを使うことで、単純バイナリファイルに変換可能です。例えば、次のようなコマンドで変換を行います。

```
% wgrib -v 入力ファイル.grb | grep ":GRIBID を記述," | sort -nr -k5 -t':' \
| wgrib -i -nh -ieee 入力ファイル.grb -o 出力ファイル.bin
```

最初の wgrib は、ファイルの中に含まれる変数をリスト化するためのものです。2 回目の wgrib は、リストを読み込んで入力ファイルからリストに対応するデータをファイルに書き出すためのものです。例えば、JRA-55 客観解析データで 2020 年 12 月 20 日 00UTC の東西風データを開きリストを表示させてみます。

```
% wgrib -v anl_p125_ugrd.2020122000
```

出力：

```
Undefined parameter table (center 34-241 table 200), using NCEP-opn
1:0:D=2020122000:UGRD:1 mb:kpds=33,100,1:anl:winds are N/S:"u wind [m/s]
2:62748:D=2020122000:UGRD:2 mb:kpds=33,100,2:anl:winds are N/S:"u wind [m/s]
3:125496:D=2020122000:UGRD:3 mb:kpds=33,100,3:anl:winds are N/S:"u wind [m/s]
...
37:2258928:D=2020122000:UGRD:1000 mb:kpds=33,100,1000:anl:winds are N/S:"u wind [m/s]
```

時刻 (2020122000) や変数名 (UGRD)、高度 (1 mb)、GRIBID、データの説明 (1:anl:winds are N/S:"u wind [m/s]) が表示されます。GRIBID はデータ毎に付けられている ID のことで、取り出したいデータの共通部分を grep コマンドで検索してリストから取り出します。このケースのように、複数の気圧面に分かれたデータを結合したい場合などには、grep ":kpds=33,100,"のような共通部分を検索します。下層を先にしたい場合には、sort -nr -k5 -t':' (-nr は数字で逆順に並べる、-k5 は 5 列目、-t':'は列の区切り文字が:) とし、リストを 5 列目の気圧の値を使って並べ替えます。2回目のwgribでデータを取り出す際には、ヘッダ無し(-nh)、ビックエンディアン(-ieee)のオプションを指定し、ビックエンディアンの単純バイナリファイルに変換します。複数の入力ファイルの出力をまとめたい場合には、-append オプションを使います。

Python3.6 の場合には、以下のように pygrib を使ったデータの読み込みが可能です。import pygrib でインポートした後、GRIB2 形式のデータを読む際には、**pygrib.open("ファイル名")**を使います。戻り値を格納した grbs には、データを取り出すための select メソッドがあり、forecastTime に予報時刻からの時間を、スライスに変数の番号を与えます。ここでは 1 時間後 (forecastTime=1) の 0 番目の変数 ([0]) としました。

```
import pygrib
grbs = pygrib.open("sample.grb") # GRIB2 ファイルを開く
grb1 = grbs.select(forecastTime=1)[0] # 1 時間後の 0 番目の変数
```

6.4.6 HDF 形式

HDF 形式は、米国立スーパーコンピュータ応用研究所 (National Center for Supercomputing Applications : NCSA) で開発され、大量のデータを格納し構造化するために設計されています。HDF 形式でも、データと一緒に格子点の情報や変数の説明なども格納することができ、格納されたデータを機種に依存することなく取り出せます。HDF 形式には、古い HDF4 形式と新しい HDF5 形式があり、両者に互換性はありませんが、どちらの形式も非営利法人である HDF グループによって現在もサポートされています。HDF5 はファイル構造が単純化されており、ディレクトリに相当するグループとファイルに相当するデータセットの 2 種類で階層構造を持って保存することができます。また、データと画像ファイル、データの説明のテキストなど、複数の種類のデータをまとめて格納しておくことが可能で、データ圧縮に対応することもできます。観測時刻、軌道の位置、観測場所の緯度経度と観測データ及びデータ品質の情報など、複数の情報をまとめて保存できることから、衛星観測データの配信にしばしば用いられています。

Python では、h5py モジュールや Pandas の pd.read_hdf("ファイル名") で HDF5 形式のファイルを読むことができます。なお、pd.read_hdf() で読めない形式の HDF5 ファイルがあり、その場合には h5py モジュールを使います。h5py モジュールは、import h5py でインポートでき、HDF5 形式のデータを読む際には、**h5py.File("ファイル名", "r")** とします。戻り値を格納した hdf に入っている kdf.keys() を表示させると、データセットの名前を取得することができます。

```
import h5py
hdf = h5py.File("sample.h5", "r") # HDF5 ファイルを開く
print(hdf.keys())
d = hdf['dataset_name'][ :, :] # データセットの名前に対応するデータを取得
```

格納されているデータが 2 次元の場合、hdf['データセットの名前'][:, :] でデータセットの中身を取得することができます。以前は hdf['データセットの名前'].value を使うことができましたが、現在のバージョンではエラーが出てします。

6.5 気象データを用いた作図

読み込んだデータを使い、等高線や陰影、矢羽、矢印などの作図を行います。作図を行う際には、読み込んだデータを Basemap で使用できる形式の 2 次元データに格納する必要があります。

6.5.1 2 次元データの準備

Basemap では、Numpy の ndarray 形式になった 2 次元データを使用可能です。先ほど 6.4.4 節でバイナリファイル output.bin の読み込みを行なったので、同じデータを使って試してみましょう。バイナリファイルを読み込み、正距円筒図法の地図に重ねて描くプログラムが basemap_contour.py です。ここでは、バイナリファイルのうち最初の 1 ヶ月分のデータを読み込んでみます。6.4.3 節で調べたように、変換元の SLP データには、経度方向に 144、緯度方向に 73 のデータ数がありました。そのため、水平方向のデータ数 (datasize) は 144×73 となります。np.fromfile で読み込む際、count オプションで読み込むデータサイズを指定することができます。count=datasize とすることで、最初の 1 ヶ月分のデータを読み込むことが可能です。ここで count に与えるのはバイト数ではなく、データ数であることに注意が必要です。読み込んだデータを din.reshape を使い 2 次元配列にしています。作図の際には、このように作成された配列 slp を使います。なお、reshape (jdim, idim) のように緯度方向が 0 番目の軸、経度方向が 1 番目の軸になるように変換を行います。python では後のインデックスが内周になるため、このように配列を設定しています。Fortran では前のインデックスが内周になっており逆なので、Fortran を使っている場合、戸惑うことがあるかもしれません。

```
idim = 144
jdim = 73
datasize = idim * jdim
din = np.fromfile("output.bin", dtype='<f4', count=datasize)
slp = din.reshape(jdim, idim) # 2 次元配列にする
print(slp.shape) # 配列のサイズ
print(slp.min(), slp.max()) # 最小値、最大値
```

出力：

```
(73, 144)  
984.89026 1037.519
```

最後に配列のサイズと最小値と最大値を表示させています。最小値と最大値を計算する統計処理のメソッドは、表4-2-1を参照してください。配列のサイズは設定したように(73, 144)の2次元配列になっており、緯度方向が1次元目、経度方向が2次元目です。次の出力は最小値と最大値ですが、この範囲で等高線を描けるように980～1040の範囲を用います。

6.5.2 等高線の作図

等高線を作図するため、まずは地図を作成しておきます。ここでは正距円筒図法を用います。

```
m = Basemap(projection='cyl', llcrnrlat=-90, urcrnrlat=90, llcrnrlon=0,  
             urcrnrlon=360, resolution='l') # 正距円筒図法  
m.drawcoastlines(linewidth=0.2, color='k') # 海岸線を描く
```

等高線を描く際には、等高線を描く2次元配列に加えて、経度、緯度方向の座標データが必要となります。ここでは経度、緯度方向に等間隔のデータを使っているので、プログラム内で設定します。経度、緯度方向の座標も、等高線のデータと合わせて2次元データとしておく必要があり、[np.indices\(緯度方向のデータ数, 経度方向のデータ数\)](#)を使い作成します。生成されたデータは3次元データになっており、1次元目がデータ、2次元目が緯度、3次元目が経度です。1次元目は0番目の要素が緯度方向の座標データ、1番目の要素が2次元の経度方向の座標データに対応しており、それぞれ[0, :, :]、[1, :, :]のスライス記法を使って取り出し、lats、lonsに格納します。そのため、lats、lons共に1次元目が緯度、2次元目が経度の2次元データです。np.indicesでは座標番号を生成するので、lats、lonsにする際に、座標番号から度への変換を行います。プログラムで読み込むoutput.binには、緯度方向には北極が先、南極が後の順で格納されているため、latsの計算では90.-delta*np.indices(..)として、北緯90度から降順になるようにしています。

経度、緯度データそのものを用いると、図法によっては正確な場所に配置されないため、図法に対応した値に変換する処理が必要です。変換処理は、Basemap を呼び出した時に生成されたインスタンス m を使い、m(経度, 緯度) で行います。ここではラジアンではなく度を用います。実際は、正距円筒図法の場合には変換を行わなくとも同じなのですが、他の図法に変えた場合にも適用可能なように変換処理を加えています。

```
nlats = jdim
nlons = idim
delta = 360. / (nlons - 1)
lats = (90.-delta*np.indices((nlats, nlons))[0, :, :]) # 緯度座標
lons = (delta*np.indices((nlats, nlons))[1, :, :]) # 経度座標
# 図法の経度、緯度に変換する（入力データの単位は度）
x, y = m(lons, lats)
```

等高線を描くには **m.contour** を用います。SLP の値は 980～1040 hPa の範囲だったので、その範囲で 4 hPa 毎に等高線を引きます。m.contour の引数は 1 番目が経度 (x)、2 番目が緯度 (y)、3 番目が等高線のデータ (slp) で、4 番目が等高線を描く値 (clevs) です。等高線を描く値は levels=clevs と書くこともできます。他には等高線を黒色にする colors='k'、等高線の幅を指定する linewidths=0.8 オプションを与えています（デフォルト値：1.0）。

```
clevs = np.arange(980, 1040, 4)
m.contour(x, y, slp, clevs, linewidths=0.8, colors='k')
```

作成されたものが図 6-5-1 です。

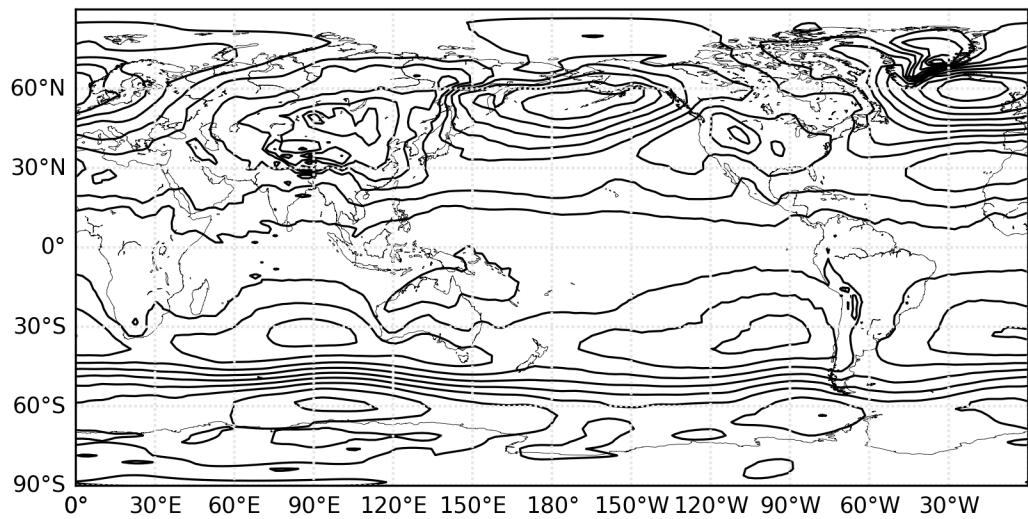


図 6-5-1 NCEP 再解析データを用いて SLP 等高線を描いた(1948 年 1 月の月平均値)

アリューシャン低気圧とシベリア高気圧付近で等高線が混んでいるのが読み取れます、等高線だけではどこが高気圧か低気圧か判別できないので、気圧の値を表示できるようにします(図 6-5-2)。作図には basemap_contour2.py を用いました。等高線を作成する m.contour の戻り値を cs に格納しておきます。cs がインスタンスになっており、**cs.clabel()**で等高線にラベルを付けることが可能ですが。ここでは、オプションとして文字サイズを変更する fontsize、等高線のフォーマットを指定する fmt を使いました。fmt に与える書式は、4.5.7 節や 5.2.4 節で出てきた"%d"(整数)のような書式を用います(表 5-2-2 参照)。

```
cs = m.contour(x, y, slp, clevs, linewidths=0.8, colors='k')
cs.clabel(fontsize=12, fmt="%d") # ラベルを付ける
```

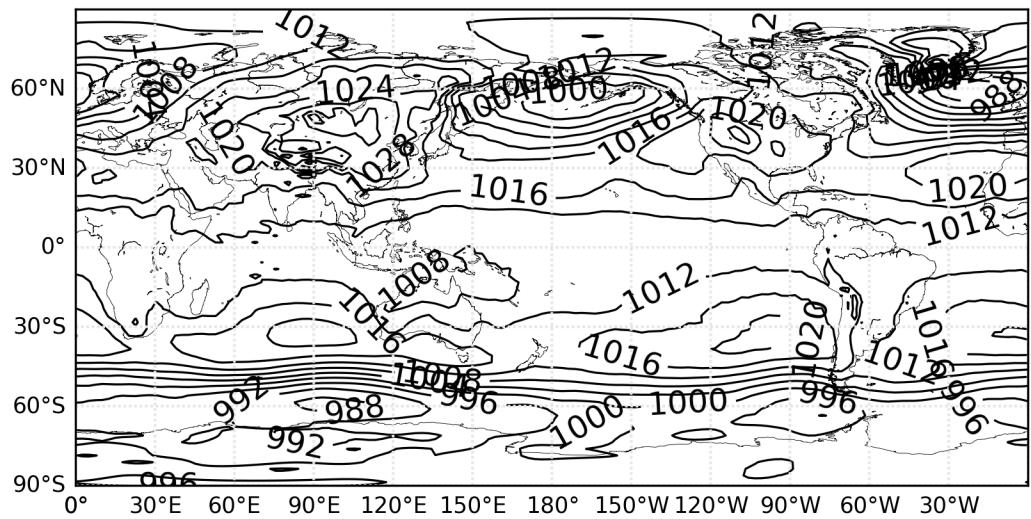


図6-5-2 等高線にラベルを付けた

少しラベルが混んでいるので、20 hPa 毎にラベルを付けてみます（図6-5-3）。作図には basemap_contour3.py を用いました。cs.levels に等高線を描いた時の値が格納されているので、それを clevels として取り出します。cs.clabel では 1 番目の引数としてラベルを描く値を取ることができます。clevels[::5]のように 5つ飛ばしの値にすることで、等高線を描いた $4 \text{ hPa} \times 5 = 20 \text{ hPa}$ の場所にラベルを描くことが可能です。なお[::5]はスライスの記法で最初の 2 つのコロンは開始点、終了点がデータの最初と最後であることを表し、3 つ目が step を表し 5 つ飛ばしにすることを意味します。

```
cs = m.contour(x, y, slp, clevs, linewidths=0.8, colors='k')
clevels=cs.levels # 等高線の値を取り出す
cs.clabel(clevels[::5], fontsize=12, fmt="%d") # ラベルを付ける
```

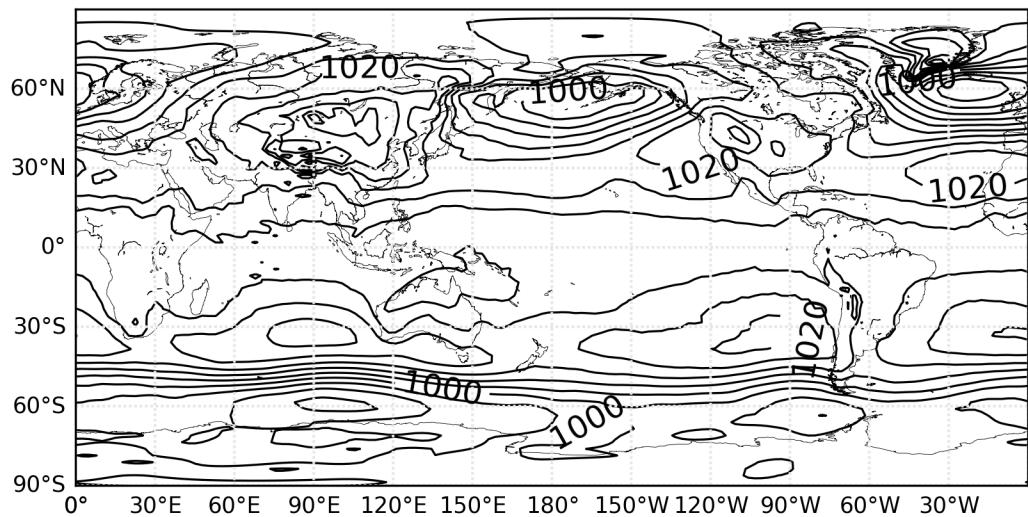


図 6-5-3 等高線のラベルの間隔を 20 hPa 毎に

ここまで の方法では、最初の 1 ヶ月分のデータを読み込むことはできましたが、例えば、同じデータの 7 番目の場所に入っている 7 月のデータを作図することはできません。いくつか方法があるのですが、まずは簡単な方法として、7 番目のデータまで読み込んでから作図する方法を紹介します。作図には basemap_contour4.py を用いました。読み込むデータ番号 `rec=7` として、ファイルの先頭から読み込むデータ数を `count=datasize * rec` のように 7 倍します。読み込んだデータは空間 2 次元 + 時間の 3 次元データになったので、`slp=din.reshape(rec, jdim, idim)` で時間方向の次元を追加した 3 次元配列に変換します。こうしておくことで、`slp[データ番号, :, :]` で利用したい時刻の空間 2 次元データを取り出すことが可能となります。なおデータ番号は 0 から始まるので、7 月の場合は 6 とする必要があります。

```

idim = 144
jdim = 73
rec = 7 # 読み込むデータ番号
datasize = idim * jdim
din = np.fromfile("output.bin", dtype='<f4', count=datasize*rec)
slp = din.reshape(rec, jdim, idim) # 3次元配列にする

```

作図部分では、`slp[rec-1,:,:]`で7月のデータを渡します。作成されたものが図6-5-4です。

```
cs = m.contour(x, y, slp[rec - 1, :, :], clevs, linewidths=0.8, colors='k')
```

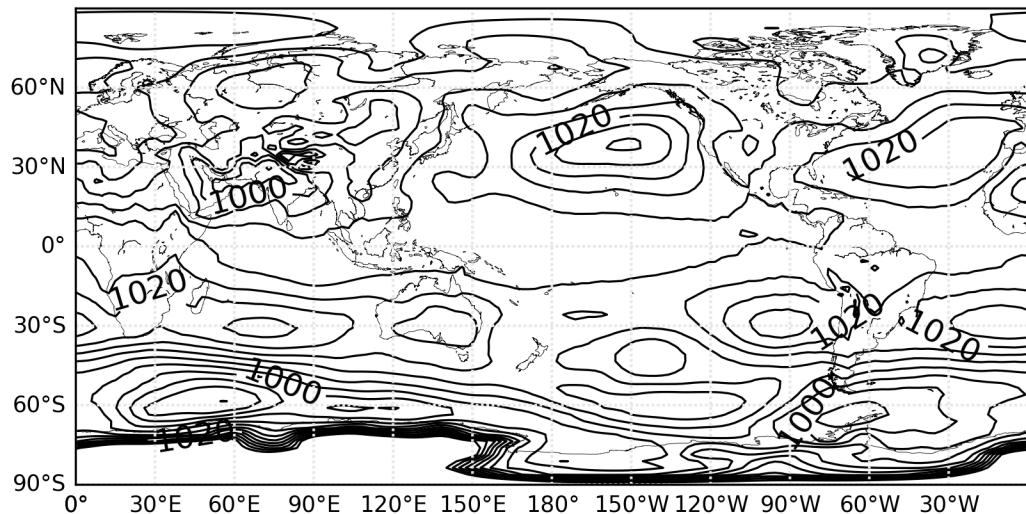


図6-5-4 1948年7月のSLP月平均値を描いた

この方法では、バイナリファイルの先頭から `count` に与えたデータ数分だけ `din` に格納されるので、その中のどの場所のデータでも読み込むことが可能です。最初にバイナリファイル全体を読むように変えれば、バイナリファイルの任意の場所のデータが取り出せるようになります。バイナリファイル全体を読むには、`count` に与える `datasize * rec` の代わりに、`datasize * num_rec` (`num_rec` は時間方向のデータ数) を使います。データ全体の統計量を計算したい場合などに便利です。

バイナリファイル全体を読んでから1月平均のSLPを作図してみます(図6-5-5)。作図には `basemap_contour5.py` を用いました。先ほど 6.4.3 節で NetCDF データを読み込んだ際には、時間方向のデータ数は 860 でした。そこで `num_rec=860` に設定しています。

```

idim = 144
jdim = 73
num_rec = 860 # 時間方向のデータ数
datasize = idim * jdim
din = np.fromfile("output.bin", dtype='<f4', count=datasize*num_rec)
slp = din.reshape(num_rec, jdim, idim) # 3次元配列にする

```

次に1月平均を行い、読み込んだ配列を3次元配列にした後と平均後の配列サイズを出力しています。読み込み後には水平方向の144、73のデータ数に時間方向の860のデータ数が入った3次元配列であったものが、144、73のデータ数になっており、時間方向の次元が落ちたことが分かります。まず `slp[::12, :, :]` の部分は、`slp[時間, 緯度, 経度]` のように1番目の要素が時間、2番目と3番目が緯度、経度方向のデータ番号を表します。1月のデータを切り出すため、スライス記法で「`::12`」（データの最初から最後まで12飛ばしに切り出す）のように記述しています。もし2月の場合には、「`1:12`」（データの2番目から最後まで12飛ばしに切り出す）です。切り出したデータを時間方向に平均するため、`mean(axis=0)`メソッドを使いました。`mean`はこれまで出てきた算術平均を行うNumpyのメソッドで（表4-2-1参照）、`axis=軸番号`、を指定すると、その軸に対してのみ平均操作を行います。ここでは軸番号0が時間方向に対応するので、時間平均となります。

```

slp_jan = slp[::12, :, :].mean(axis=0) # 1月平均の計算
print(slp.shape)
print(slp_jan.shape)

```

出力：

```

(860, 73, 144)
(73, 144)

```

計算した1月平均値を等高線で描きます。

```

cs = m.contour(x, y, slp_jan, clevs, linewidths=0.8, colors='k')

```

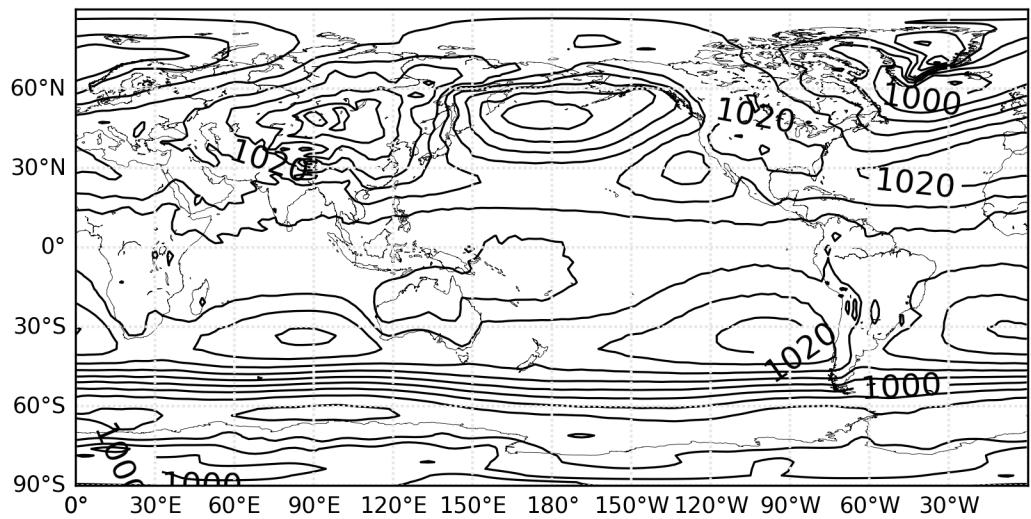


図 6-5-5 1948 年から 2019 年までの 1 月 SLP を平均した

なお、バイナリファイルのサイズが小さい時にはここで紹介した方法で問題ないのですが、巨大なバイナリファイルを読み込む場合には、大量のメモリを消費してしまいます。少し複雑になりますが、そうした場合にバイナリファイルを部分的に読み出す方法を 6.5.5 節に載せています。

6.5.3 陰影の作図

ここまでではバイナリファイルから作図する方法でしたが、6.4.3 節で紹介したように NetCDF ファイルは直接読み込むことが可能です。ここでは、既にダウンロードしてある slp.mon.mean.nc を読み込み、7月の平均的な SLP を等高線で、2018年7月の SLP が平均値からどれだけ離れていたかという偏差を陰影で描いてみます。2018年7月は、4.1 節で紹介したように、中旬から下旬にかけて高温が続き各地で真夏日や猛暑日の日数が更新されるなど猛暑となっていました。

まず NetCDF ファイルを読み込み、7月の平均値を等高線で作図してみます (basemap_contour6.py)。ファイルの読み込みには、netCDF4.Dataset を使います。

```
# NetCDF データの読み込み
file_name = "slp.mon.mean.nc"
nc = netCDF4.Dataset(file_name, 'r')
```

先ほど行ったように、nc.variables を使い変数を取り出します。戻り値の配列は、NetCDF ファイルに格納された時の次元になっています。確認のため、lon、lat、slp の次元を書き出してみます。lon、lat は 1 次元で経度方向のデータ数 144 と緯度方向のデータ数 73 になっており、slp は 3 次元で(860, 73, 144) のデータ数 (時間方向、緯度方向、経度方向の順) になっています。

```
# 変数の読み込み
lon = nc.variables["lon"][:] # 経度
lat = nc.variables["lat"][:] # 緯度
time = nc.variables["time"][:] # 時刻
slp = nc.variables["slp"][:] # SLP データ
nc.close() # ファイルを閉じる
print("lon:", lon.shape)
print("lat:", lat.shape)
print("slp:", slp.shape)
```

出力：

```
lon: (144,)  
lat: (73,)  
slp: (860, 73, 144)
```

次に7月のSLP平均値を計算します。全期間なら `slp[6:12, :, :].mean(axis=0)` となります。ここでは気象庁の平年値と同じ計算方法を行ってみます。気象庁では、2018年の偏差の計算に用いる平年値は1981～2010年の30年間の平均値を用いているので、これに合わせて平年値からの偏差を計算します。スライス記法では、「開始点：終了点：ステップ」の順で記述すると、開始点から終了点の1つ前までステップ数だけ飛ばして切り出すことになっていました。開始点を1981年7月にして終了点を2010年8月以降（ここでは12月）にし、ステップを12にしておけば、1981～2010年の30年間の7月のデータが取り出されます。これらのデータを時間方向（axis=0）に算術平均することで、7月の平年値になります。計算した `slp_jul` の配列サイズを表示すると、確かに(73, 144)のサイズになっています。

```
tstr = (1981 - 1948) * 12 + 6 # 開始点：1981年7月  
tend = (2010 - 1948 + 1) * 12 # 終了点：2010年の12月  
slp_jul = slp[tstr:tend:12, :, :].mean(axis=0) # 7月のSLP平年値  
print("slp_jul:", slp_jul.shape)
```

出力：

```
slp_jul: (73, 144)
```

1次元の経度、緯度データを等高線の作成時に必要な2次元メッシュデータに変換するため、`np.meshgrid` を使っています。変換後の `lons, lats` の配列サイズを表示すると、(73, 144)になりました。最後に図法の経度、緯度に変換しています。NetCDFファイルに格納されていた経度、緯度の単位は度であったので、単位変換しないまま用いています。

```

# 緯度・経度座標の準備
lons, lats = np.meshgrid(lon, lat)
print("lats:", lats.shape)
print("lons:", lons.shape)
# 図法の経度、緯度に変換する (NetCDF データの経度・緯度の単位は度)
x, y = m(lons, lats)

```

出力：

```

lats: (73, 144)
lons: (73, 144)

```

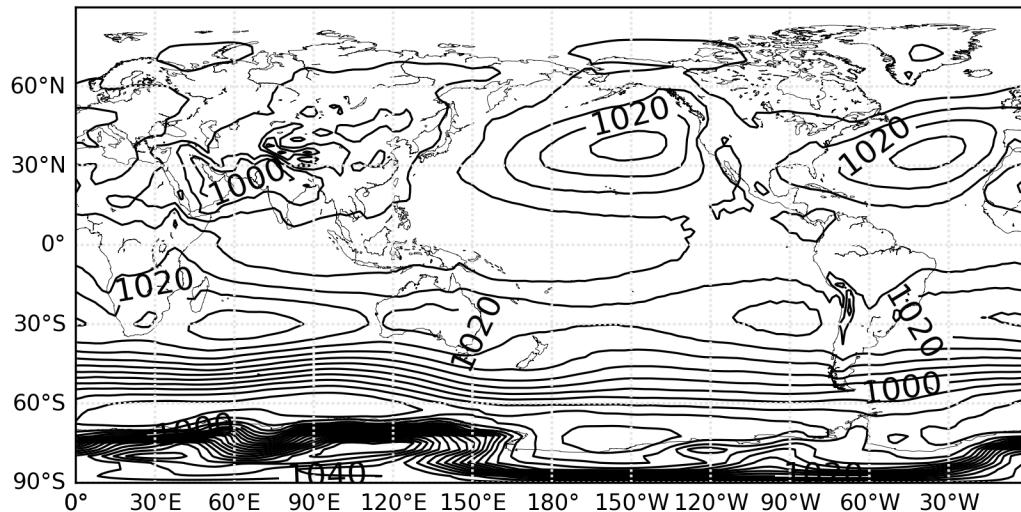


図 6-5-6 1981 年から 2010 年までの 30 年間の 7 月 SLP を平均した

ここで、SLP の等高線を 4 hPa 每、ラベルを 20 hPa 毎に付けるために、`clevs` を作成するときに次のような工夫をしています。等高線を付ける値は、`np.arange(最小値、最大値、ステップ)` で設定します。最大値は、データの最大値の天井（最大値よりも大きい最小の整数）よりも 1 大きい数にしています。最小値は少し複雑で、データの最小値の床（最小値以下の最大の整数）から最小値を 20 で割った余りを減ることで、必ず 20 の倍数になるようにしています。こうすることで、等高線を付ける値が 980 や 960 など 20 の倍数から始ま

るため、clevels[::-5]のように4 hPa毎のラベルを5飛ばしで切り出した時に、必ず20の倍数にラベルが付くようになります。この計算に用いているNumpyの数学関数は表3-5-5を参照してください。

```
clevs = np.arange(np.floor(slp_jul.min() - np.fmod(slp_jul.min(), 20)), ¥  
                  np.ceil(slp_jul.max()) + 1, 4) # 等高線を描く値  
cs = m.contour(x, y, slp_jul, clevs, linewidths=0.8, colors='k') # 等高線を描く  
clevels=cs.levels # 等高線の値を取り出す  
cs.clabel(clevels[::-5], fontsize=12, fmt="%d") # ラベルを付ける
```

等高線を描けたので、2018年7月の偏差を計算し陰影で重ねてみましょう(basemap_contourf.py)。2018年7月の偏差は、次のように計算できます。SLPデータの対応する時刻データを取り出し、7月の平年値を引くことで偏差になります。

```
n = (2018 - 1948) * 12 + 6 # 2018年7月  
slp_anom = slp[n, :, :] - slp_jul # 偏差の計算
```

陰影の作図部分です。陰影を描くにはm.contourfを用います。最初の4つの引数は等高線を描くm.contourと同じで、1番目が経度(x)、2番目が緯度(y)、3番目が等高線のデータ(slp_anom)で、4番目が等高線を描く値(clevs)です。陰影を描く値はcmapオプションで与えます。等高線を描く値は、正が赤、負が青となるように領域の範囲の中央を0にしています。またslp_anomの全ての値が範囲内に収まるように、データの最大値の天井とデータの最小値の床の絶対値をそれぞれ取り、そのうちの大きな方をデータ範囲の端を決める値workとして用います。clevsの計算では(-(work+1), work+1)のように両端をworkより絶対値が1大きい値にしました。色テーブルの名前は、図4-5-7に載せている名前で指定します。ここでは"bwr"を使いました。

```
work = max(np.abs(np.floor(slp_anom.min())), np.abs(np.ceil(slp_anom.max())))  
clevs = np.arange(-(work+1), work+1)  
m.contourf(x, y, slp_anom, clevs, cmap="bwr") # 陰影を描く
```

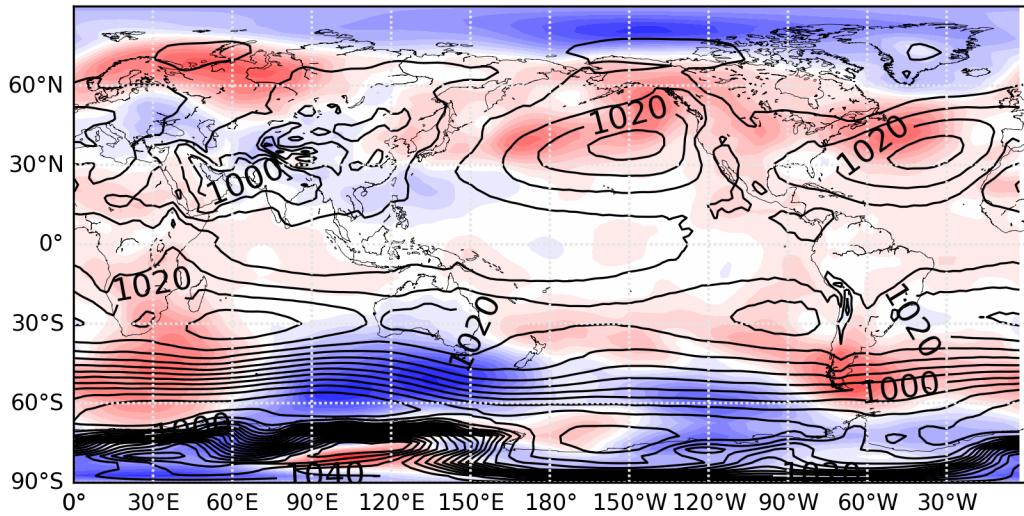


図 6-5-7 2018 年 7 月の SLP を陰影で重ねた

日本付近から東の海上にかけて全体的に高圧偏差となっていることが分かります。2018 年 7 月に太平洋高気圧の張り出しが強かったことに対応するような偏差です。この図では陰影の基準が分からないので、カラーバーを付けたいと思います。カラーバーを付けるには、5.3.3 節で出てきた plt.colorbar を使います。オプションは表 5-3-1 のものを使うことができ、サイズを小さくするため shrink=0.8、水平のカラーバーにするため orientation='horizontal'を付けています。作図に用いたプログラムは、basemap_contourf2.py です。ラベルも同様に cbar.set_label です。ここで色テーブルの指定の方法を変えてみました。描かれる陰影は同じですが、plt.get_cmap("bwr")で生成したインスタンスを使い陰影を付けます。

```

cmap = plt.get_cmap('bwr') # 色テーブル取得
m.contourf(x, y, slp_anom, clevs, cmap=cmap) # 陰影を付ける
cbar = plt.colorbar(shrink=0.8, orientation='horizontal') # カラーバーを付ける
cbar.set_label('SLP anom. in Jul. 2018', fontsize=14) # ラベルを付ける

```

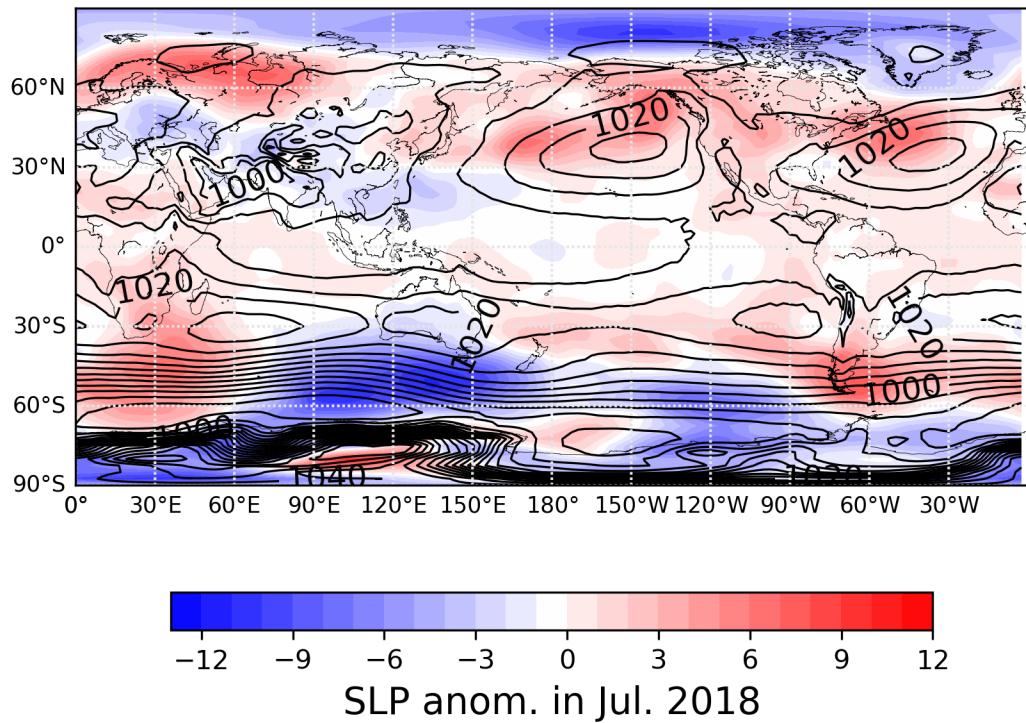


図 6-5-8 カラーバーを付けた

カラーバーとグラフの間が離れすぎているようなので、調整してみます (basemap_contourf3.py)。間隔を指定するオプションが pad で、デフォルト値は pad=0.15 です (横方向のカラーバーの場合)。ここでは pad=0.06 に変えました。ちょうど良い間隔になったと思います (図 6-5-9)。

```
cbar=plt.colorbar(shrink=0.8, orientation='horizontal', pad=0.06)
```

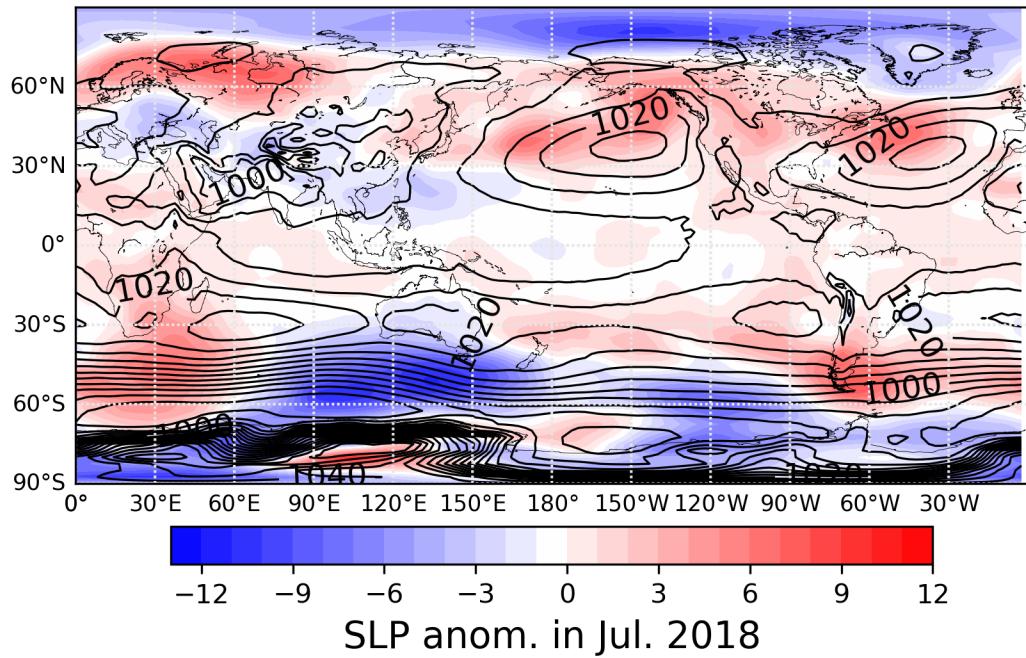


図 6-5-9 カラーバーとグラフの間を狭くする

なおカラーバーが縦方向の場合には、`pad=0.05`がデフォルト値です（図 6-5-10）。縦方向にするには `orientation='vertical'`を使いますが、デフォルトで縦方向なので、オプション無しでも同じです。`basemap_contourf4.py` を用いています。縦方向の場合、カラーバーとグラフの間はデフォルトで問題ないですが、カラーバーが長すぎるようです。

```
cbar=plt.colorbar(shrink=0.8, orientation='vertical')
```

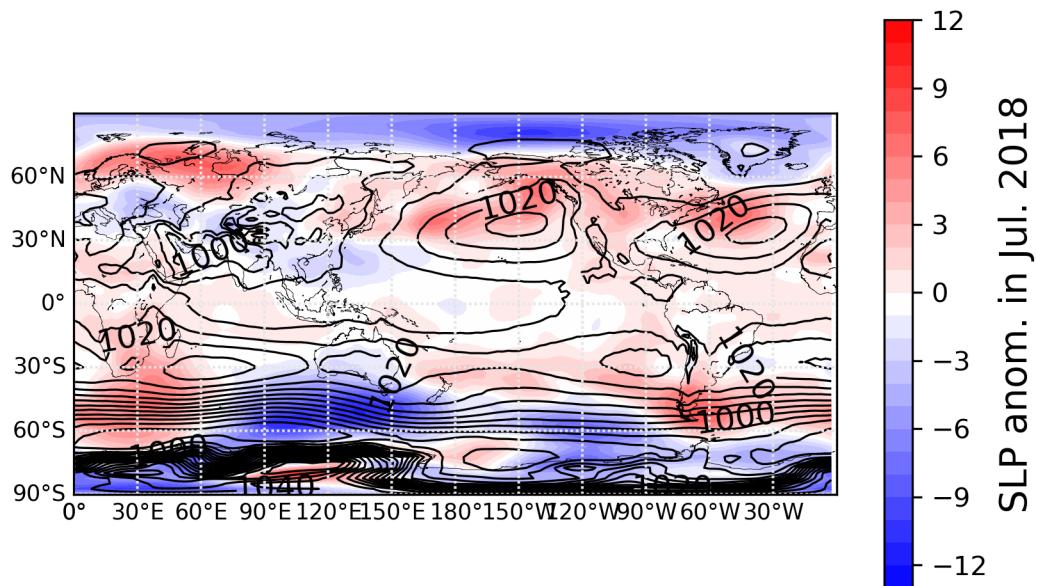


図 6-5-10 カラーバーを縦方向にした。カラーバーが長い

カラーバーの長さは shrink オプションで調節し、カラーバーのラベルの長さは fontsize で調節します (basemap_contourf5.py)。shrink オプションは、この図に合わせると 0.55 が最適でした (図 6-5-11)。カラーバーが小さくなるとカラーバーとグラフの間が広すぎるよう見えてしまうので、pad=0.03 に調整しました。ラベルをカラーバーの長さに収めるために、cbar.set_label では fontsize=12 と小さくしました。

```
cbar=plt.colorbar(shrink=0.55, orientation='vertical', pad=0.03)
cbar.set_label('SLP anom. in Jul. 2018', fontsize=12)
```

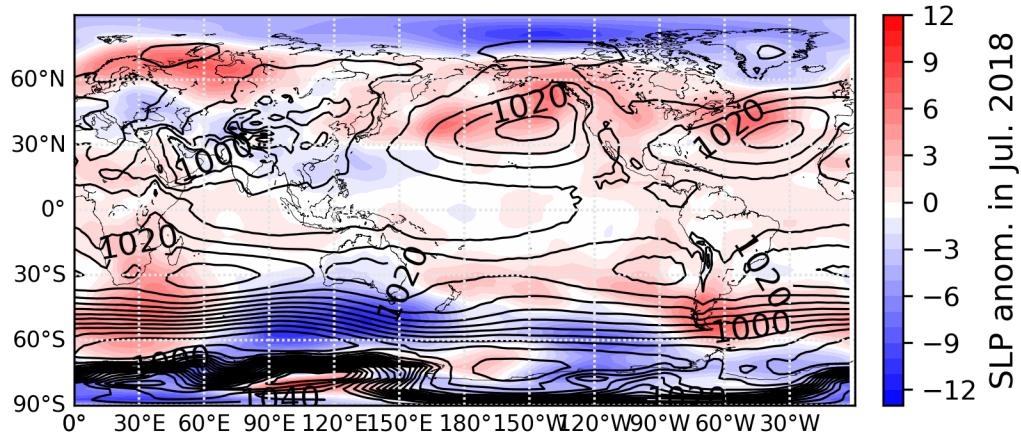


図 6-5-11 カラーバーとラベルの長さを調節した

カラーバーのラベルに表示する数字について、書式を設定することも可能です (basemap_contourf6.py)。等高線のラベルの場合には、`m.contour` の戻り値 `cs` のメソッド `cs.clabel` に `fmt` オプションを付けることで書式設定を行いました。カラーバーを描く `plt.colorbar` の場合にはオプションの名前が異なり、`format` オプションで行います。ここでは `format="%5.1f"` として、浮動小数点数を小数点以下第一位まで、全体の文字数としては最低 5 文字で表示します (書式の例については表 5-2-2 参照)。最低 5 文字に設定したため、4 文字以下の文字の左に空白が入り文字の右側の位置がほとんど揃いました (図 6-5-12)。

```
cbar = plt.colorbar(shrink=0.55, orientation='vertical', pad=0.03,
                    format="%5.1f")
```

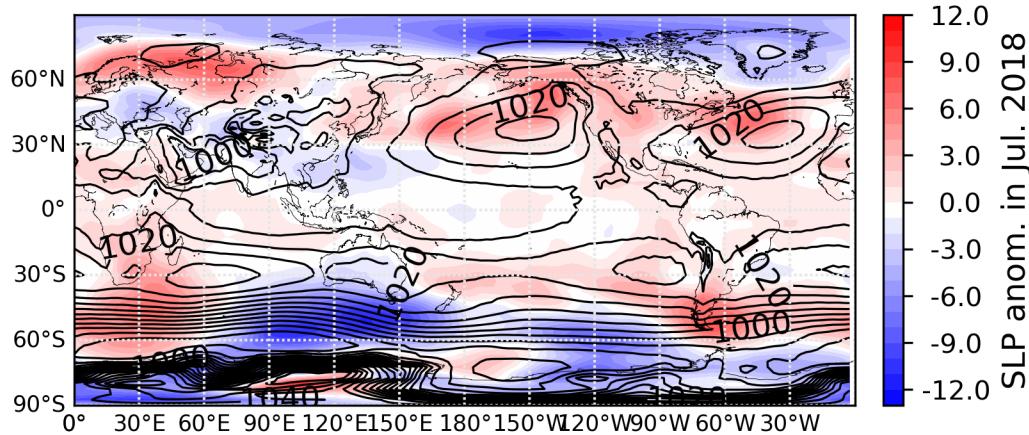
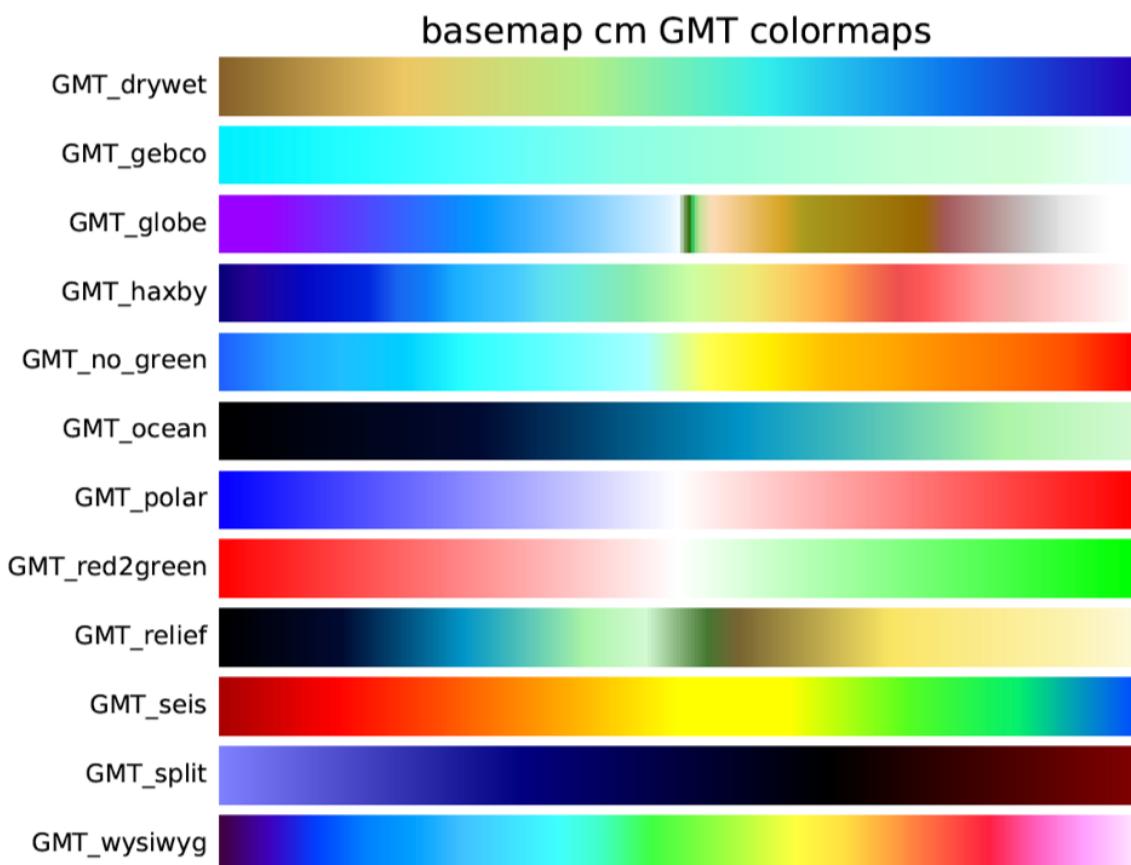


図 6-5-12 カラーバーのラベルの数字を小数点以下第一位まで表示

Basemap には、`plt.get_cmap` で設定可能な色テーブル（図 4-5-7 参照）の他に、Basemap 独自の色テーブルが含まれています（図 6-5-13）。Basemap と一緒に `cm` をインポートすることで利用できるようになります。色テーブルは、`cm.色テーブルの名前` の戻り値です。ここでは水色～黄色～赤色のように変化する `GMT_no_green` を利用してみます（図 6-5-14）。作図に用いたプログラムは `basemap_contourf7.py` です。

```
from mpl_toolkits.basemap import Basemap, cm
...
cmap=cm.GMT_no_green
```



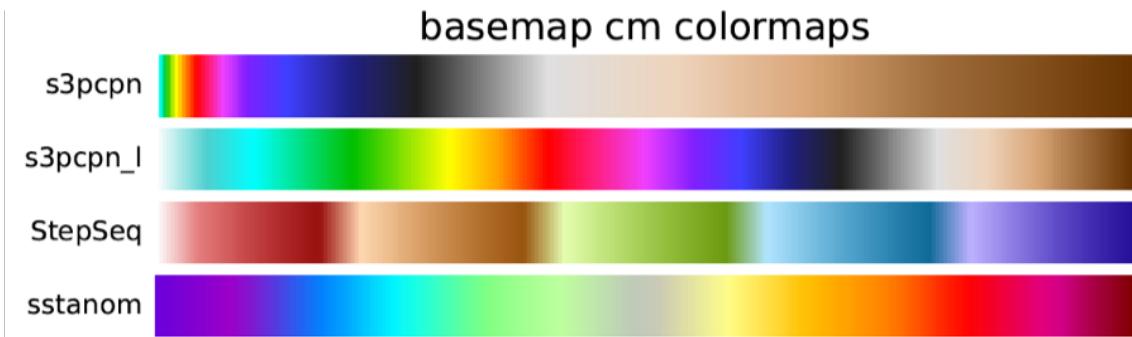


図 6-5-13 Basemap で利用可能な色テーブルの一覧

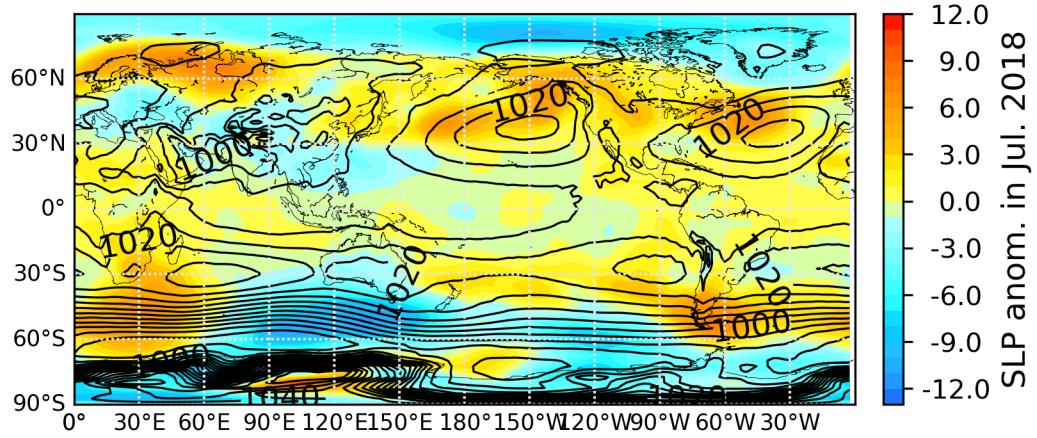


図 6-5-14 Basemap の色テーブル GMT_no_green を利用した

なお、Basemap の色テーブルのうち cm.GMT_polar は、これまで使っていた plt.get_cmap("bwr") の色テーブルと同じ色が設定されます。

6.5.4 巨大なバイナリファイルを読み込む場合

6.5.2 節で紹介したバイナリファイルの読み込み方法では、一旦、全てのデータをメモリ上に読み込む必要があり、大量のメモリを消費してしまいます。ここでは、メモリ消費を抑えるためバイナリファイルの必要な部分だけを読み出す方法を紹介します。例として図 6-5-4 の作成の所で行なった、1948 年 7 月の SLP 月平均値を作図してみます(図 6-5-15)。作図には basemap_array.py を使います。

NumPy の機能を使わないため読み込みが複雑になりますが、open を使ってファイルを開き、データの入力には array モジュールを使います。そのために最初に **import array** を行なっています。このモジュールでは読み込んだデータをリストのような配列に変換することができます。まず `with open("ファイル名", 'rb') as fin:` を使い、バイナリモード (rb) でファイルを開きます。ファイルを開けた際に fin というオブジェクトが生成され、fin が持っている `fin.seek` というデータを読み飛ばすメソッドを利用できるようになります。`fin.seek(読み飛ばすサイズ, 0)` のように使い、読み飛ばすサイズは byte で指定します。後ろの 0 は、ファイルの最初からの絶対的な位置を意味します(他に 1 は現在の位置から、2 はファイルの最後からの相対的な位置を表します)。このように不要なデータを読み飛ばすことで、メモリに保持されるデータを節約できます。

```
import array
idim = 144 # 経度方向のデータ数
jdim = 73 # 緯度方向のデータ数
datasize = idim * jdim
tstr = 7 # 開始点：1948 年 7 月
tend = 7 # 終了点：1948 年 7 月
num_rec = tend - tstr + 1
with open("output.bin", 'rb') as fin:
    # tstr の前まで読み飛ばす
    fin.seek(4 * datasize * (tstr - 1), 0)
    # データの読み込み
    buf = array.array('f')
    buf.fromfile(fin, datasize*num_rec)
```

データの読み込み部分では、`buf = array.array('f')`で読み込むデータが4バイト（単精度）浮動小数点数（'f'）であることを指定します。8バイト（倍精度）浮動小数点数の場合には'd'になります。戻り値のbufはメソッドを持っていて、`buf.fromfile(fin, データサイズ)`でデータの読み込みを行うことができます。データサイズは、byteではなくデータの個数なので注意が必要です。1つの時刻あたり `datasize = idim * jdim` 個のデータがあり、`num_rec` 時刻（ここでは1ヶ月データなので1時刻）のデータを読み込むので、`datasize * num_rec` です。

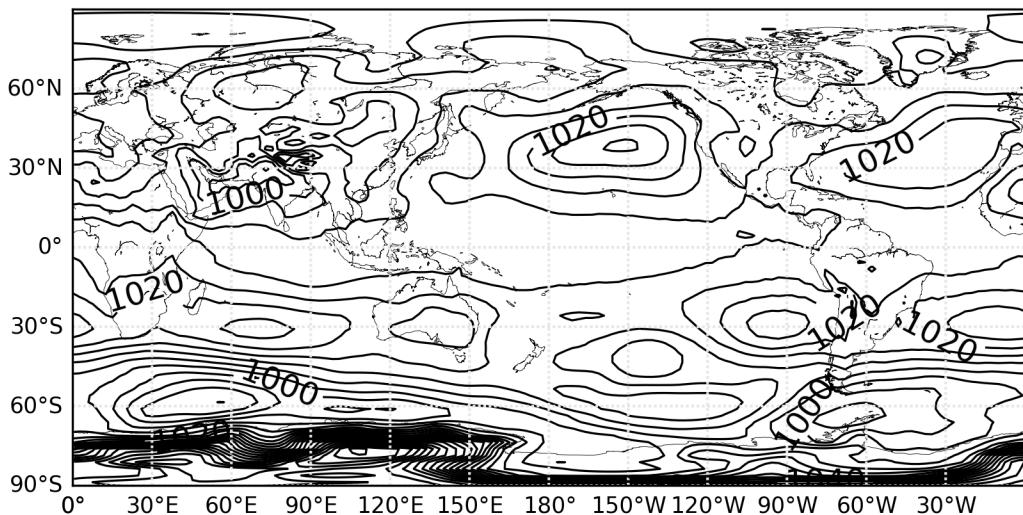


図6-5-15 図6-5-4と同じだが、データ読み込み方法を変えた

ここではNetCDFデータから変換したリトルエンディアンのデータを読み込んだので、そのまま利用することができました。ビックエンディアンのデータ場合には、次のようなエンディアン変換が必要となります。`sys.byteorder` は `little` か `big` を返すようになっており、`big` の場合にのみエンディアン変換を行います。`buf.byteswap()` の部分がエンディアン変換の操作になります。

```
import sys
...
if sys.byteorder == 'big':
    buf.byteswap()
```

6.5.5 矢羽、矢印の作図

5.1 節には矢羽や矢印を作成する方法を載せましたが、Basemap を使うと地図上に矢羽や矢印を描くことが可能になります。basemap_barbs.py は、日本付近の地図を描き、東西風、南北風データから矢羽をプロットするプログラムです。読み込むデータを準備するために、basemap_nc2bin2.py で月平均データの NetCDF ファイルを読み込み、バイナリ形式に変換しておきます。変換されたデータが output_uwnd.bin と output_vwnd.bin です。作図の前に、それぞれのファイルを読み込み u , v の配列に入力しておきます。ここでは最初の時刻データだけを使っているので、1948 年 1 月のデータが入ります。

```
din = np.fromfile("output_uwnd.bin", dtype='<f4', count=datasize) # U 入力
u = din.reshape(jdim, idim)
din = np.fromfile("output_vwnd.bin", dtype='<f4', count=datasize) # V 入力
v = din.reshape(jdim, idim)
```

日本付近の地図を描くため、basemap_jp.py で行なったようにランベルト正角円錐図法で領域を限定した作図を行います。

```
m = Basemap(projection='lcc', lat_0=35, lon_0=135, width=8000000, ¥
             height=6000000) # ランベルト正角円錐図法
m.drawcoastlines(linewidth=0.2, color='k') # 海岸線を描く
```

矢羽を描くには、**m.barbs** を用います。1 番目の引数は経度データ (x)、2 番目が緯度データ (y)、3 番目が東西風データ (u)、4 番目が南北風データ (v) です。1948 年 1 月の SLP 等値線に矢羽で東西風、南北風データを重ねた図が描かれます (図 6-5-16)。矢羽を間引くために、 $x[::3, ::3]$ のような工夫をしています。 $[::3, ::3]$ は作図に用いるデータを経度方向、緯度方向ともに 3 飛ばしに与えるスライス記法で、経度方向の 3 グリッド毎、緯度方向の 3 グリッド毎に矢羽が描かれるようになります。全部のグリッドに描いた場合にどのようになるか、**m.barbs(x , y , u , v)** で試してみましょう。

```
m.barbs( $x[::3, ::3]$ ,  $y[::3, ::3]$ ,  $u[::3, ::3]$ ,  $v[::3, ::3]$ ) # 矢羽を描く
```

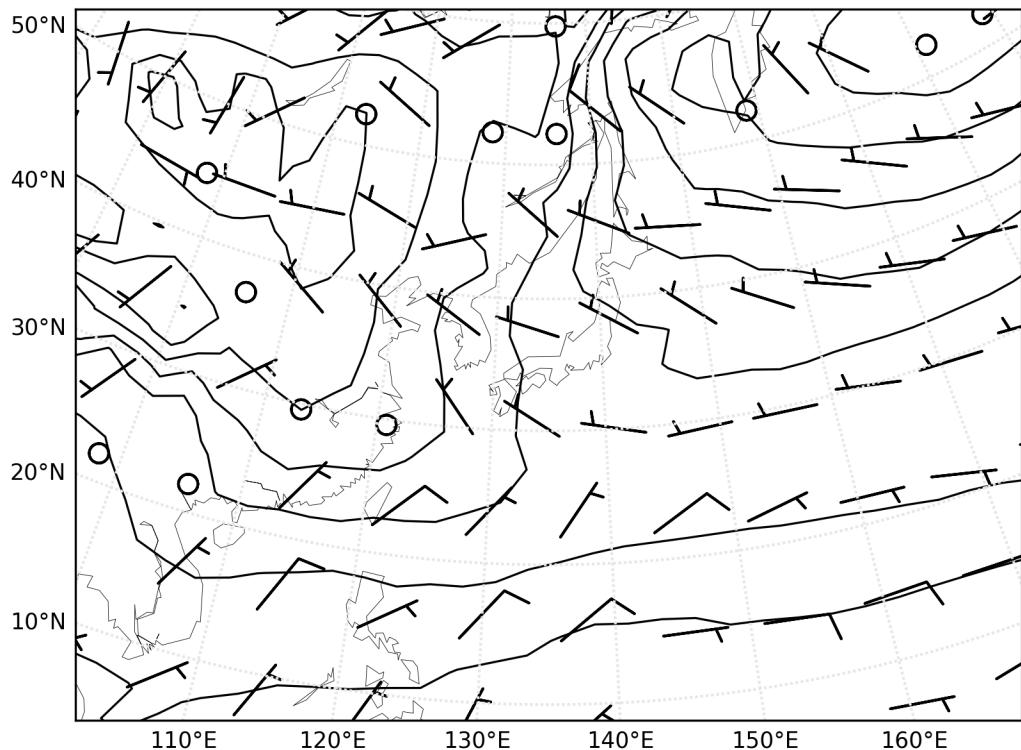


図 6-5-16 1948 年 1 月の SLP に矢羽で東西風、南北風データを重ねた

矢羽は元データの東西風、南北風の単位が m/s なので、短矢羽が 5 m/s、長矢羽が 10 m/s、旗矢羽が 50 m/s で描かれます。5.1.1 節で行なったようにノット表記にすることも可能です（図 6-5-17）。作図には basemap_barbs2.py を用いました。

m.barbs でも barb_increments オプションを与えることができ、5.1.1 節同様に矢羽を描く基準値を与えています。

```
m.barbs(x[::3, ::3], y[::3, ::3], u[::3, ::3], v[::3, ::3], ¥
        barb_increments=dict(half=2.57222, full=5.14444, flag=25.7222))
```

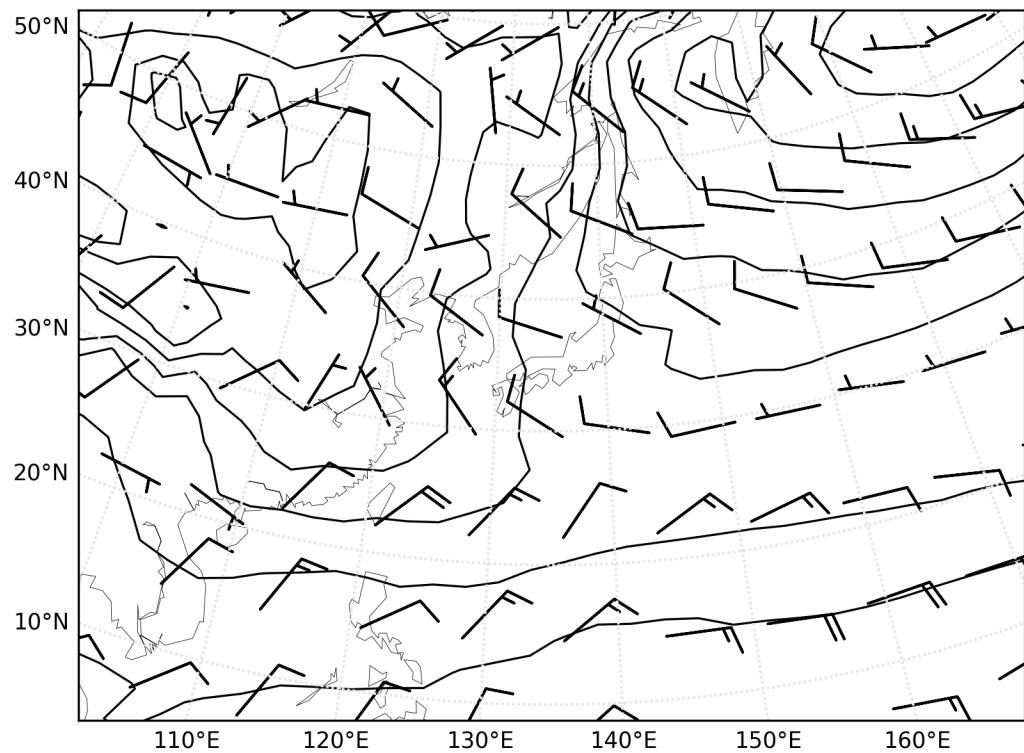


図 6-5-17 ノット表記にした場合

図 6-5-16 では、風速の小さい場所の矢羽が丸印になっています。丸印のサイズは 5.1.3 節と同様に、sizes オプションで変更可能です（図 5-1-3）。図 6-5-18 のように、丸印が表示されないようにするには、emptybarb=0.0 を与えます（basemap_barbs3.py）。

```
m.barbs(x[::3, ::3], y[::3, ::3], u[::3, ::3], v[::3, ::3], ¥
         sizes=dict(emptybarb=0.0))
```

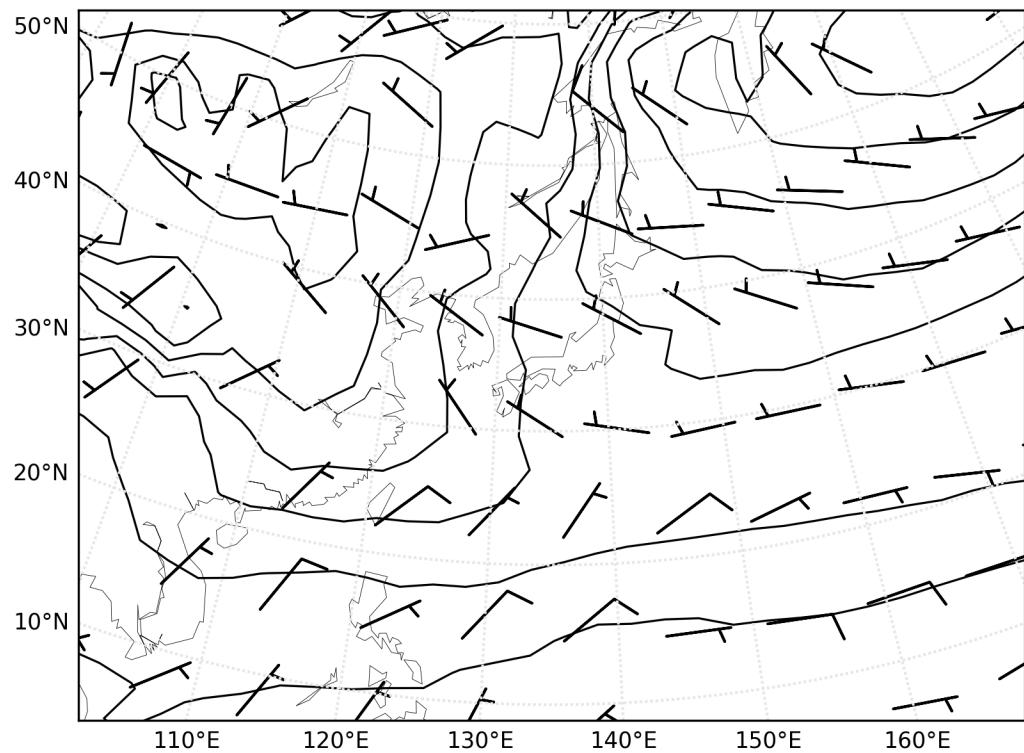


図 6-5-18 値の小さい矢羽を丸印で表示しない

5.1.3 節と同様に、矢羽のサイズは `length` オプションで、色は `color` オプションで変えることができます(デフォルト値: 7)。ここでは、`length=6`、`color='r'` (赤色) に変更します (図 6-5-19)。作図には `basemap_barbs 4.py` を用いました。

```
m.barbs(x[::3, ::3], y[::3, ::3], u[::3, ::3], v[::3, ::3], ¥
        sizes=dict(emptybarb=0.0), length=6, color='r')
```

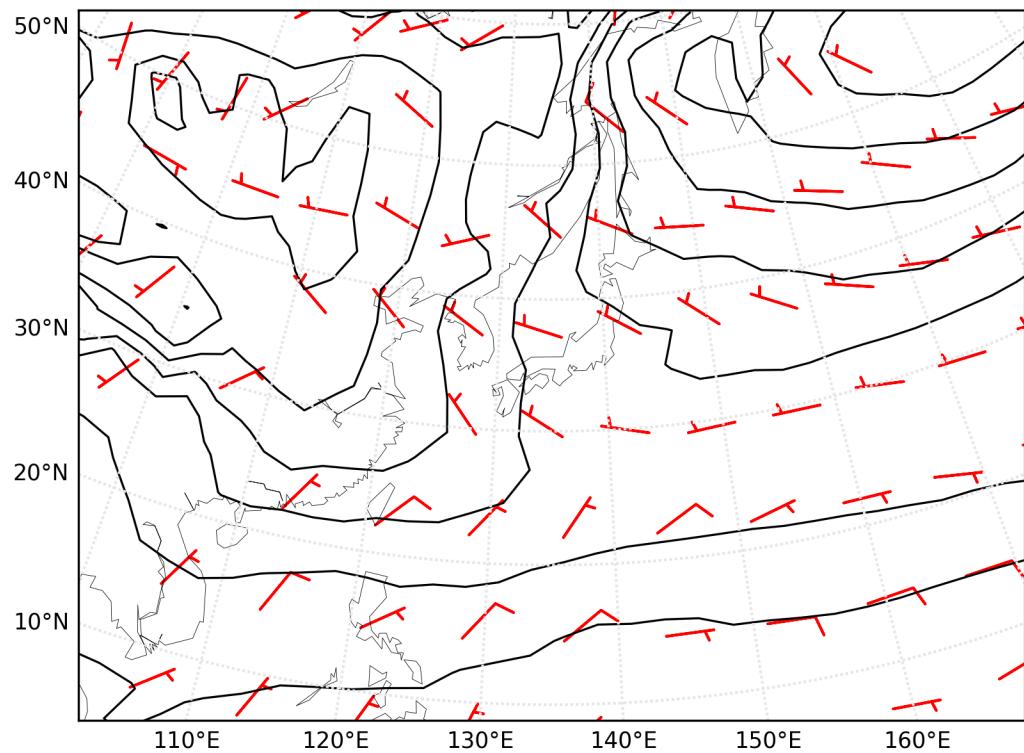


図 6-5-19 矢羽のサイズを小さくする

矢印を描くには、`m.quiver` を用います。図 6-5-16 の矢羽の代わりに矢印を描いてみます（図 6-5-20）。作図には `basemap_quiver.py` を用いました。引数は矢羽の時と同様に 1 番目が経度 (x)、2 番目が緯度 (y)、3 番目が東西風 (u)、4 番目が東西風 (v) です。

```
m.quiver(x[:3, :3], y[:3, :3], u[:3, :3], v[:3, :3]) # 矢印を描く
```

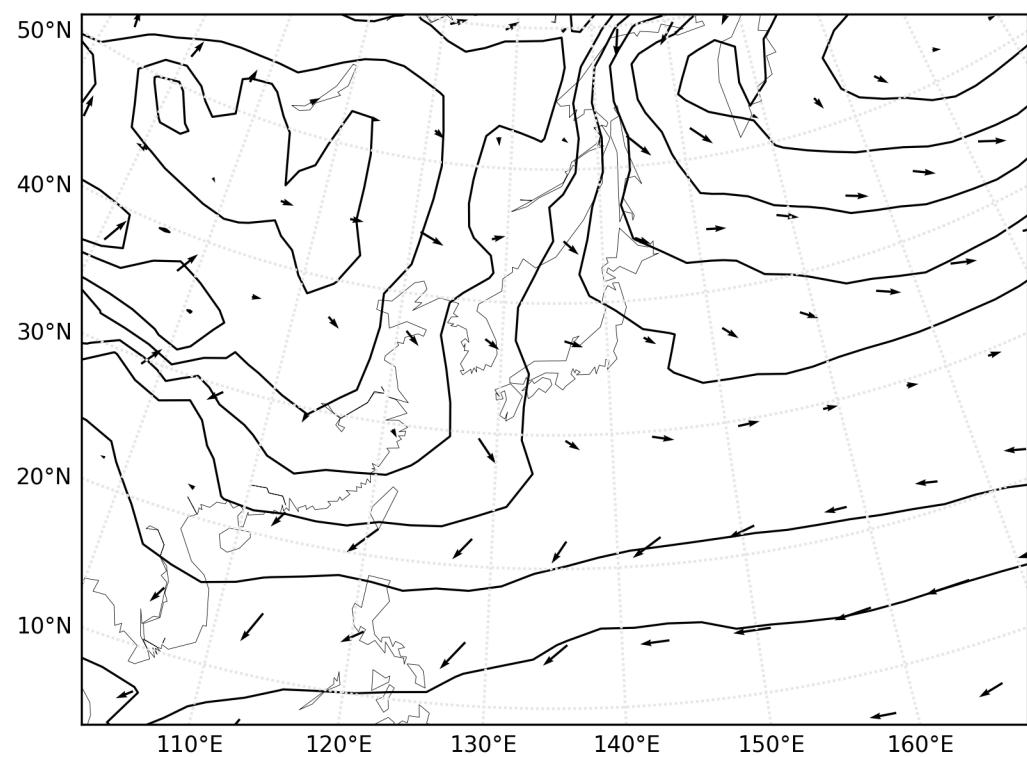


図6-5-20 図6-5-16の矢羽を矢印で描く

6.6 図の体裁

ここでは、様々な図法で体裁を整える方法を紹介します。

6.6.1 極座標表示で等高線、陰影を描く

図 6-5-12 を北極中心の極座標表示で描いてみます（図 6-6-1）。作図には、`basemap_polar.py` を用いました。`lon_0=180` なので、北太平洋が手前側に来るような極座標表示になりました。

```
m = Basemap(projection='npstere', lon_0=180, boundinglat=20, resolution='l')
```

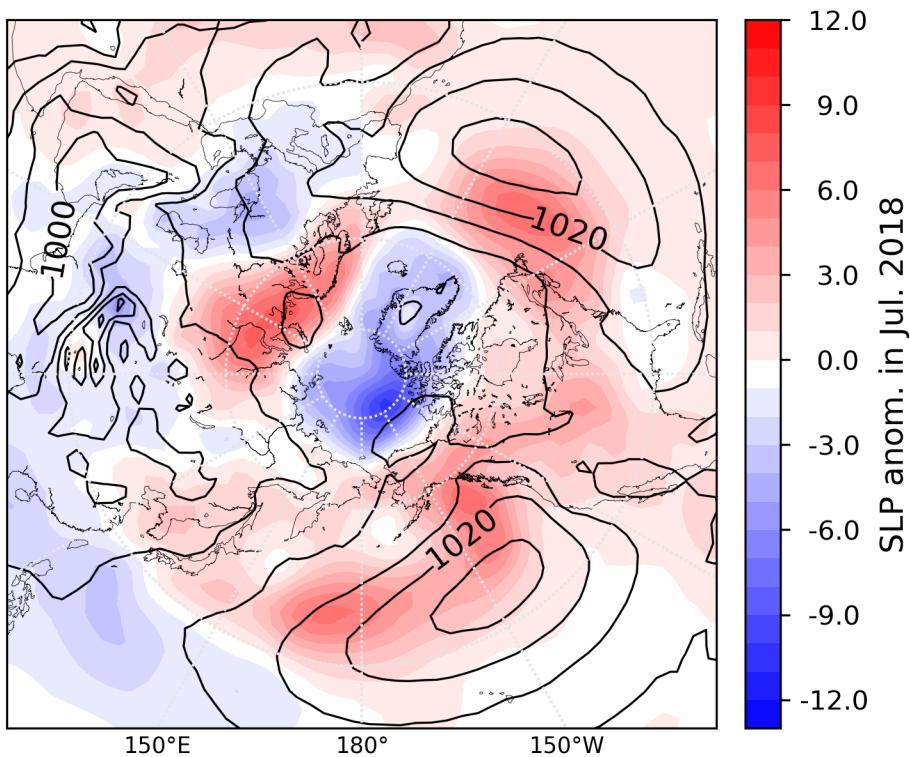


図 6-6-1 図 6-5-12 を北極中心の極座標表示で描く

図 6-6-1 をよく見ると、東経 0 度の場所で等高線や陰影が滑らかにつながらないことが分かります。東西方向には 0 ~ 360 度までのデータが入るようになっていますが、読み込んだデータの場合には、357.5 度が最後で 360 度のデータが入っていないためです。

図6-6-2は、東経0度の場所を滑らかにつなげる様にしました。例えばイギリス付近で等高線が切れていた場所がつながるようになりました。また、スペイン付近の負偏差が東経0度で切られていたものが、西経側にも現れる様になっています。作図には、`basemap_polar2.py`を用いました。

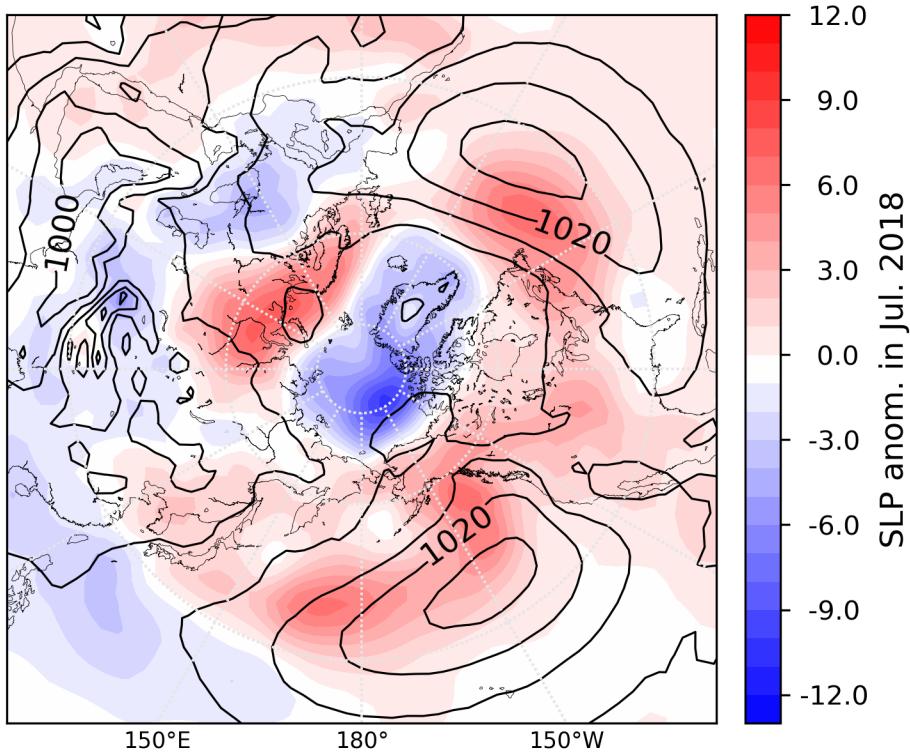


図6-6-2 図6-6-1の東経0度を滑らかにつなげる

東経0度の場所で滑らかにつなげるため、東経360度のデータを追加するようにしています。東経360度は東経0度と同じなので、東経0度のデータをコピーします。まずNumpyの`np.zeros`を使い、`slp=np.zeros((num_rec, jdim, idim+1))`で、`slp`データを入力するための配列を作成します。作成時に値は全て0で埋められます。Numpyには他にも、配列を作成して値を全て1で埋める`np.ones`、要素を初期化しないで配列を作成する`np.empty`があります。次に0~357.5度までのデータを`slp[:, :, 0:idim] = din.reshape(num_rec, jdim, idim)`で`slp`にコピーします。データの形状を使い、ファイルから入力したデータを経度、緯度、時間の次元を持ったデータに変え、`slp`にコピーしています。ここで、`0:idim`のスライス記法

を使っていて、0 番目から(idim-1)番目までにコピーされます。まだ idim 番目には初期値の 0 が入っているままなので、slp[:, :, idim] = slp[:, :, 0]で東経 0 度のデータをコピーします。最後に idim の値を 1 増やし、後の処理で参照する経度方向のデータ数が正しくなるようにします。

```
idim = 144 # 経度方向のデータ数
jdim = 73 # 緯度方向のデータ数
num_rec = 860 # 時間方向のデータ数
datasize = idim * jdim
din = np.fromfile("output.bin", dtype='<f4', count=datasize*num_rec) # 入力
slp = np.zeros((num_rec, jdim, idim + 1)) # 値が 0 の配列を作成
slp[:, :, 0:idim] = din.reshape(num_rec, jdim, idim) # 0~357.5 度までコピー
slp[:, :, idim] = slp[:, :, 0] # 0 度のデータを 360 度のデータにコピー
idim = idim + 1 # 後の処理で使う経度方向のデータ数を 1 増やす
```

6.6.2 極座標表示で矢羽を描く

北極中心の極座標表示で矢羽を描く場合を考えてみます。まずは、矢羽を3飛ばしに描きます（図6-6-3）。東西風、南北風データは、6.6.1節同様に読み込みます。読み込んだデータから2018年7月を取り出し、矢羽を描きます。先ほど同様に `emptybarb=0.0` を設定し、値が小さい場合の丸印が表示されないようにします。作図には、`basemap_polar3.py` を用いました。

```
# 2018年7月の東西風、南北風
n = (2018 - 1948) * 12 + 6
# 矢羽を描く
m.barbs(x[::-3], y[::-3], u[n, ::3, ::3], v[n, ::3, ::3], \
         color='k', sizes=dict(emptybarb=0.0), length=6)
```

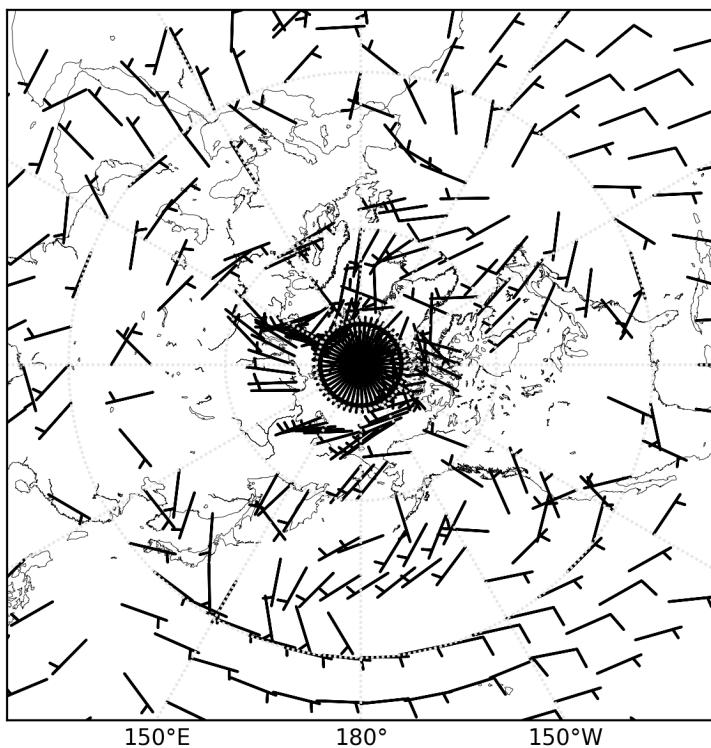


図6-6-3 2018年7月の東西・南北風を矢羽で描く

図6-6-3では、北極付近の矢羽が混んでいるように見えます。そこで、作図前に北極から緯度方向20番目までのグリッドは経度方向2つ飛ばしに0

を、北極から 10 番目までのグリッドには 5 つ飛ばしに 0 を入力しておきます。こうしておくことで、作図時には該当するグリッドに矢羽が描かれないようになります。作図時に 3 飛ばしに描いているので、北極から 11 番目から 20 番目は 6 飛ばし（これまでの矢羽の数が半分に）になります。さらに北極から 10 番目までは、15 毎に矢羽を描かないグリッドが現れます。

```
u[:, 0:20, ::2] = 0.0 # 北極から 20 番目までは 2 つ飛ばしに 0
u[:, 0:10, ::5] = 0.0 # 北極から 10 番目までは 5 つ飛ばしに 0
...
m.barbs(x[::-3, ::3], y[::-3, ::3], u[n, ::3, ::3], v[n, ::3, ::3],
         color='k', sizes=dict(emptybarb=0.0), length=6)
```

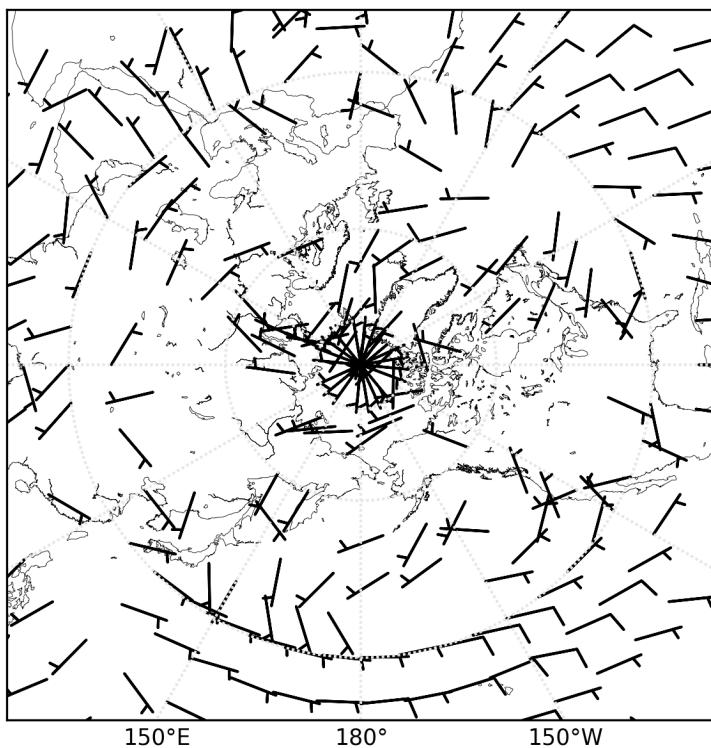


図 6-6-4 北極に近いグリッドで矢羽が混まないようにする

6.6.3 正射投影図法で衛星画像風に描く

6.2.2 節で紹介した正射投影図法を使い、衛星画像風に雲を描いてみます。

2020年12月末には、この時期としては数年に一度クラスの寒波が到来し、日本海側を中心に大雪となりました。ここでは、京大生存圏データベースで公開されている GRIB2 形式の気象庁 GSM データを使い、衛星観測の水蒸気画像を模して簡易的に相対湿度が高いところを白で低いところを黒で表示します。簡易的に作図することを目的としていますので、衛星観測と比較する目的に用いる場合には、衛星データ・シミュレータを利用してください。

まずは 2020 年 12 月 31 日 18UTC の GSM 全球データを取得し、GRIB2 形式のデータを NetCDF 形式へ変換します。サンプルプログラムには、データの取得と NetCDF 形式への変換も含まれていますが、これから行う手順を理解するため、コマンドラインで操作しておきます。GSM 全球データは、

http://database.rish.kyoto-u.ac.jp/arch/jmadata/data/gpv/original/2020/12/31/Z__C_RJTD_20201231180000_GSM_GPV_Rgl_FD0000_grib2.bin から取得します。GRIB2 形式から NetCDF 形式への変換には、6.4.5 節で紹介した wgrib2 を使います。

```
% wgrib2 Z__C_RJTD_20201231180000_GSM_GPV_Rgl_FD0000_grib2.bin  
-netcdf 20201231180000_GSM_FD0000.nc
```

データの詳細は、ncdump で取得します。

```
% ncdump -h 20201231180000_GSM_FD0000.nc
```

出力：

```
netcdf ¥20201231180000_GSM_FD0000 {  
dimensions:  
    latitude = 361 ;  
    longitude = 720 ;  
    time = UNLIMITED ; // (1 currently)  
variables:  
    double latitude(latitude) ;  
        latitude:units = "degrees_north" ;  
        latitude:long_name = "latitude" ;  
    double longitude(longitude) ;  
        longitude:units = "degrees_east" ;  
        longitude:long_name = "longitude" ;  
    double time(time) ;  
        time:units = "seconds since 1970-01-01 00:00:00.0 0:00" ;  
    ...  
    float RH_1000mb(time, latitude, longitude) ;  
        RH_1000mb:_FillValue = 9.999e+20f ;  
        RH_1000mb:short_name = "RH_1000mb" ;  
        RH_1000mb:long_name = "Relative Humidity" ;  
        RH_1000mb:level = "1000 mb" ;  
        RH_1000mb:units = "percent" ;  
    ...
```

出力を見ると緯度の変数名は latitude、経度の変数名は longitude、相対湿度の変数名は「RH_気圧 mb」と分かれます。気圧面は、1000 hPa、925 hPa、850 hPa、700 hPa、600 hPa、500 hPa、400 hPa、300 hPa のデータが提供されています。なお変数名の mb は古い気圧の単位ですが、現在使われている hPa にそのまま置き換えられます。衛星観測の水蒸気画像は、対流圏中上層の水蒸気量を表すため、700～300 hPa のデータを使って作図します（図 6-6-5）。作図には basemap_ortho.py を使いました。

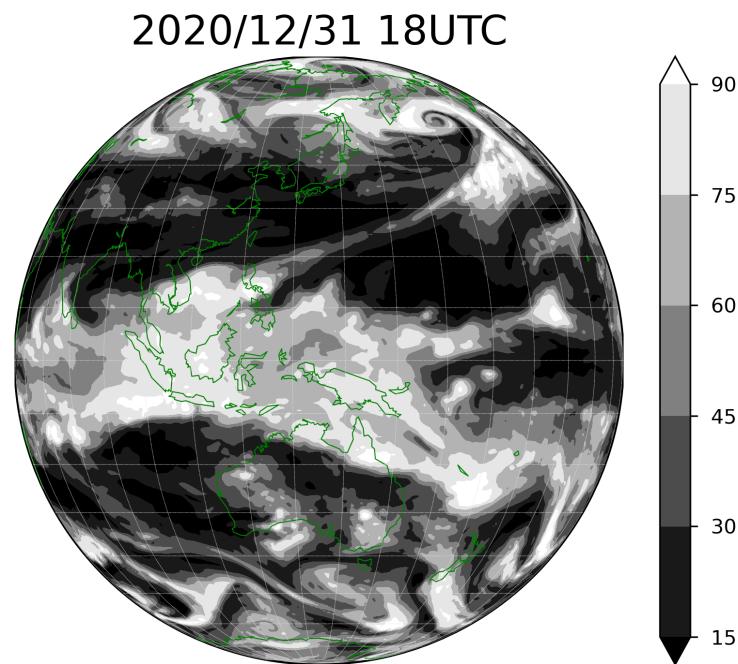


図 6-6-5 2020 年 12 月 31 日 18UTC (日本時間 2021 年 1 月 1 日 3 時) の 700~300 hPa 相対湿度を使い、衛星観測の水蒸気画像を模して作図した

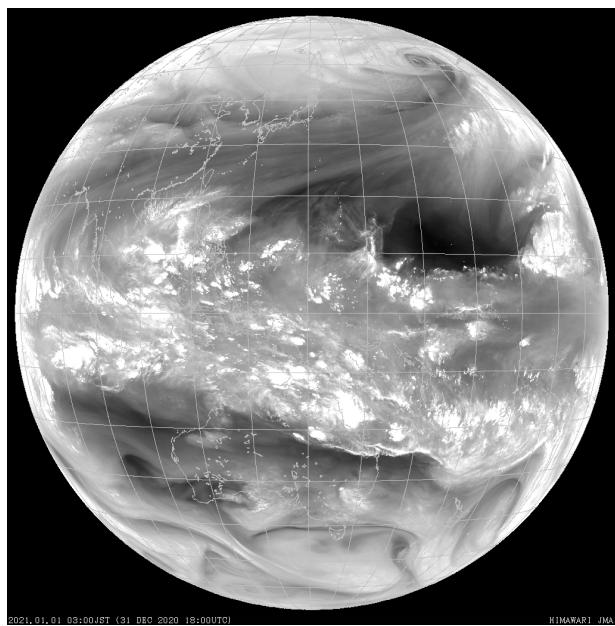


図 6-6-6 2020 年 12 月 31 日 18UTC (日本時間 2021 年 1 月 1 日 3 時) のひまわり水蒸気画像 (気象庁、<http://www.jma.go.jp/jp/gms/>)

実際の水蒸気画像（図6-6-6）と比較すると、乾いているところが黒くなりすぎていますが、日本に寒波をもたらしていたベーリング海付近の低気圧の渦巻き構造やフィリピン付近まで伸びた前線の構造がよく見えています。

最初に、`urllib.request.urlretrieve` を使い、GRIB2 形式の気象庁 GSM データをダウンロードします。次に、`wgrib2` を使い GRIB2 形式から NetCDF 形式への変換を行います。Python には外部のコマンドを実行することができる `subprocess` モジュールがあり、ここでは `subprocess.run` で `wgrib2` コマンドを実行し、コマンドの標準出力を画面に表示させています。コマンドのオプションはコマンドと一緒にリストにし、最初の引数として渡します。ここでは、`wgrib2 file_name -netcdf file_name_nc` を行いたいので、コマンドでスペースを開ける代わりにリストをカンマで区切り、`["wgrib2", file_name, "-netcdf", file_name_nc]` のように要素を追加します。

```
import subprocess
import urllib.request
# データ取得の有無
retrieve = True
# URL
url = "http://database.rish.kyoto-
u.ac.jp/arch/jmadata/data/gpv/original/2020/12/31/Z__C_RJTD_202012311800
00_GSM_GPV_Rgl_FD0000_grib2.bin"
file_name ="Z__C_RJTD_20201231180000_GSM_GPV_Rgl_FD0000_grib2.bin"
file_name_nc = "20201231180000_GSM_FD0000.nc"
#
# ファイルのダウンロードと変換
if retrieve:
    urllib.request.urlretrieve(url, file_name)
    res = subprocess.run(["wgrib2", file_name, "-netcdf", file_name_nc],
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE)
    print(res.stdout.decode("utf-8")) # 標準出力を画面に表示
```

NetCDF ファイルができたので、読み込んでデータを取り出します。緯度・経度の変数名が latitude・longitude、相対湿度の変数名が RH_700mb、、、RH_300mb だったので、それぞれの変数名でデータを取り出し、lon、lat（1 次元）、rh700、rh600、rh500、rh400、rh300（2 次元）に格納します。最後に対流圏中上層の平均を計算して rh に格納します。ここでは、簡単化のために高度による寄与の違いは考えていません。

```
nc = netCDF4.Dataset(file_name_nc, 'r') # NetCDF データの読み込み
# データサイズの取得
idim = len(nc.dimensions['longitude']) # 経度方向のデータ数
jdim = len(nc.dimensions['latitude']) # 緯度方向のデータ数
# 変数の読み込み
lon = nc.variables["longitude"][:] # 経度
lat = nc.variables["latitude"][:] # 緯度
rh700 = nc.variables["RH_700mb"][:].reshape(jdim, idim) # 700hPa 相対湿度
rh600 = nc.variables["RH_600mb"][:].reshape(jdim, idim) # 600hPa 相対湿度
rh500 = nc.variables["RH_500mb"][:].reshape(jdim, idim) # 500hPa 相対湿度
rh400 = nc.variables["RH_400mb"][:].reshape(jdim, idim) # 400hPa 相対湿度
rh300 = nc.variables["RH_300mb"][:].reshape(jdim, idim) # 300hPa 相対湿度
rh = (rh700 + rh600 + rh500 + rh400 + rh300) / 5. # 中上層の平均
nc.close() # ファイルを閉じる
```

正射投影図法による作図部分です。プロット領域を作成してタイトルを付けた後、正射投影図法（projection='ortho'）を指定して Basemap を呼び出します。衛星画像では東経 135 度の赤道上が視点の中心となっているため、lon_0=135、lat_0=0 として同じ視点で作図を行います。

```
fig, ax = plt.subplots(figsize=(6, 6)) # プロット領域の作成
ax.set_title(title, fontsize=20) # タイトルをつける
# 正射投影図法の準備
m = Basemap(projection='ortho', lon_0=135, lat_0=0)
```

海岸線を描き、経度線・緯度線を引きます。衛星画像では海岸線を白で表示していますが、判別しやすいように緑色 (color='g') とし、太さは 0.5 (linewidth=0.5) としました。経度線・緯度線は、太さを 0.3 としました。

```
m.drawcoastlines(linewidth=0.5, color='g') # 海岸線を描く
# 経度線を引く
m.drawmeridians(np.arange(0, 360, 10), color="0.9",
                 fontsize='small', linewidth=0.3)
# 緯度線を引く
m.drawparallels(np.arange(-90, 90, 10), color="0.9",
                 fontsize='small', linewidth=0.3)
```

陰影を作図する準備として、1次元の経度、緯度データ (lon, lat) を正射投影図法上の経度、緯度 (x, y) に変換したものを用意します。

```
lons, lats = np.meshgrid(lon, lat) # 経度・緯度座標の準備
# 図法の経度、緯度に変換する (NetCDF データの緯度・経度の単位は度)
x, y = m(lons, lats)
```

陰影の作図に用いる色テーブルとしては、これまで図 4-5-7 や図 6-5-13 のような既存のものを使ってきました。ここでは、matplotlib.colors モジュールを使用して新しい色テーブルを作成する方法を紹介します。色テーブルの作成には、色の名前をリストにして渡すことで作成する ListedColormap([色の名前のリスト](#))、RGB 情報から作成する LinearSegmentedColormap 等を使います。色テーブルを滑らかに変化させるため、LinearSegmentedColormap を使います。RGB 情報を segment_data に格納して、

```
cmap = LinearSegmentedColormap('colormap_name', segment_data)
```

のように色テーブル cmap を生成します。segment_data では、0 ~ 1 の範囲で red、green、blue の値を指定していきます。ここでは黒から白に変化させるため、0.0 で黒、0.5 で灰色、1.0 で白となるような値に設定しました。

```

# 色テーブルの設定
segment_data = {
    'red': [
        (0.0,    0 / 255,    0 / 255),
        (0.5, 128 / 255, 128 / 255),
        (1.0, 255 / 255,    0 / 255),
    ],
    'green': [
        (0.0,    0 / 255,    0 / 255),
        (0.5, 128 / 255, 128 / 255),
        (1.0, 255 / 255,    0 / 255),
    ],
    'blue': [
        (0.0,    0 / 255,    0 / 255),
        (0.5, 128 / 255, 128 / 255),
        (1.0, 255 / 255,    0 / 255),
    ],
}
cmap = LinearSegmentedColormap('colormap_name', segment_data)

```

作成した色テーブルと 2 次元の経度 x、緯度 y、相対湿度 rh データを使い、陰影を描きます。これまで紹介した plt.colorbar でも描くことが可能ですが、ここでは陰影を描いた戻り値 cs を使い **m.colorbar** で描きました。m.colorbar では、水平・鉛直を指定する orientation の代わりに、カラーバーを付ける位置を location で指定します。位置としては、右側 ('right')、左側 ('left')、下側 ('bottom')、上側 ('top') の指定が可能です。

```

cs = m.contourf(x, y, rh, cmap=cmap, extend='both') # 陰影を描く
# カラーバーを付ける
cbar = m.colorbar(cs, location='right', pad=0.25, format="%3.0f")

```

6.7 まとめ

最後に Basemap で利用可能な作図のメソッドを一覧にまとめます。

表 6-7-1 Basemap で利用可能な作図のメソッド (m は Basemap() の戻り値)

m.メソッド	説明
m.drawcoastlines	海岸線を描く
m.drawparallels	緯度線を引く
m.drawmeridians	経度線を引く
m.drawcountries	国境線を描く
m.drawstates	州の境界線を描く (南北アメリカ、オーストラリア)
m.drawrivers	河川を描く
m.plot	マーカーをプロット
m.text	文字列をプロット
m.contour	等値線
m.contourf	陰影
m.pcolormesh	擬似カラープロット
m.colorbar	カラーバーを付ける
m.fillcontinents	大陸と湖を塗り潰す
m.drawmapboundary	背景を塗り潰す
m.barbs	矢羽
m.quiver	矢印