

5. 様々な1次元・2次元プロット

5.1 矢羽や矢印を描く

matplotlibには、矢羽や矢印（ベクトル）を描く機能も含まれています。

5.1.1 矢羽を描く

矢羽を描くには `plt.barbs(引数1, 引数2, 引数3, 引数4)` を使います。1番目の引数がx軸上の位置、2番目の引数がy軸上の位置で、3番目の引数が東西風データ、4番目の引数が南北風データに当たります。`barbs_sample.py` は、サンプルの矢羽を描くためのプログラムです。図5-1-1は作成された矢羽で、下の文字が（東西風速、南北風速）を表します。風向は元データの東西風速(u)と南北風速(v)から計算され、矢羽の向きは天気図等で使われるものと同様に風向と逆向き（西風なら西向きの矢羽）になります。風速も東西風速と南北風速から $(u^2+v^2)^{1/2}$ のように計算されます。表示される矢羽の種類や数は計算された風速で決まっており、デフォルトでは5が短矢羽、10が長矢羽、50が旗矢羽です。単位は元データの単位で、m/sなら5 m/sが短矢羽、10 m/sが長矢羽、50 m/sが旗矢羽で表示されます。そのため、例えば左端の15 m/sの西風（東西風速が15 m/s）では、長矢羽1本、短矢羽1本で西向きの矢羽になっており、左端の50 m/sの東風（東西風速が-50 m/s）では、旗矢羽1本で東向きの矢羽になっています。同様に、中央の15 m/sの北風（南北風速が-15 m/s）では北向き、左から2番目の東西風速15 m/s、南北風速15 m/sでは南西向きの矢羽で長矢羽2本、右から2番目の東西風速-35 m/s、南北風速25 m/sでは南東向きの矢羽で長矢羽4本、短矢羽1本です。なお、元データの単位がノット(kt)なら、5 ktが短矢羽、10 ktが長矢羽、50 ktが旗矢羽になります。

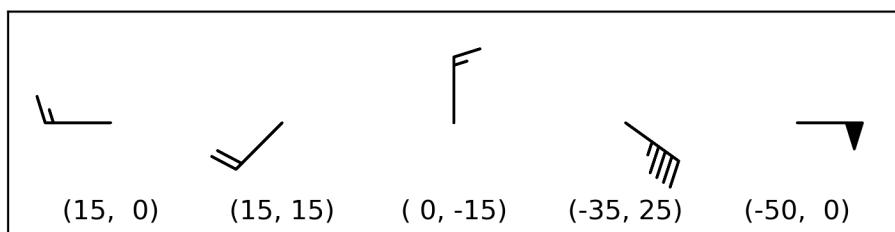


図5-1-1 基本的な矢羽、下側の数字は（東西風速、南北風速）

このプログラムでは、矢羽の作図に用いるデータをプログラム中に記載しており、x 軸上の位置 (x)、y 軸上の位置 (y)、東西風速 (u)、南北風速 (v) をそれぞれタプルとして作成しました。矢羽を作図する際には、これら 4 つのタプルを plt.barbs の引数として渡しています。plt.barbs では、引数として与えられた 4 つのタプルの同じ番号の要素を 1 つのデータとして処理しています。

```
x = (1.0, 1.25, 1.5, 1.75, 2.0)
y = (1.5, 1.5, 1.5, 1.5, 1.5)
u = (15.0, 15.0, 0.0, -35.0, -50.0)
v = (0.0, 15.0, -15.0, 25.0, 0.0)
plt.barbs(x, y, u, v) # 矢羽を描く
```

矢羽の下側に (u, v) の値を表示していますが、これは plt.text でプロットしています。まず、x、y、u、v のデータを zip に渡して、同じ番号の要素を x1、y1、u1、v1 として取り出します。取り出した x1、y1 からテキストの位置 (x1, y1-0.2) を、u、v から表示するテキスト name を作成します。

```
for x1, y1, u1, v1 in zip(x, y, u, v):
    name = "{s1:s}{f1:2.0f}{s2:s}{f2:2.0f}{s3:s}".format(s1="(", ¥
        f1=u1, s2=", ", f2=v1, s3=")")
    plt.text(x1, y1-0.4, name, ha='center', va='center')
```

5.1.2 元データが m/s のものをノットで表示

元データが m/s のものを天気図と合わせたスケールの矢羽で描きたい場合など、ノットで表示したいこともあるかと思います。短矢羽、長矢羽、旗矢羽を描く基準となる風速は、barb_increments オプションで個別に指定することができます。短矢羽が half、長矢羽が full、旗矢羽が flag です。デフォルトでは half=5、full=10、flag=50 なので、元データがノットであれば、これらの値を 5 kt、10 kt、50 kt に相当する 2.57222 m/s、5.14444 m/s、25.7222 m/s に書き換えます。barbs_sample2.py は、前節と同じ m/s で入力したサンプルを kt 表記にするプログラムです。barb_increments オプションには、次のように half、full、flag を key とした辞書で渡します。

```
plt.barbs(x, y, u, v,   
         barb_increments=dict(half=2.57222, full=5.14444, flag=25.7222))
```

図5-1-2のように、矢羽から読み取れる風速の値がm/s表記のおおよそ倍になっており、m/sで入力したデータをktで表示できたことが分かると思います。

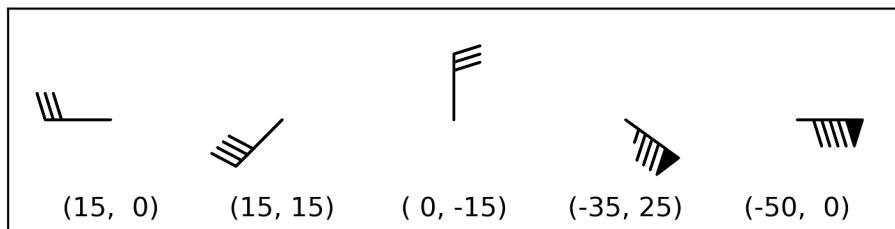


図5-1-2 ノット表示にしたもの

5.1.3 矢羽のスタイルを変更

ここまでではデフォルトのスタイルを使ってきましたが、図によっては矢羽のサイズや間隔を調整したいなど、矢羽のスタイルを変更したいこともあると思います。矢羽のサイズを指定するのはlengthオプションで、デフォルト値はlength=8です。この値を変更すると矢羽の縦横同時にサイズが変更されます。基本的には、lengthの変更だけで問題ないでしょう。

どうしても矢羽の各部分の見た目を変えたい場合には変更可能なので、方法を紹介しておきます。各部分の見た目はsizesオプションで調整できます。sizesオプションには、emptybarb、spacing、height、widthをkeyとし、その値を浮動小数点数とした辞書を渡します。それぞれのkeyは、矢羽を描けないほど風速が小さかった場合に描かれるシンボル(円)の大きさ(emptybarb)、矢羽同士の間隔(spacing)、矢羽の横方向の長さ(height)、旗矢羽の幅(width)を表します(図5-1-3)。作図にはデフォルト値を用いていて、それぞれemptybarb=0.15、spacing=0.12、height=0.4、width=0.25です。

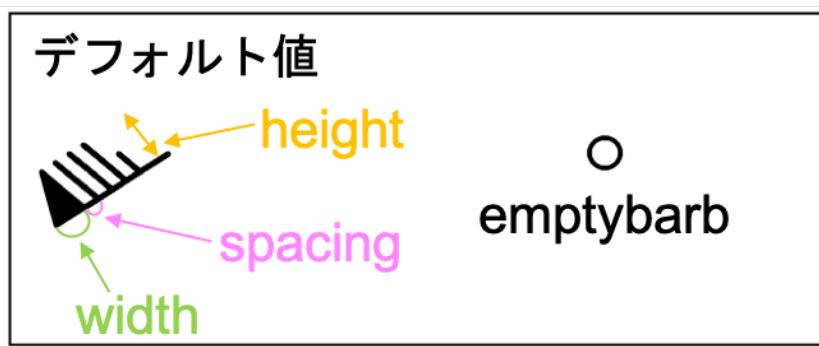


図 5 – 1 – 3 矢羽のデフォルトスタイル
(emptybarb=0.15、spacing=0.12、height=0.4、width=0.25)

図 5 – 1 – 4 の左側は emptybarb をデフォルト値に対して小さく、spacing を 2 倍間隔に、height を 2 倍にしたものです。次のように sizes オプションを設定しました。

```
sizes=dict(emptybarb=0.05, spacing=0.24, height=0.8, width=0.25)
```

右側の図は左側の図に対して旗矢羽の幅を 2 倍にしたもので、sizes オプションは次のようにしています。

```
sizes=dict(emptybarb=0.05, spacing=0.24, height=0.8, width=0.50)
```

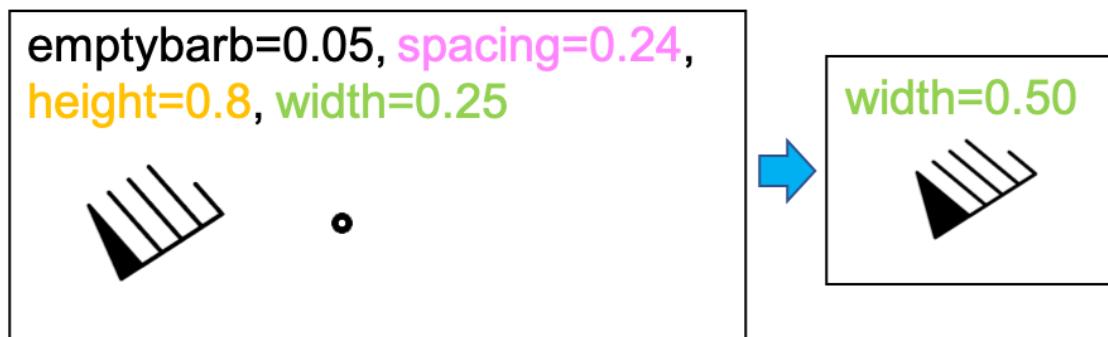


図 5 – 1 – 4 表示を変える

なお emptybarb は 0.0 に設定することもでき、その場合、風速が小さいと何も描かれないようになります。

矢羽の色を変更することもでき、plt.plot 等と同様に color オプションで行います。省略した c は無効です。色を変更する場合、例えば color='b' で全体の色を青色に変更できます（図 5 – 1 – 5）。color のデフォルト値は None ですが、

その場合には黒色 (color='k') に自動設定されます。

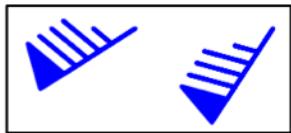


図 5-1-5 色を変更

色については細かい変更が可能なので、いくつか紹介しておきます。barbs_sample3.py は、色を図 5-1-6 のように青と緑で交互に変えるプログラムです。描かれた順（ここでは左から右）に色が変わります。

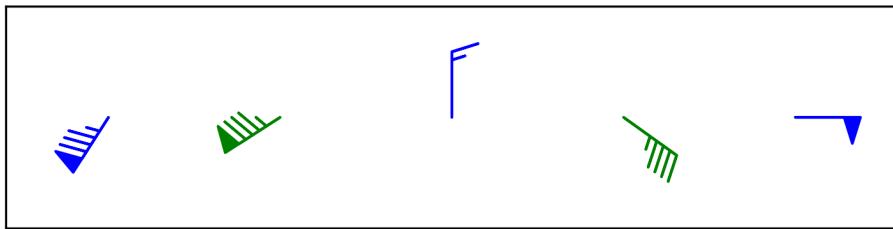


図 5-1-6 色を交互に変える

このように設定するには barbcolor オプションを使い、barbcolor=['b', 'g'] のように色を繰り返す順をリストとして渡します。なお、color オプションで color=['b', 'g'] としても同じになります。ちなみに、リストの要素を['b', 'g', 'k'] のように増やせば、青、緑、黒が繰り返されるようになります。

```
plt.barbs(x, y, u, v, barbcolor=['b', 'g'])
```

実用的ではありませんが、旗矢羽の内側の色だけを変える flagcolor というオプションもあり、flagcolor='r' とすれば、図 5-1-7 のように変わります。

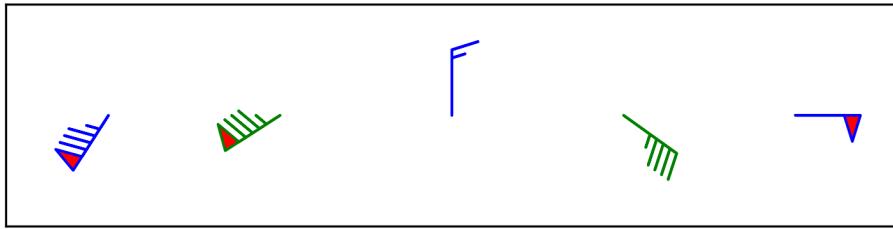


図 5－1－7 旗矢羽の内側の色を変更

作図に用いたのは barbs_sample4.py で、次のように作図しました。

```
plt.barbs(x, y, u, v, barbcolor=['b', 'g'], flagcolor='r')
```

なお、flagcolor は barbcolor と一緒に用いると、このように旗の内側だけ変更できますが、flagcolor 単独で用いた場合には、color オプション同様に矢羽全体の色を変更します。color オプションを指定した場合には、これらのオプションよりも color オプションが優先されます。

実際の使い道が想像できませんが、矢羽を軸の逆側に配置する flip_barb オプションもあります（図 5－1－8）。作図に用いたのは barbs_sample5.py です。デフォルトは flip_barb=False ですが、flip_barb=True で逆側に配置されます。

```
plt.barbs(x, y, u, v, barbcolor=['b', 'g'], flagcolor='r', flip_barb=True)
```

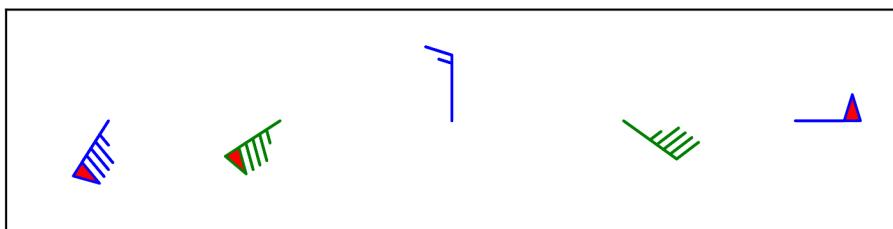


図 5－1－8 矢羽を軸の逆側に配置

他にも plt.plot 等と同様に不透明度 alpha (デフォルト値 1)、線の幅 linewidth (or lw、デフォルト値 1)、線のスタイル linestyle (or ls、デフォル

トは実線)が変更可能になっています。試してみると見栄えが悪くなるだけだったので、変更しない方が良いでしょう。

5.1.4 矢印を描く

矢羽の代わりに矢印を描くこともできて `plt.quiver(引数1, 引数2, 引数3, 引数4)` を使います。引数は `plt.barbs` の場合と同じで、左から順に x 軸上の位置、y 軸上の位置、東西風データ、南北風データが対応します。図 5-1-1 を作成した時と同じデータを使い、矢羽の代わりにベクトルを描きました(図 5-1-9)。作図に用いたプログラムは `quiver_sample.py` です。

```
plt.quiver(x, y, u, v) # 矢印を描く
```

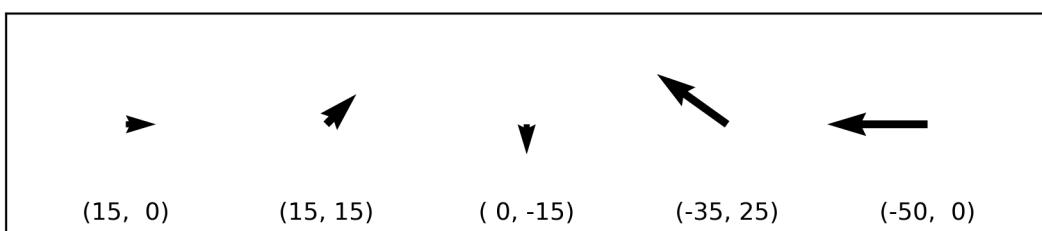


図 5-1-9 矢印を描く

5.1.5 矢印のスタイルを変更

矢印についても色の指定を行うことができ、`color` オプションを指定します(図 5-1-10)。省略した `c` は無効です。`quiver_sample2.py` で作図しました。

```
plt.quiver(x, y, u, v, color='b')
```

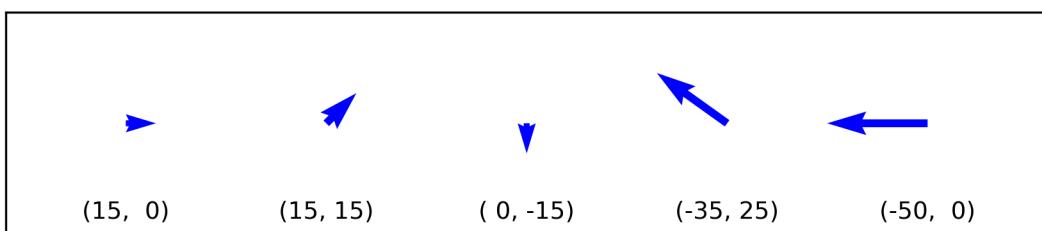


図 5-1-10 色を付けた

矢印を描いた場合と同様に、色を繰り返す順をリストとして渡すこともできて、図 5-1-11 のようになります。quiver_sample3.py で作図しました。

```
plt.quiver(x, y, u, v, color=['b', 'r', 'k'])
```

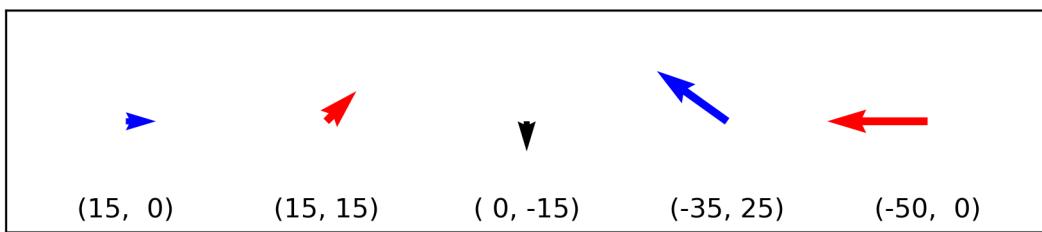


図 5-1-11 色を繰り返す順番を指定する

矢印の場合には、1 番目と 2 番目の引数とした与えた座標(x, y)に対して矢印の始点をどの位置に配置するのかを pivot オプションで指定することができますようになっています(図 5-1-12)。デフォルトは矢印の最後を表す pivot='tail' で、座標(x, y)を始点とします。

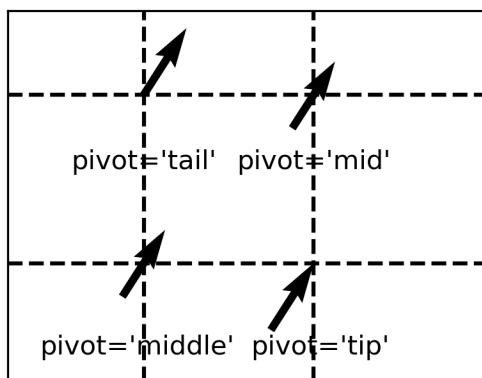


図 5-1-12 矢印の始点の位置を変える quiver の pivot オプション

矢印の幅を変えるには width オプションを使います。デフォルト値は width=0.005 です。矢印の幅を左から順に大きくしながら配置するようにしたものが図 5-1-13 です。quiver_sample4.py を使って作図しました。

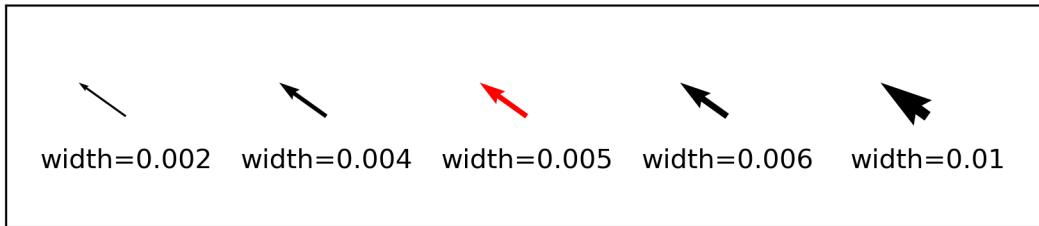


図 5 – 1 – 13 矢印の幅を変更、デフォルト値は赤で表示

矢印を描く際には、次のように width オプションを順に変更しています。与える width オプションの値を 2 倍程度まで大きくすると矢印の頭の部分が大きすぎ、半分程度まで小さくすると全体が判別できないので、あまり変更しない方が良いかもしれません。

```
plt.quiver(x[0], y[0], u[0], v[0], color='k', width=0.002)
plt.quiver(x[1], y[1], u[1], v[1], color='k', width=0.004)
plt.quiver(x[2], y[2], u[2], v[2], color='r', width=0.005) # デフォルト値
plt.quiver(x[3], y[3], u[3], v[3], color='k', width=0.006)
plt.quiver(x[4], y[4], u[4], v[4], color='k', width=0.01)
```

矢印の頭の部分の幅や長さを個別に設定することも可能です。矢印の頭の幅を指定するオプションは headwidth で、頭の長さは headlength、頭の中心軸部分の長さは headaxislength で指定します（図 5 – 1 – 14）。作図にはデフォルト値を用いていて、それぞれ、headwidth=3、headlength=5、headaxislength=4.5 です。矢印の頭の長さは 5、中心軸部分の長さは 4.5 で、中心軸部分の長さの方が少し短いので、頭の形は三角形よりも心棒側が少し削られた形をしています。

デフォルト値

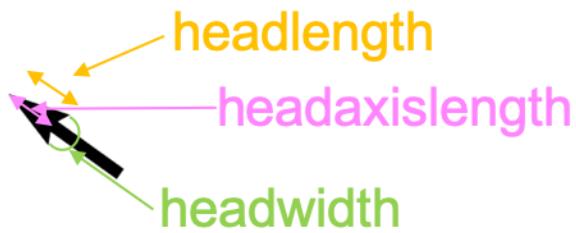


図 5 – 1 – 14 矢印のデフォルトスタイル
(`headwidth=3`、`headlength=5`、`headaxislength=4.5`)

まず `headwidth` を順に変えていくとどのようになるか見ていきます（図 5 – 1 – 15）。`quiver_sample5.py` を使って作図しました。デフォルトの 3 より横幅を小さくしていくと、`headwidth=2` で矢印の頭が判別できず、デフォルトより大きくしていくと、`headwidth=4~5` では頭でっかちな矢印に見えてしまします。デフォルトの 3 からあまり変えない方が良さそうです。

```
plt.quiver(x[0], y[0], u[0], v[0], color='k', headwidth=1)
plt.quiver(x[1], y[1], u[1], v[1], color='k', headwidth=2)
plt.quiver(x[2], y[2], u[2], v[2], color='r', headwidth=3)
plt.quiver(x[3], y[3], u[3], v[3], color='k', headwidth=4)
plt.quiver(x[4], y[4], u[4], v[4], color='k', headwidth=5)
```



headwidth=1 headwidth=2 headwidth=3 headwidth=4 headwidth=5

図 5 – 1 – 15 `headwidth` の値を徐々に大きくした場合、デフォルト値は赤で表示

次に、矢印の長さ（`headlength`）についても見ていきます。作図には `quiver_sample6.py` を使いました。この値を変更した場合、矢印の頭の外側部

分の長さが徐々に変わっています(図5-1-16)。ここでは頭の中心軸部分の長さ headaxislength は、いずれもデフォルトの4.5のままにしているので、中心軸部分の長さは不变です。そのため headlength に headaxislength より小さい値を使うと、矢印の頭の外側の長さが頭の中心軸の長さよりも短くなりますが、headlength=3~4 のケースで示されているように矢印の頭の形が変になるので使わない方が良いでしょう。headlength が headaxislength より大きい場合は、矢印の頭の外側が中心軸よりも長くなるため、矢印の頭の両端が鋭くなります。好みの問題ですが、headlength=6 くらいまでは許容範囲に思えます。

```
plt.quiver(x[0], y[0], u[0], v[0], color='k', headlength=3)
plt.quiver(x[1], y[1], u[1], v[1], color='k', headlength=4)
plt.quiver(x[2], y[2], u[2], v[2], color='r', headlength=5)
plt.quiver(x[3], y[3], u[3], v[3], color='k', headlength=6)
plt.quiver(x[4], y[4], u[4], v[4], color='k', headlength=7)
```

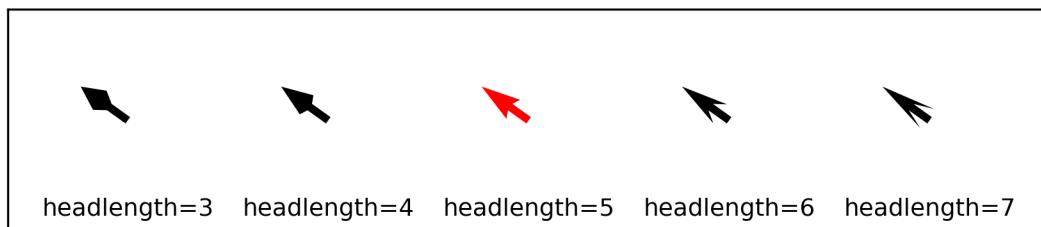


図5-1-16 headlength の値を徐々に大きくした場合、デフォルト値は赤で表示

頭の中心軸部分の長さ (headaxislength) についても、順に変更してみます(図5-1-17)。作図には quiver_sample7.py を使いました。頭の外側の長さはデフォルトの headlength=5 にしています。そのため、中心軸部分の長さを同じ headaxislength=5 にした場合に、頭の部分が二等辺三角形になっています。デフォルトの5よりも中心軸部分の長さを短くしていくと、中心軸が外側よりも短くなるため矢印の頭の両端が鋭くなります。headaxislength=3 では矢印の頭の両端が鋭くなりすぎているように見えます。中心軸部分の長さを5よりも長くしていくと、中心軸が外側よりも長くなるのでheadaxislength=6 のように矢印の頭の形が変になります。headaxislength=4~5 (あるいは headlength-

$1 \leq \text{headaxislength} \leq \text{headlength}$) くらいまでが許容範囲に思えます。

```
plt.quiver(x[0], y[0], u[0], v[0], color='k', headaxislength=3)
plt.quiver(x[1], y[1], u[1], v[1], color='k', headaxislength=4)
plt.quiver(x[2], y[2], u[2], v[2], color='r', headaxislength=4.5)
plt.quiver(x[3], y[3], u[3], v[3], color='k', headaxislength=5)
plt.quiver(x[4], y[4], u[4], v[4], color='k', headaxislength=6)
```

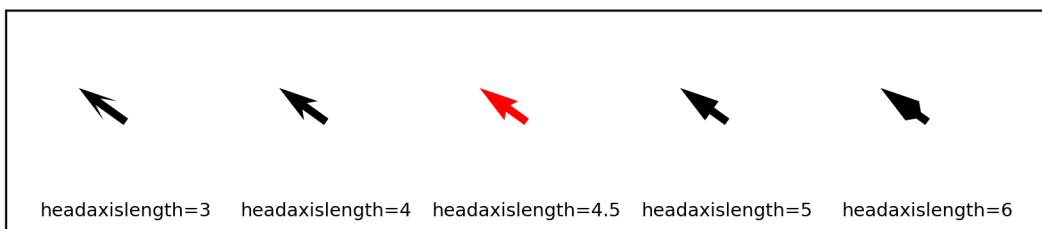


図 5 – 1 – 17 headaxislength の値を徐々に大きくした場合、デフォルト値は赤で表示

他には、矢印が削られずに心棒が表示される下限値 minshaft や、矢印を描く値の最小値 minlength というオプションもあります。デフォルト値はいずれも minshaft=1、minlength=1 です。図 5 – 1 – 18 は左側から順に矢印の作図に用いる東西風速 u の値を 5 から 25 まで大きくしていったものです。上下方向の棒では minshaft のみ変更して比較しました。作図には quiver_sample8.py を使いました。minshaft=0.5 では、一番左側の矢印がおかしくなっています。minshaft=1 より小さい値は使わない方が良いでしょう。minshaft=2 とした場合には、u=5 ~ 15 程度の矢印の幅が細くなり矢印全体に占める心棒の割合が大きくなりました。minshaft=5 まで大きくしてしまうと u=5 ~ 25 の矢印が全て細くなってしまうので、u=5 ~ 25 程度であれば minshaft=1 ~ 2 が許容範囲のようです。

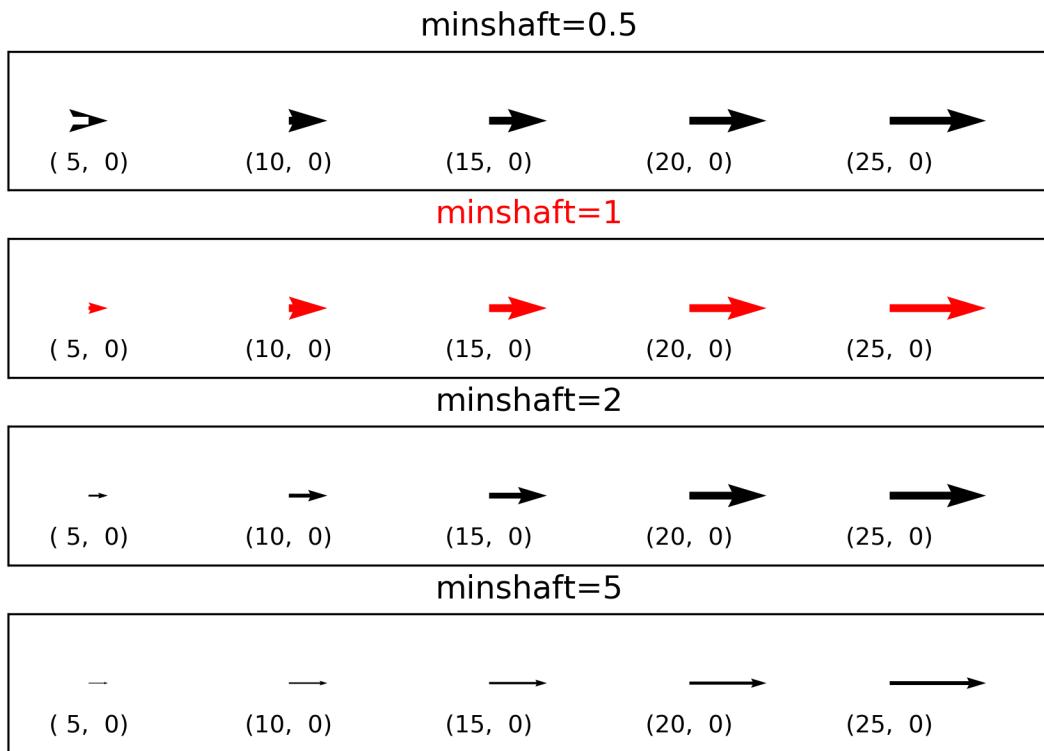


図 5 – 1 – 18 矢印の作図に用いる値を左から順に大きくした。上下方向は `minshaft` を変えた比較。デフォルト値は赤で表示

サンプルプログラムでは、縦方向に 4 つの図を並べるためにサブプロットで分割しました。プログラムの最初では、それぞれのサブプロットで作図する際のオプションを `STYLES` で辞書をリストに入れて定義しています。

```
STYLES = [
    dict(minshaft=0.5, color='k'),
    dict(minshaft=1, color='r'),
    dict(minshaft=2, color='k'),
    dict(minshaft=5, color='k'),
]
```

プロットエリアにサブプロットを並べていきますが、サブプロット生成部分をループの中に入れています。ループの番号は `np.arange(len(STYLES))` で決めているので、`STYLES` の長さ（ここでは 4）が入り、 $n=0, 1, 2, 3$ のルー

普となります。サブプロットの番号は1から始まるので、fig.add_subplotに与える3つの引数は、len(STYLES)、1、n+1となっています。タイトルとしては、STYLESでminshaftをキーとして作成した辞書のitemを取り出して文字型に変換して使ってています。それがstr(STYLES[n]['minshaft'])の部分で、STYLES[n]でリストの要素（ここでは辞書）を取り出し、辞書['minshaft']でminshaftをキーとするitemを取り出しています。色についても同様で、STYLES[n]['color']を使ってcolorオプションに与える色を取り出しています。

```
fig = plt.figure(figsize=(7, 5)) # プロットエリアの定義
ax = list() # サブプロットを参照する軸のリスト作成
for n in np.arange(len(STYLES)):
    ax.append(fig.add_subplot(len(STYLES), 1, n+1)) # サブプロット定義
    # x 軸の範囲
    plt.xlim([0.9, 2.2])
    # タイトルを付ける
    plt.title("minshaft=" + str(STYLES[n]['minshaft']), color=STYLES[n]['color'])
    # 矢羽を描く
    plt.quiver(x, y, u, v, **STYLES[n])
```

さらに軸のリストからサブプロットを参照してx軸、y軸ともに主目盛、副目盛を消します。

```
# x 軸の目盛り
ax[n].xaxis.set_major_locator(ticker.NullLocator())
ax[n].xaxis.set_minor_locator(ticker.NullLocator())
# y 軸の目盛り
ax[n].yaxis.set_major_locator(ticker.NullLocator())
ax[n].yaxis.set_minor_locator(ticker.NullLocator())
```

次にminlengthについても見ていきます（図5-1-19）。quiver_sample9.pyを使って作図しました。minlength=1では、u≤2の矢印が描かれないようになりますが、ここではuは5以上ですので全て描かれています。minlength=0.5

でも同じです。minlength=5 では左側 2 つの矢印が、minlength=10 では左側 4 つの矢印が描かれないようになりました。矢印が描かれない場合には、マークが六角形になってしまふので見栄えが悪いかもしれません。

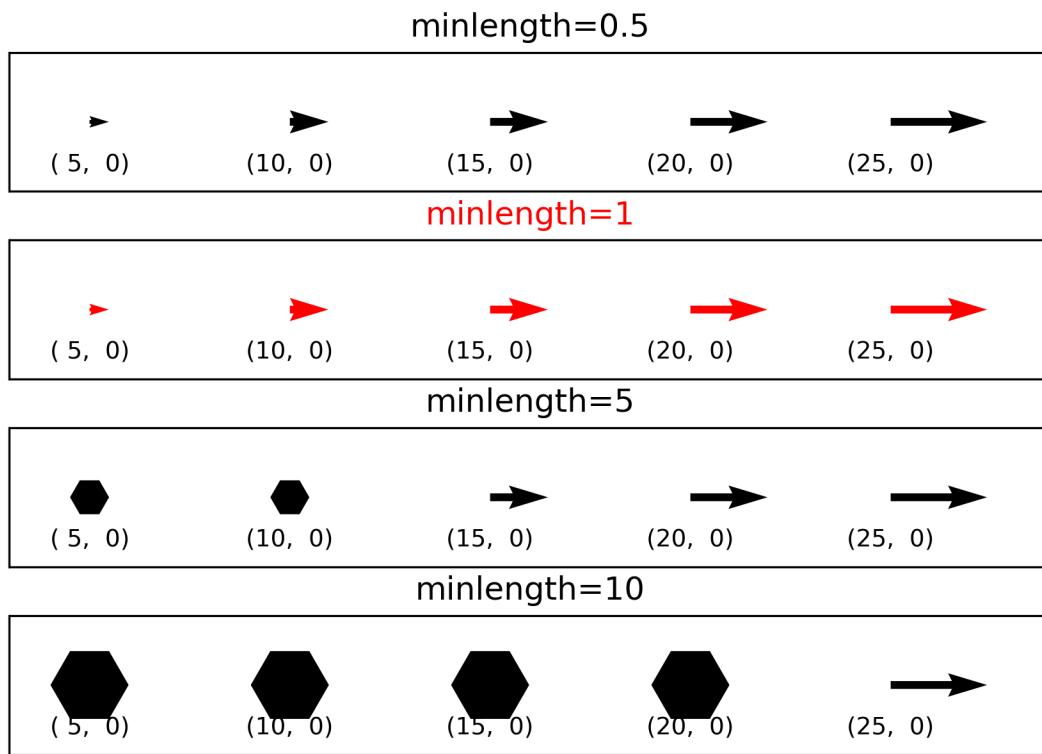


図 5 – 1 – 19 上下方向は minlength を変えた比較。デフォルト値は赤で表示

サンプルプログラムでは、作図の際に与える STYLES を次のように変更して minshaft の値を反映させています。

```
STYLES = [
    dict(minlength=0.5, color='k'),
    dict(minlength=1, color='r'),
    dict(minlength=5, color='k'),
    dict(minlength=10, color='k'),
]
```

矢印を描く基準値となるスケールを変えることもでき、scale で行います。デ

デフォルト値は None です。また scale_units というオプションもあり、矢印の x 成分 u と y 成分 v の両方を使う'xy'が扱いやすいです。scale を変えて比較したものを図 5-1-20 にしました。作図には quiver_sample10.py を使いました。scale を大きくするほど矢印は小さくなっています。デフォルト値の None を使った場合、この u、v 値では scale=200 程度に調整されていたようです。

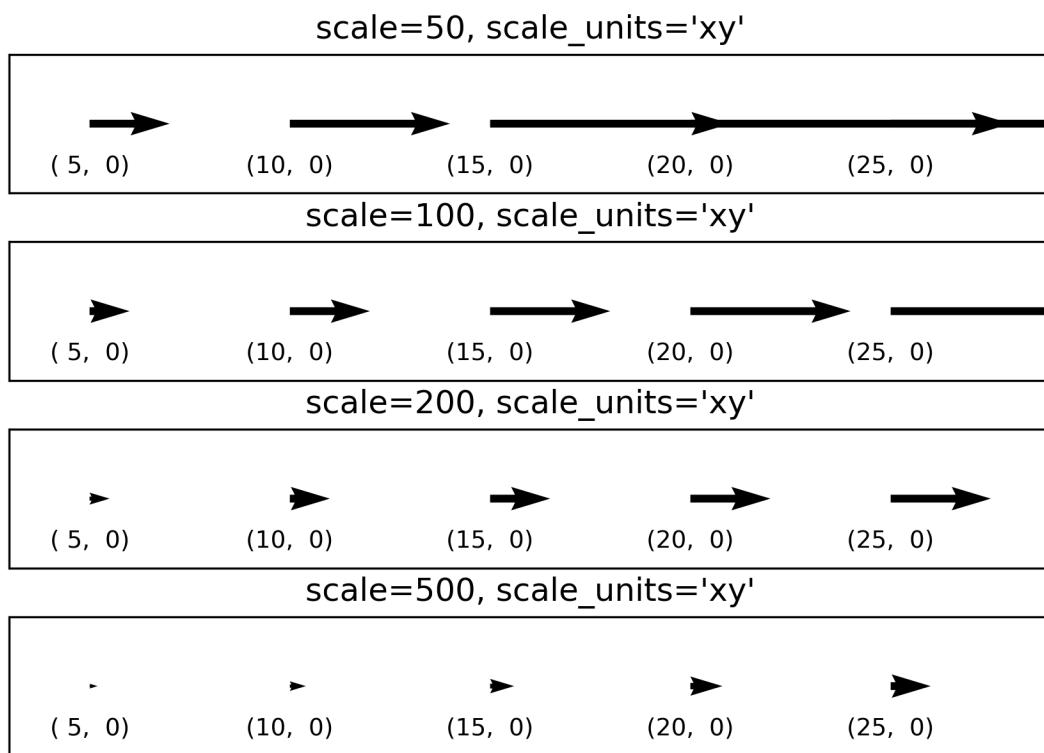


図 5-1-20 上下方向は scale を変えた比較

```
STYLES = [
    dict(color='k', scale=50, scale_units='xy'),
    dict(color='k', scale=100, scale_units='xy'),
    dict(color='k', scale=200, scale_units='xy'),
    dict(color='k', scale=500, scale_units='xy'),
]
```

凡例を付けることもでき、plt.plot 等の作図同様 label オプションを指定し、その後で plt.legend()を行います。しかし、この方法では凡例に矢印の色が矩形

で表示されるだけで、矢印の形までは表示されません。

なお plt.plot 等と同様に線の幅 linewidth (or lw、デフォルト値 1)、線のスタイル linestyle (or ls、デフォルトは実線) も与えることができ、エラーにはなりませんが、矢印のスタイルは変更されないので使用する意味はありません。

5.1.6 矢羽と矢印のまとめ

最後に矢羽と矢印の作図で使用可能なオプションをまとめておきます（表 5 – 1 – 1、表 5 – 1 – 2）。ここでは、矢羽と矢印を 1 次元でしか描いていませんが、2 次元平面上に描くことも可能です。2 次元平面上での作図については、6 章の basemap の所で紹介します。

表 5 – 1 – 1 matplotlib の pyplot.barbs で使用可能なオプション一覧

オプション	説明
x, y	x軸, y軸に使用する配列
u, v (必須)	東西風データ、南北風データの配列
color	矢羽全体の色か色順のリスト、デフォルト値：None（黒色に自動設定される）
barbcolor	矢羽全体の色か色順のリスト、デフォルト値：None
flagcolor	旗の内側の色、barbcolorを設定しない場合は矢羽全体の色、デフォルト値：None
length	矢羽のサイズ、デフォルト値：8
barb_increments	矢羽を描く基準風速の辞書、half：短矢羽、full：長矢羽、flag：旗矢羽 デフォルト値：{half: 5.0}、{full: 10.0}、{flag: 50.0}
edgecolor	枠線の色、デフォルト値：None
sizes	矢羽の各部分のサイズ、emptybarb：シンボル（円）の大きさ、spacing：矢羽同士の間隔、height：矢羽の横方向の長さ、width：旗矢羽の幅 デフォルト値：{emptybarb: 0.15}、{spacing: 0.12}、{height: 0.4}、{width: 0.25}
alpha	不透明度、デフォルト値：1.0
linewidth or lw	線の太さ、デフォルト値：1
linestyle or ls	線の種類、デフォルト値：'-'
label	凡例を付ける場合（凡例では矢羽の色のみ表示）、デフォルト値：None

使用方法：plt.barbs(x, y, u, v, オプション)： 矢羽作成、plt.barbs(u, v, オプション)：矢羽作成（位置指定なし）

表 5 – 1 – 2 matplotlib の pyplot.quiver で使用可能なオプション一覧

オプション	説明
x, y	x軸, y軸に使用する配列
u, v (必須)	東西風データ、南北風データの配列
color	矢羽全体の色、デフォルト値 : None (黒色に自動設定される)
units	矢印を描く単位、{'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'} width, height : 軸の幅か高さにより矢印のサイズを変更 dots, inches : 図の解像度により変わるピクセルやインチで矢印のサイズを変更 x, y, xy : データの単位でX, Y、 $XY=(X^2+Y^2)^{1/2}$ で矢印のサイズを変更
angles	矢印のアングルを決める方法、{'uv', 'xy'}、デフォルト値 : uv uv : 東西風・南北風の大きさ、xy : (x,y)から(x+u, y+v)に矢印を描く
scale	矢印を描く基準値で指定、デフォルト値 : None
scale_units	矢印を描く基準の単位、{'width' 'height' 'dots' 'inches' 'x' 'y' 'xy'}、デフォルト値 : None
width	矢印の幅、デフォルト値 : 0.005
headwidth	矢印の頭の幅、デフォルト値 : 3
headlength	矢印の頭の長さ、デフォルト値 : 5
headaxislength	矢印の頭の中心軸部分の長さ、デフォルト値 : 4.5
minshaft	これ以下の矢印が削られる長さ、デフォルト値 : 1、1より小さな値は推奨されない
minlength	矢印を描く値の最小値、デフォルト値 : 1
pivot	矢印の開始点、{'tail', 'mid', 'middle', 'tip'}、デフォルト値 : 'tail'
alpha	不透明度、デフォルト値 : 1.0
label	凡例を付ける場合 (凡例では矢羽の色のみ表示)、デフォルト値 : None

使用方法 : plt.quiver(x, y, u, v, オプション) : 矢印作成、plt.quiver (u, v, オプション) : 矢印作成 (位置指定なし)

5.1.7 実際のデータを使った矢羽の作図

矢羽の作成方法を学んだところで、実際のアメダスデータを使った作図を試してみましょう。2018年9月4日には、台風21号が25年ぶりとなる非常に強い勢力で12時頃に徳島県南部に上陸し、大阪湾を抜けて14時頃に神戸に上陸する経路を辿りました。関西国際空港では、高潮と越波による浸水被害と連絡橋へのタンカー衝突事故が起こっていました。この日に関西国際空港のアメダス地点で観測された風向風速データを使って矢羽を描き1時間毎の時間変化を並べて見ていきます(図5-1-20)。作図に用いたのは amedas_wind.py です。14時に35m/sの平均風速となっています。その時刻を境に東よりの風が南西よりの風に変化しており、台風が関西国際空港付近の西側を通過したことが分かります。

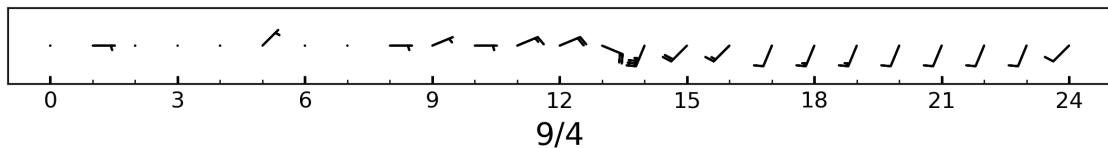


図5-1-20 関西空港のアメダス地点で観測された2018年9月4日の風向風速（短矢羽：5m/s、長矢羽：10m/s）

関西国際空港のアメダス地点データは、20180903-20180905KIX.csv です。ファイルは csv データになっており、Pandas の read_csv で読み込みます。1列目が時刻、2列目が気温、3列目が風速、4列目が風向になっています。時刻データは parse_dates=[0] と index_col=[0] で読み込まれ、Pandas の DataFrame である dat_i の行方向の index になります。その時、気温は 0 列目、風速は 1 列目、風向は 2 列目のデータとして dat_i に格納されます。

```
input_file = "20180903-20180905KIX.csv"  
dat_i = pd.read_csv(input_file, parse_dates=[0], index_col=[0])
```

風向のデータは、「北北東」など日本語の16方位データになっているので、数値に変換します。北風が0度で時計回りに北北東が22.5度、北東が45度、…、南が180度、…、西が270度のように変換します。まず列番号でデータを取

り出すため、DataFrame のメソッド.iloc を使い、dat_i.iloc[:, 2]のように風向データを取り出します。さらに、メソッド str.replace(元の文字列, 置き換える文字列)を使って文字列を数字に置き換えます。この操作はリストの要素数分のループを回して行っており、リストの先頭から順に置き換えの操作を行っています。「北」と「北東」、「北北東」のように同じ文字が複数使われているので、文字数が多い方から順に置き換えの操作を行うことで、どの風向も問題なく変換できるようにしています。

```
list_org = ["静穏", "北北東", "東北東", "東南東", "南南東", "南南西", ¥  
"西南西", "西北西", "北北西", "北東", "南東", "南西", "北西", ¥  
"北", "東", "南", "西"] # 元データの風向  
list_new = ["NaN", "22.5", "67.5", "112.5", "157.5", "202.5", ¥  
"247.5", "292.5", "337.5", "45.0", "135.0", "225.0", "315.0", ¥  
"0.0", "90.0", "180.0", "270.0"] # 変換する風向  
for i in range(len(list_org)): # 風向の書き換え  
    dat_i.iloc[:,2] = dat_i.iloc[:, 2].str.replace(list_org[i], list_new[i])
```

次に、文字列データを全て数値データに変換して新しい DataFrame の dat に格納します。その際に、読み込んだデータを倍精度浮動小数点型に変換するため、dtype="float64"としています。

```
dat = DataFrame(dat_i.copy(), dtype="float64") # データ型の変換
```

作図を行う際に必要な形式にするため、開始時刻(stime)から終了時刻(etime)までの風速データ(ws)、風向データ(wd)を取り出します。またデータの長さ(length)を取得しておきます。

```
stime = "2018-09-04 00:00:00" # データの開始  
etime = "2018-09-05 00:00:00" # データの終了  
length = len(dat.loc[stime:etime,:]) # データの長さを取得  
ws = dat.loc[stime:etime, "ws"].copy() # 風速データ取り出し  
wd = dat.loc[stime:etime, "wd"].copy() # 風向データ取り出し
```

矢羽を描く際には、風向風速データではなく東西風速、南北風速データが必要になるため、 $(u, v) = (ws * \cos(270 - wd), ws * \sin(270 - wd))$ の変換を行います。x、y 平面上では、東向きが 0 度で反時計回りに角度が増加します。そのため、北風（南向きの風）が 270 度に、東風（西向きの風）が 180 度に、南風（北向きの風）が 90 度に、西風（東向きの風）が 0 度となるように変換しています。

```
u = np.array(ws * np.cos((270.0 - wd)/180.0 * np.pi)) # 東西風速に変換  
v = np.array(ws * np.sin((270.0 - wd)/180.0 * np.pi)) # 南北風速に変換
```

作図部分です。先ほど取得した length を使い、矢羽を描く x、y 座標を $(0, 5)$ 、 $(1, 5)$ 、、、 $(length-1, 5)$ のようにして、x、y の配列にしておきます。なお np.arange(length) は 0 から length-1 までの整数を生成し、np.ones(length) は length の長さを持ち要素が全て 1 の配列を生成します。これら x、y 座標のデータと東西南北風速データ u、v を使い、ax.barbs で矢羽を作図します。

```
ax.set_xlim([-1, length]) # x 軸の範囲  
ax.set_ylim([0, 10]) # y 軸の範囲  
x = np.arange(length) # x 座標  
y = np.ones(length) * 5 # y 座標  
# 矢羽の作図  
ax.barbs(x, y, u, v, color='k', length=4.5, sizes=dict(emptybarb=0.001))
```

x 軸の大目盛り線を 3 時間毎、小目盛り線を 1 時間毎につけるため、ticker.MultipleLocator を使います。

```
ax.xaxis.set_major_locator(ticker.MultipleLocator(3)) # x 軸大目盛り  
ax.xaxis.set_minor_locator(ticker.MultipleLocator(1)) # x 軸小目盛り
```

5.2 等高線を描く

matplotlib には、等高線を描く機能も含まれています。気象データの作図では地図と一緒に使うことが多いので、ここでは基本的な使い方を紹介し、6章で Basemap を 7 章で cartopy を用いて実際の気象データを作図する方法を詳述しています。

5.2.1 等高線を描く準備

等高線を描くには `plt.contour(引数 1, 引数 2, 引数 3)` を使います。1 番目の引数が x 軸上の位置、2 番目の引数が y 軸上の位置で、3 番目の引数が等高線を描くための z 軸データに当たります。これまでとは異なり、x—y 平面上に値をプロットするため、x 軸上、y 軸上の位置は x—y 平面上の格子点全てについて (x, y) の組み合わせで与えるようにする必要があります。そのため、1 番目の引数 (X) 、2 番目の引数 (Y) とも 2 次元配列で作成し、 X 、 Y 配列の各要素 (X_{ij}, Y_{ij}) が格子点の座標に対応するようにします。等高線を描く前にそのような格子点の作り方の例を挙げておきます (`contour_sample.py`)。

まず x、y 軸のサンプルデータを 1 次元で作成します。`np.linspace` を使い、x 軸上では $(0, 1, 2)$ 、y 軸上では $(0, 100, 200)$ のデータを生成します。

```
# x, y 軸サンプルデータ
x = np.linspace(0, 2, 3)
y = np.linspace(0, 200, 3)
print("x = ", x)
print("y = ", y)
```

出力：

```
x =  [0. 1. 2.]
y =  [ 0. 100. 200.]
```

x 軸上の 1 次元データ $(x=\{x_i\})$ 、y 軸上の 1 次元データ $(y=\{y_j\})$ から x—y 平面上の 2 次元メッシュデータ $(X=\{X_{ij}\}, Y=\{Y_{ij}\})$ を作成する際には、`np.meshgrid` を使います。出力の X、Y データを組み合わせると、 $(0, 0), (1, 0), \dots, (1, 200), (2, 200)$ のように x—y 平面上における格子点の座標になります。

```
# x, y 軸メッシュデータ  
X, Y = np.meshgrid(x, y)  
print("X = ")  
print(X)  
print("Y = ")  
print(Y)
```

出力：

```
X =  
[[0. 1. 2.]  
 [0. 1. 2.]  
 [0. 1. 2.]]  
Y =  
[[ 0.  0.  0.]  
 [100. 100. 100.]  
 [200. 200. 200.]]
```

生成された2次元メッシュデータを使い、x—y平面上における格子点の座標に黒丸をプロットします（図5-2-1）。

```
# 格子点をプロット  
plt.plot(X, Y, marker='o', ls='', color='k')  
# グリッド線を付ける  
plt.grid(color='gray', ls='--')
```

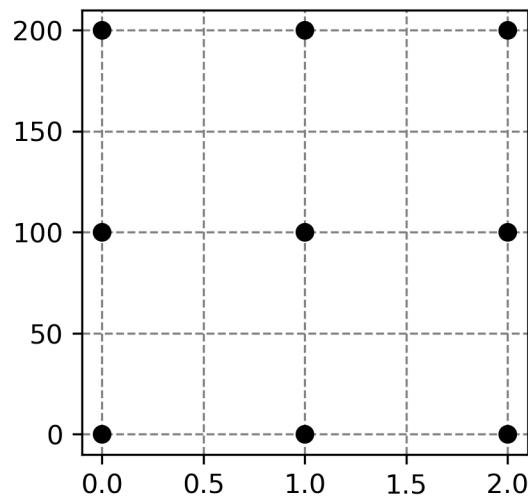


図 5 – 2 – 1 x—y 平面上に 2 次元メッシュデータをプロット

5.2.2 基本的な等高線

等高線を描く場合には、これら x—y 平面上の格子点に対して z の値を与える必要があります。contour_sample2.py では、そのような z の値を座標 (0, 0) からの距離 $Z=(X^2+Y^2)^{1/2}$ で与えています。先ほどのような 9 点では等高線を描くには少なすぎるので、x 軸方向に 200、y 軸方向に 200 のデータを作成します。そのため、生成される格子点は 200×200 となります。

```
n = 200 # サンプル数
x = np.linspace(-10, 10, n) # x 軸データ
y = np.linspace(-10, 10, n) # y 軸データ
# x, y 軸メッシュデータ
X, Y = np.meshgrid(x, y)
# z 軸データ
Z = np.sqrt(X**2 + Y**2)
```

このデータを使い等高線を作図します（図 5 – 2 – 2）。plt.contour(X, Y, Z) で等高線が描かれます。ax.contour(X, Y, Z) でも同じです。この状態では縦長になっていて見た目が悪いと思います。これは figsize=(3,5) のように縦横比が 1 : 1 ではないプロットエリアに作図したためです。

```
fig=plt.figure(figsize=(3,5)) # プロットエリアの定義  
ax=fig.add_subplot(1,1,1) # サブプロット生成  
plt.contour(X, Y, Z) # 等高線をプロット
```

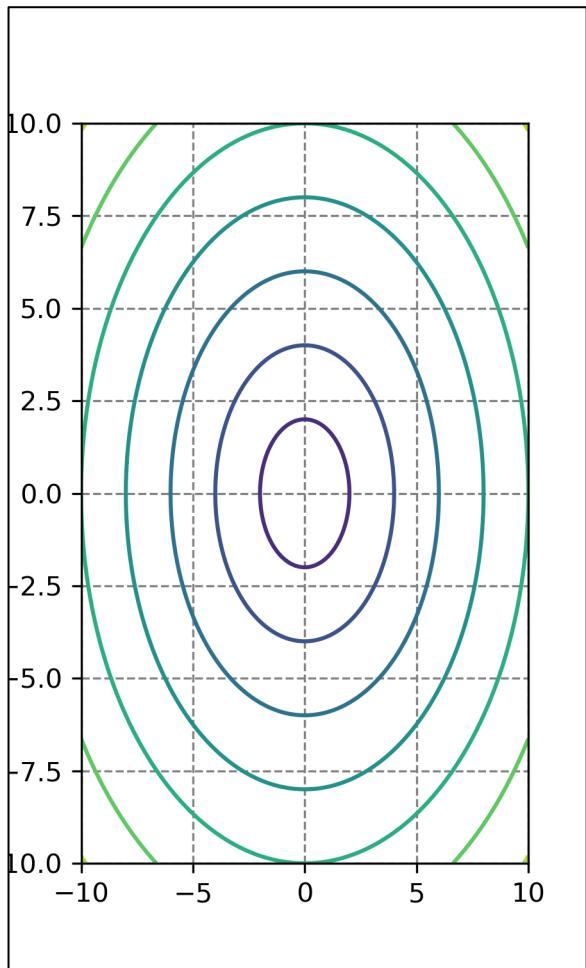


図 5－2－2 等高線を描く（縦長になってしまう）

縦横の比率を 1 : 1 にするには、`plt.gca().set_aspect('equal')` を使います (`contour_sample3.py`)。図 5－2－3 のように、プロットエリアに関係なく縦横比が 1 : 1 に変更されました。

```
plt.gca().set_aspect('equal') # 縦横比を 1 : 1
```

なお図を保存する際に `bbox_inches='tight'` を使っていると、プロットエリアの空白が削除されて分かりにくいので、図 5-2-2 と図 5-2-3 の作図ではデフォルトのままで作図し、プロットエリアの端に枠を付けたものを載せました。

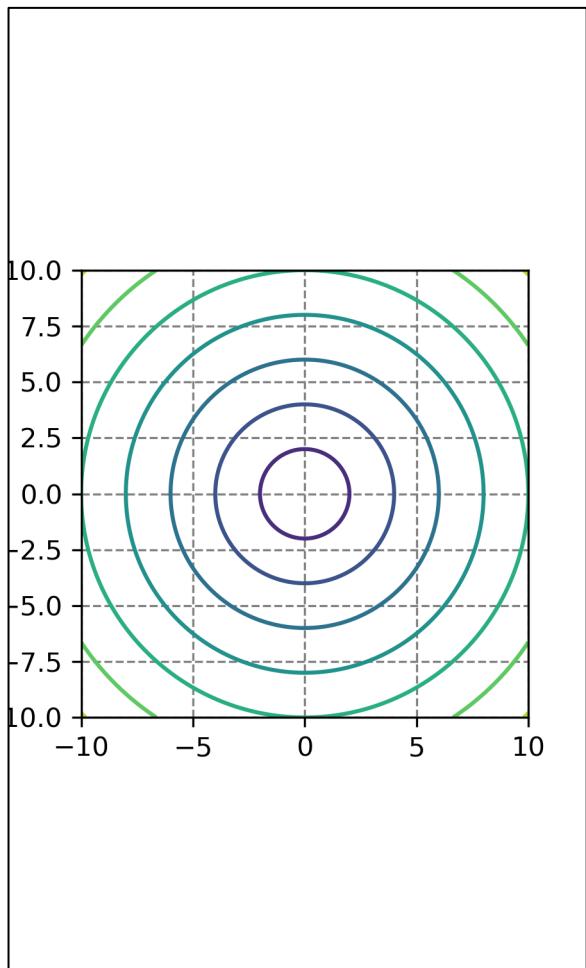


図 5-2-3 縦横の比率を 1 : 1 にした

5.2.3 等高線の色と等高線を付ける間隔

等高線の色は、デフォルトでは z の値の範囲をチェックして z の最小値から最大値の範囲でグラデーションになるような色が使われます。等高線を付ける間隔も、デフォルトでは z の値の範囲から自動的に設定されます。等高線の色や等高線を付ける間隔は、オプションで設定することも可能です。

まずは色を変更する方法です。色は `colors='k'` のように設定します (`contour_sample4.py`)。等高線が黒色に変わりました (図 5-2-4)。等高線の場合は `color` ではなく `colors` なので注意が必要です。

```
plt.contour(X, Y, Z, colors='k')
```

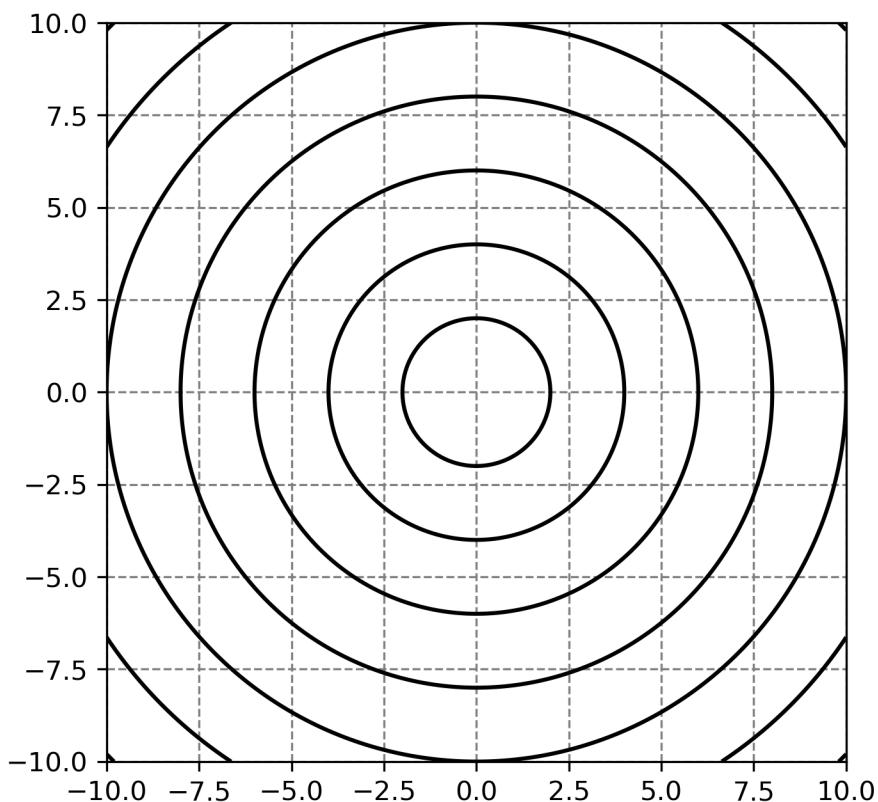


図 5-2-4 等高線の色を黒に設定

`colors` に与える色は単色だけではなく、順番に色が変わるように与え方も可能です。その場合、色を並べる順をリストで与えます (`contour_sample5.py`)。ここでは `colors=['b', 'r', 'k']` のように与えたので、青、黒、赤の順で繰り返され

る等高線が描かれました（図 5-2-5）。

```
plt.contour(X, Y, Z, colors=['b', 'r', 'k'])
```

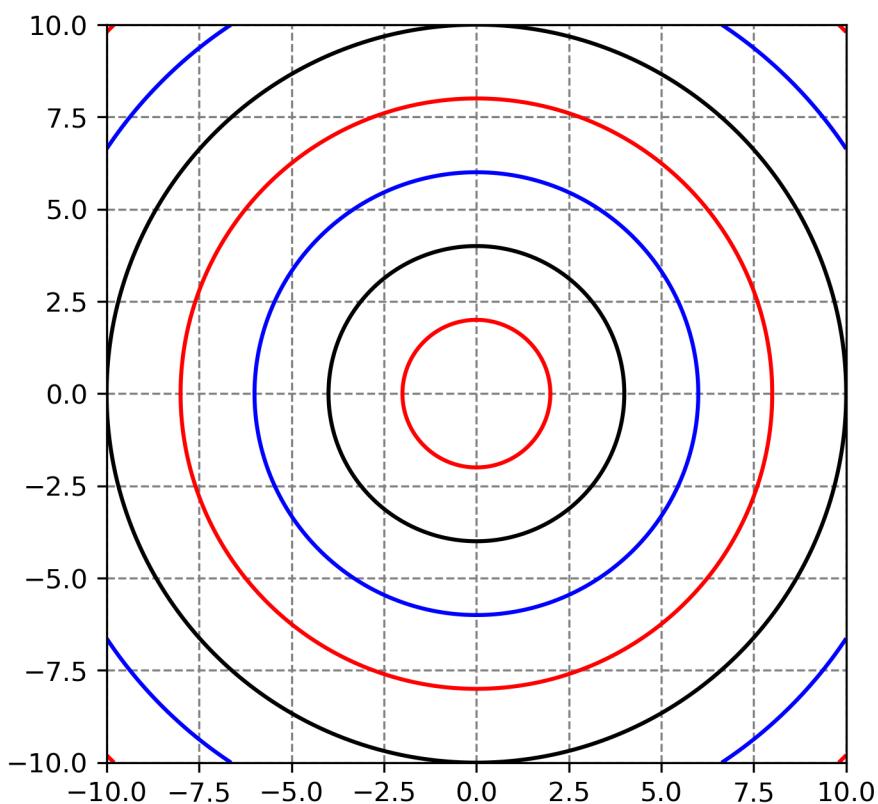


図 5-2-5 色順を指定する

次に等高線の間隔を指定する方法です。plt.contour には等高線を描く値を指定するための levels オプションがあります。等高線を描きたい Z の値をリストにし、levels オプションに与えます (contour_sample6.py)。このプログラムでは、1、2、5、10、12 を levels というリストに入れ、等高線を描く際に plt.contour のオプションとして levels=levels でリストを渡しました。ややこしいですが、オプション名の levels とリストの levels は別のものとして解釈されています。図 5-2-6 のように指定した値の所に等高線が描かれます。

```

levels = [1, 2, 5, 10, 12] # 等高線の値のリスト
plt.contour(X, Y, Z, colors='k', levels=levels) # 等高線を描く

```

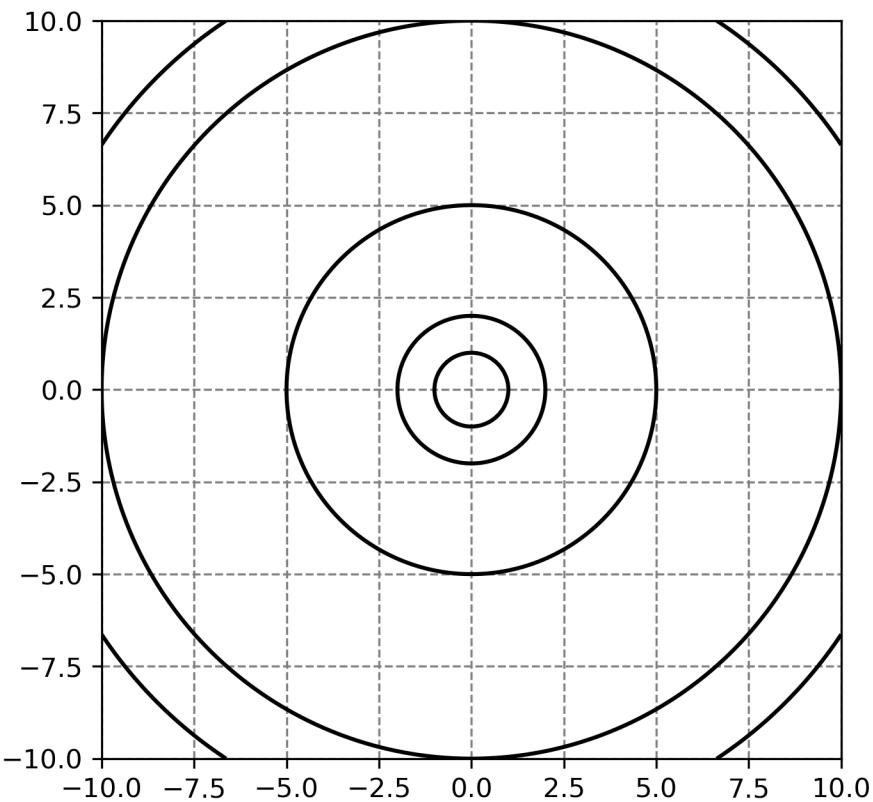


図 5 – 2 – 6 等高線の値を 1、2、5、10、12 に指定した

実用上は等高線を等間隔に描きたいこともあると思います。そのような作図を行うプログラムが、`contour_sample7.py` です。ここでは Numpy の機能を使い、Z の値の最小値を `Z.min()`、最大値を `Z.max()` のように求めます。さらに最小値よりも小さい最大の整数を `np.floor(Z.min())`、最大値よりも大きい最小の整数を `np.ceil(Z.max())` で計算し、その範囲内で 1.5 毎にリストを生成します。図 5 – 2 – 7 のように、1.5 每に等間隔の等高線が描かれます。

```

levels = np.arange(np.floor(Z.min()), np.ceil(Z.max()), 1.5) # 値のリスト作成
plt.contour(X, Y, Z, colors='k', levels=levels) # 等高線を描く

```

出力：

```
[ 0.   1.5   3.   4.5   6.   7.5   9.   10.5  12.   13.5]
```

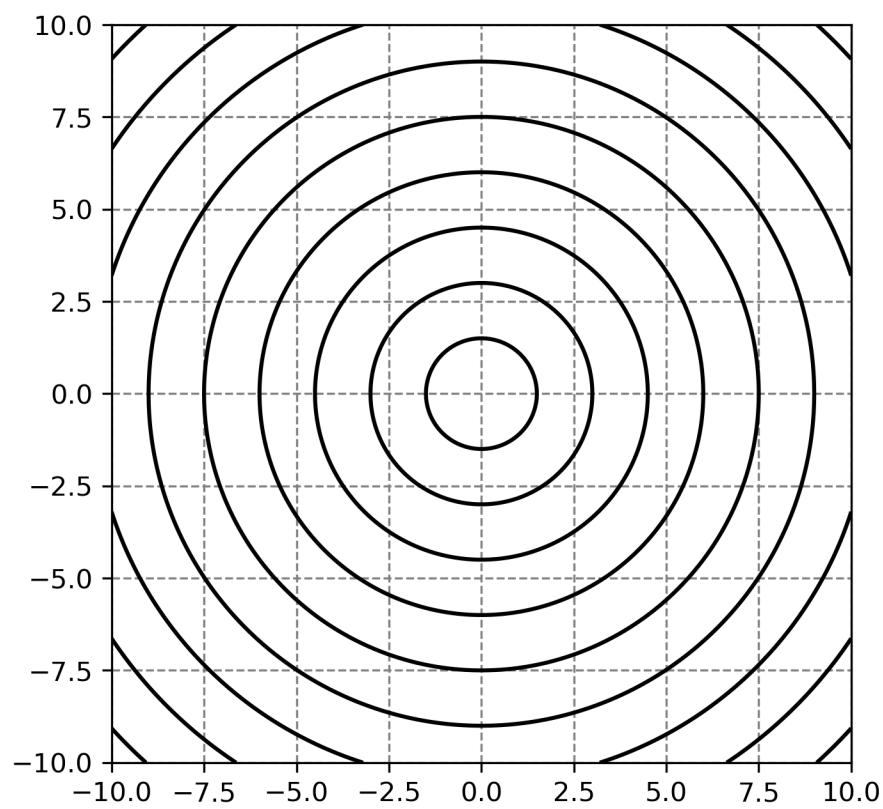


図 5－2－7 1.5 毎に等間隔の等高線を描く

5.2.4 等高線にラベルを付ける

図5-2-8のように等高線にラベルを付けることも可能で、`plt.contour` の戻り値を利用します。作図を行うプログラムが、`contour_sample8.py` です。まず `plt.contour` を行う際に戻り値を `cs` に格納します。`cs` に対して `cs.clabel(オプション)` を行うことによってラベルが生成されます。`fontsize` で文字の大きさ、`fmt` で文字のフォーマットを指定します。ここでは小数点以下第1位まで表示したかったので `fmt="%.1f"` としました。`fmt` に与える書式は、4.5.7節で出てきましたが、`"%.1f"` のように%の後に記述する古い書式を用います。`plt.contour` の戻り値がインスタンスになっているため、`cs.clabel` のようなインスタンスマソッドが利用できます。

```
cs = plt.contour(X, Y, Z, colors='k', levels=levels) # 戻り値 cs  
cs.clabel(fmt=".1f", fontsize=12) # ラベルを付ける
```

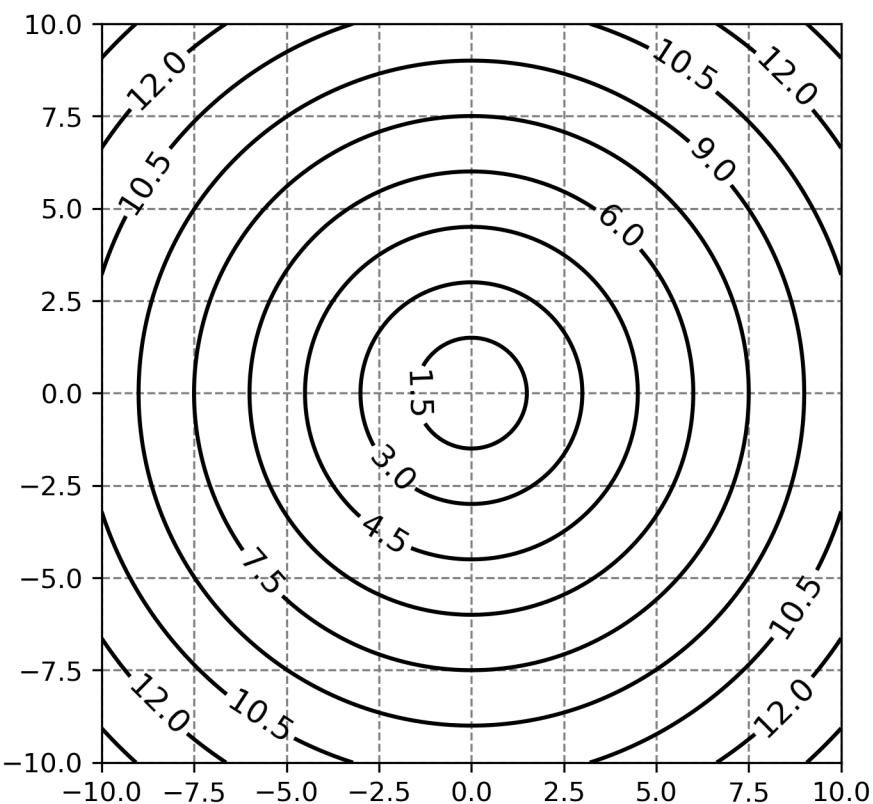


図5-2-8 等高線にラベルを付ける

全ての等高線にはラベルを付けたくない場合もあるかと思います（図5-2-9）。その場合には少し工夫が必要です（`contour_sample9.py`）。まず `cs` に対して `levels=cs.levels` を行い、等高線を描いたレベル全てを `levels` に格納します。`cs.clabel` の1番目の引数としてラベルを付ける等高線の値を渡すことが可能で、`levels[::2]` とすることで、等高線のリストを1つ飛ばしで渡すことができます（`[::2]`はスライスの記法で、開始点、終了点、ステップの順に並べる）。今度は3、6、9、12のように整数値になるので、`fmt="%d"`としてラベルの数字を整数値で表示します。

```
cs = plt.contour(X, Y, Z, colors='k', levels=levels) # 戻り値 cs
levels = cs.levels # 等高線を描いたレベルを取得
cs.clabel(levels[::2], fmt="%d", fontsize=12) # ラベルを付ける
```

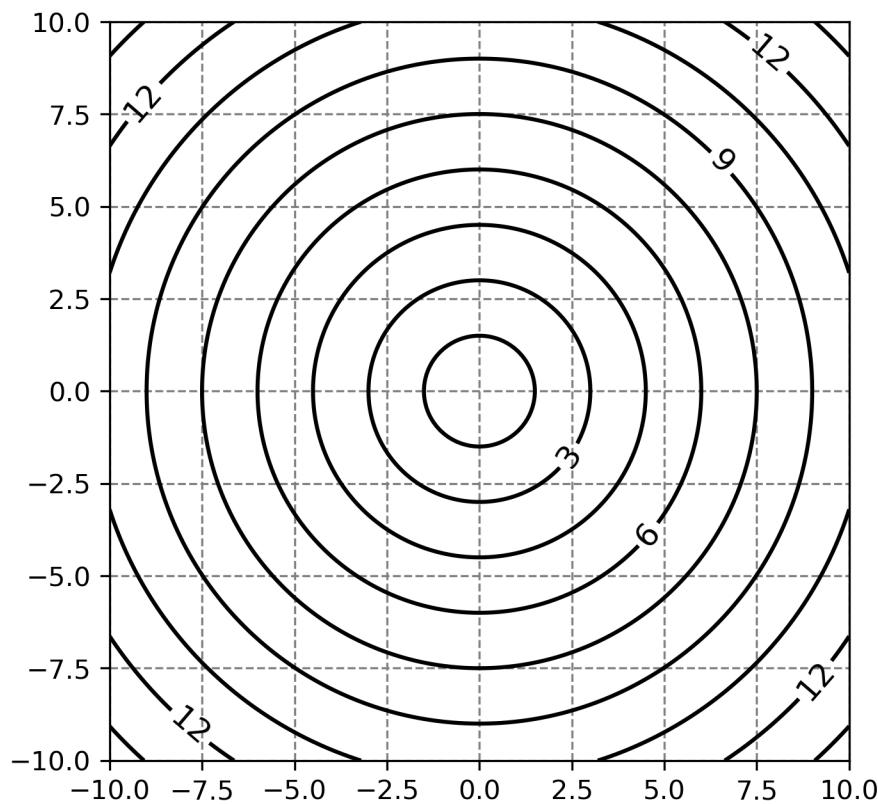


図5-2-9 ラベルを1つ飛ばしに表示

5.2.5 等高線の幅や線種を変更

ここまでプロットでは幅1の実線で等高線を描いてきました。ラインプロットの時のように線の幅や線種を指定することも可能で、それぞれ、`linewidths`と`linestyles`というオプションです。`linewidth`、`linestyle`ではないので注意が必要です。オプションに`linestyles=':'`（点線）、`linewidths=3`（幅3）を指定して作図したものが図5-2-10です。作図には`contour_sample10.py`を用いました。

```
cs = plt.contour(X, Y, Z, colors='k', levels=levels, linestyles=':', linewidths=3)
```

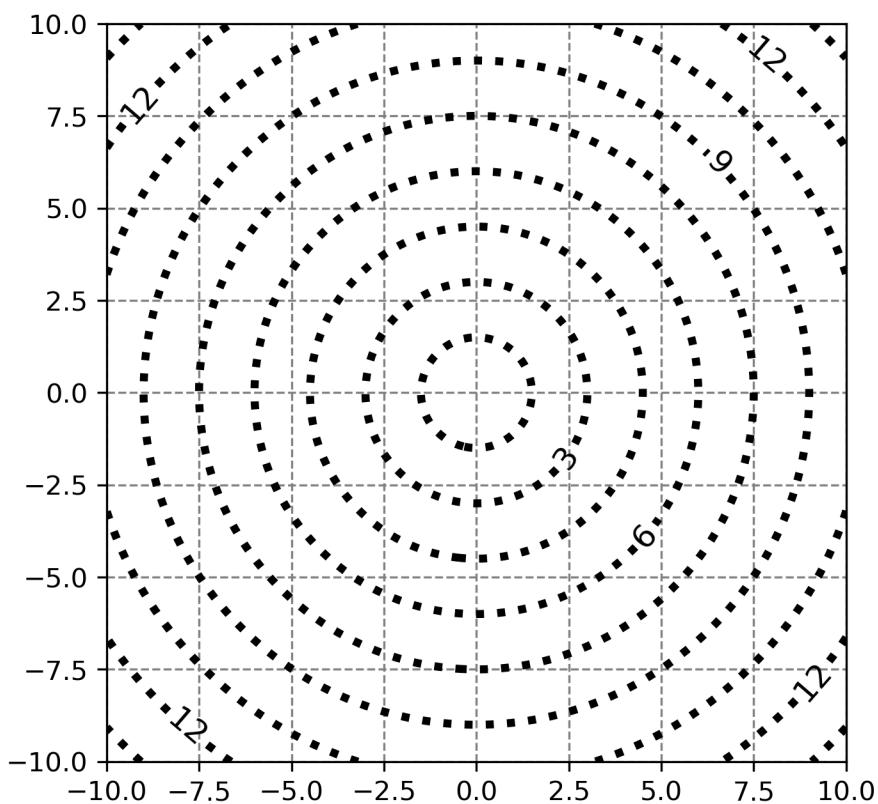


図5-2-10 等高線の幅を太くした点線を描く

他には、等高線の色をカラーマップから指定する`cmap`オプションも利用可能です (`contour_sample11.py`)。`cmap`オプションには図4-5-7のような指定が可能ですが、ここでは、`cmap='brg'`として等高線に青～赤～緑と変化する色を付けました (図5-2-11)。なお`cmap`オプションを使う場合には、`colors`

オプションを同時に使うとエラーになります。

```
cs = plt.contour(X, Y, Z, levels=levels, cmap='brg')
```

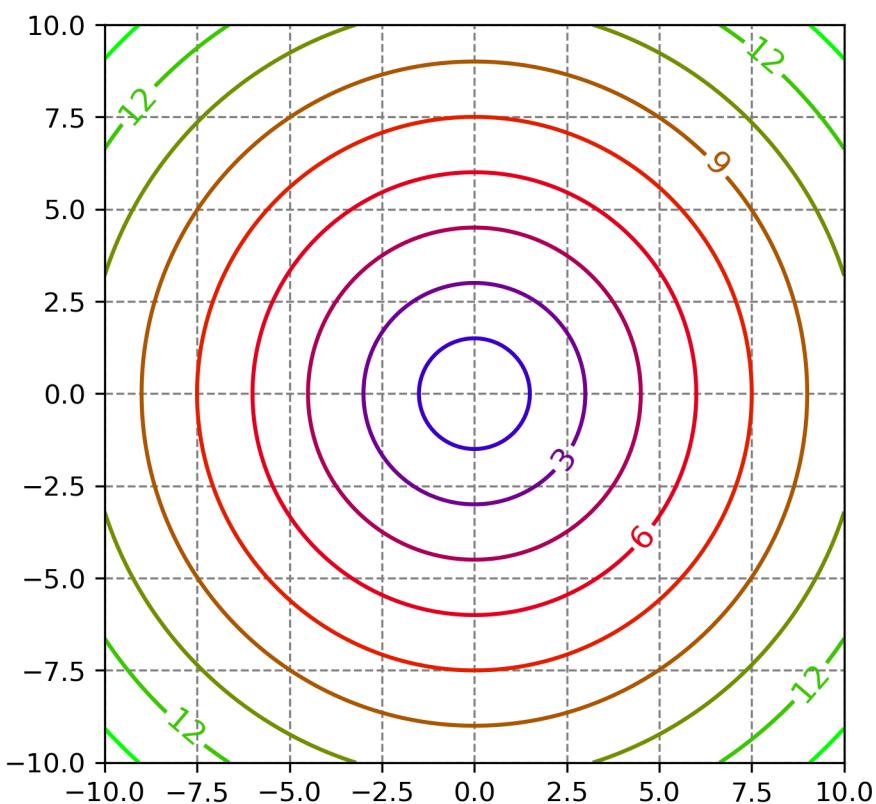


図 5 – 2 – 11 色テーブルを参照して等高線に色を付けた

なお、等高線の色、太さ、線種は、複数同時に指定することも可能です（図 5 – 2 – 12）。作図には `contour_sample12.py` を使いました。ここでは、等高線の色を黒赤交互 (`colors=['k', 'r']`)、太さは太線（幅 2）と細線（幅 1）を交互 (`linewidths=[2, 1]`)、線種は実線と点線を交互 (`linestyles=['-', ':']`) としました。

```
cs = plt.contour(X, Y, Z, colors=['k', 'r'], levels=levels, ¥  
linestyles=['-', ':'], linewidths=[2, 1])
```

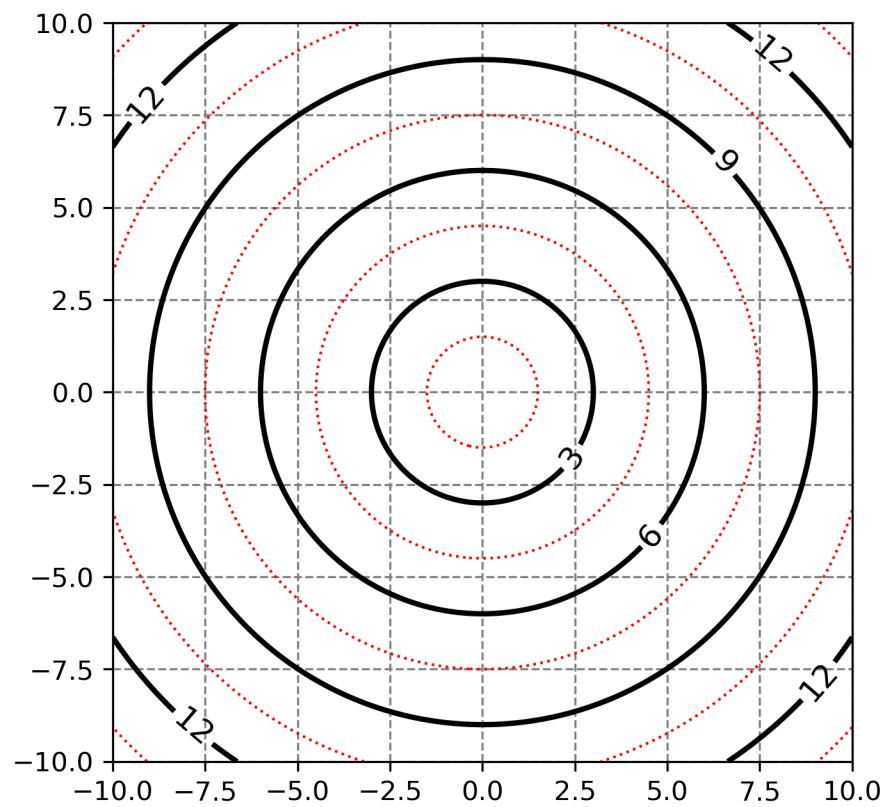


図 5－2－12 等高線の色、太さ、線種を交互に変える

5.2.6 等高線のまとめ

最後に等高線の作図で使用可能なオプションを表5-2-1にまとめておきます。等値線のラベルを指定する際に用いた書式指定子については分かりにくいので、表5-2-2に使用例を載せています。

表5-2-1 matplotlib の pyplot.contour で使用可能なオプション一覧

オプション	説明
X, Y	x軸, y軸に使用する配列、Zデータに対応した大きさの2次元配列で与える
Z (必須)	Zデータの配列
levels	等高線を描くZの値のリスト
colors	等高線の色、デフォルト値：None（自動設定される）
cmap	カラーマップ、デフォルト値：False
norm	正規化を行う場合に参照するクラス、デフォルト値：None
vmin	等高線の最小値、デフォルト値：None（データの最小値）
vmax	等高線の最大値、デフォルト値：None（データの最大値）
linewidths	線の幅、デフォルト値はrcParams["lines.linewidth"]参照（1に設定）
linestyles	等高線のスタイル、{None, 'solid', 'dashed', 'dashdot', 'dotted'}、デフォルト値：None（solidに設定）
locator	等高線を描くZの値を決めるticker.Locator、デフォルト値：ticker.MaxNLocator
alpha	不透明度、デフォルト値：1.0

使用方法：plt.contour(X, Y, Z, オプション)

表5-2-2 書式指定子の例

書式指定子	説明	例
%d	整数（切り捨て）	15
%.0f	整数（四捨五入）	16
%03d	整数（3桁、先頭を0埋め）	016
%3d	整数（3桁、先頭を空白）	16
%+d	整数（先頭に符号）	+16
%.1f	小数点以下第一位まで（四捨五入）	15.7
%.2f	小数点以下第二位まで（四捨五入）	15.65
%+.2f	小数点以下第二位まで（先頭に符号）	+15.65
%5.1f	小数点以下第一位まで（四捨五入、最低5文字）	15.7
%0e	指数表記（有効数字1桁、切り捨て）	1e+01
%1e	指数表記（有効数字2桁、切り捨て）	1.5e+01
%2e	指数表記（有効数字3桁、切り捨て）	1.56e+01

使用方法：cs.clabel(fmt="%1f")：小数点以下第一位まで

5.3 陰影を描く

等高線を描く機能と似ている機能として陰影を描く機能があります。

5.3.1 基本的な等高線

等高線の場合は plt.contour でしたが、陰影の場合は **plt.contourf** を使います。等高線を描いた時と同じデータを使い、陰影を描くプログラムが contourf_sample.py です。plt.contourf(X, Y, Z)で Z の値を参照して作図を行い、図 5 – 3 – 1 のように陰影が描かれます。ax.contourf(X, Y, Z)でも同じです。

```
plt.contourf(X, Y, Z) # 陰影を描く
```

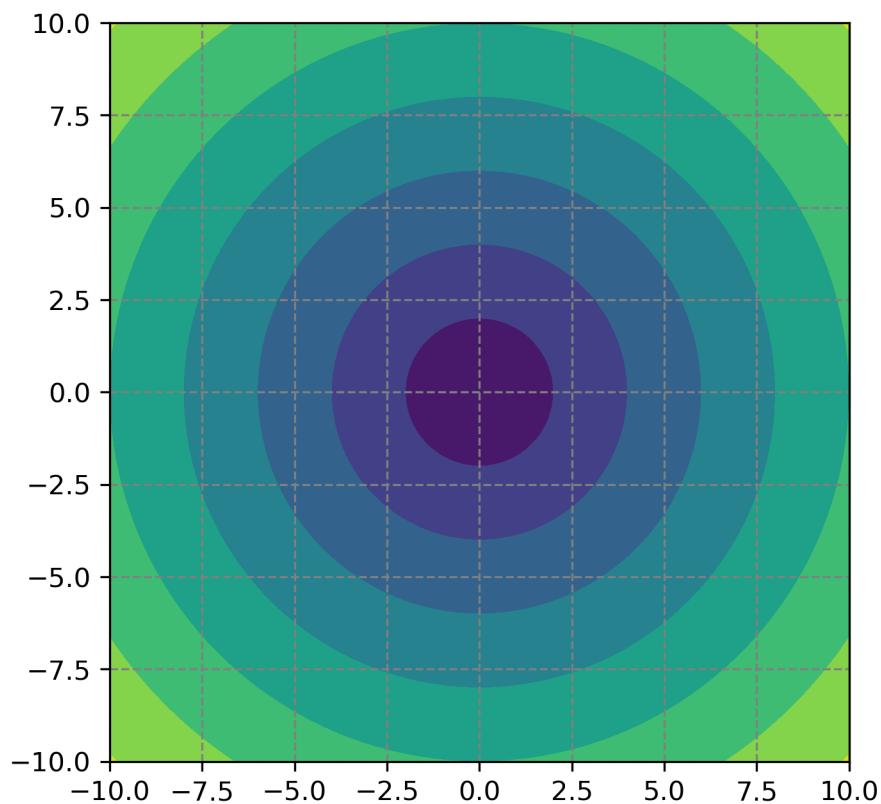


図 5 – 3 – 1 陰影を描く

5.3.2 陰影のスタイル

陰影を付ける間隔や色を変えることもできます (contourf_sample2.py)。陰影を付ける間隔を決めるのは、等高線の場合と同じで levels オプションです。陰影の色はカラーマップから指定する cmap オプションで設定します。ここでは `cmap='bwr'` (青～白～赤と変化) としました (図 5-3-2)。等高線の時とは異なり、levels の最大値は `np.ceil(Z.max())+1` で設定しています。陰影の場合には最大値を超えた領域に色が付かないためです (試しに +1 を消した状態で作図してみましょう)。

```
levels = np.arange(np.floor(Z.min()), np.ceil(Z.max())+1, 1.5)
plt.contourf(X, Y, Z, levels=levels, cmap='bwr')
```

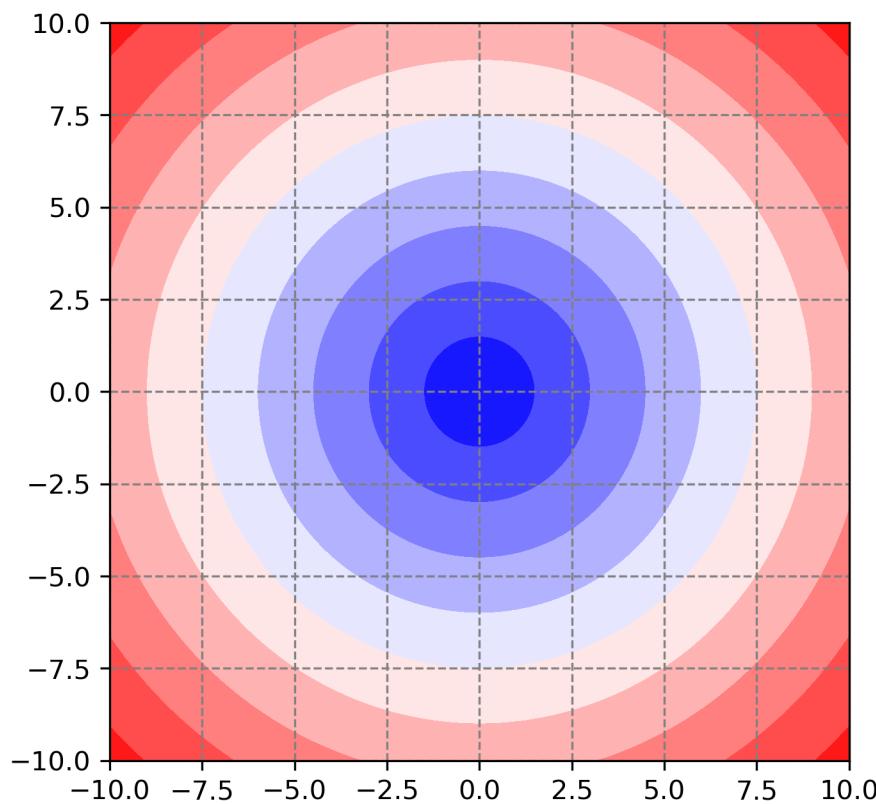


図 5-3-2 陰影のパターンを変更

陰影を付ける Z の範囲を変更することも可能ですが (contourf_sample3.py)。ここでは Z が 3 ~ 9 の範囲で色を変化させるように設定してみます (図 5-3-

3)。範囲外では最小値に設定された青色と最大値に設定された赤色が使われます。なお最小値、最大値は、それぞれ `vmin`、`vmax` オプションで指定します。

```
plt.contourf(X, Y, Z, levels=levels, cmap='bwr', vmin=3, vmax=9)
```

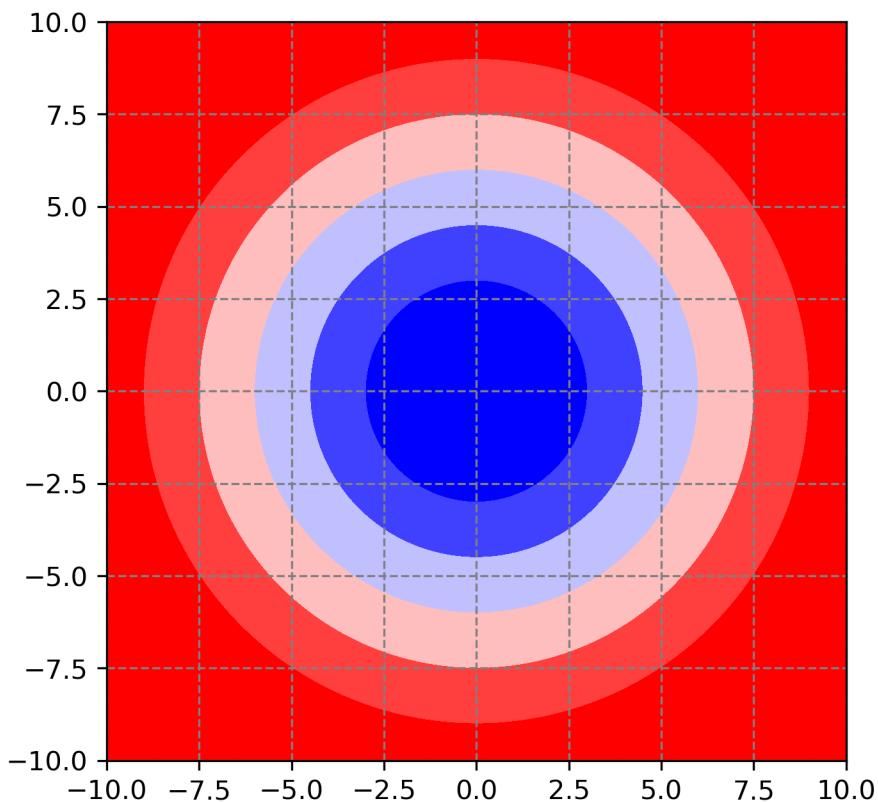


図 5－3－3 3～9 の範囲で色を変化させる

`vmin`、`vmax` オプションで指定した範囲外の色に別の色を付けたい場合もあるかと思います。その場合には、`cmap` オプションで指定する色テーブルの代わりに、[plt.get_cmap\(色テーブルの名前\)](#) の戻り値から生成されたインスタンス `cmap` を利用します。この `plt.get_cmap` では、`cmap` オプションで使う色テーブルと同じ名前で色テーブルを取得できます。

ここでは青～白～赤と変化する'bwr'に設定したので、インスタンス `cmap` にはこの設定が反映された状態です。`cmap` に対して、`cmap.set_under(色の名前)`、`cmap.set_over(色の名前)`、というメソッドを使い、それぞれ下限を下回った場合と上限を上回った場合の色を設定します。ここでは下限を下回った場合

に灰色 ('gray')、上限を上回った場合に白色 ('w') に設定しました。

```
import matplotlib as mpl
import copy
...
cmap = plt.get_cmap('bwr') # 色テーブル取得
cmap.set_under('gray') # 下限を下回った場合の色を指定
cmap.set_over('w') # 上限を超えた場合の色を指定
```

なお matplotlib バージョン 3.3 から 3.6 では、色テーブルの変更で Warning が出ていました。該当するバージョンを使っている場合、下記のような方法で Warning を回避可能です。まず、mpl.colormaps.get_cmap で色テーブルを取得し、copy.copy(mpl.cm.get_cmap(色テーブルの名前)) でコピーを保持します。なお、これらのモジュールを使用するために、matplotlib を mpl の名前でインポートし、copy もインポートしました。

```
import matplotlib as mpl
import copy
cmap = copy.copy(mpl.colormaps.get_cmap("bwr")) # 色テーブル取得
```

この時点の cmap には下限から上限までの範囲には青～白～赤の範囲で変化する色が、下限を下回った場合に灰色、上限を上回った場合に白色が設定されています。陰影を付ける部分では、このインスタンス cmap を cmap=cmap として渡すことで、設定した色で陰影描くことができます（図 5-3-4）。

```
plt.contourf(X, Y, Z, levels=levels, cmap=cmap, vmin=3, vmax=9)
```

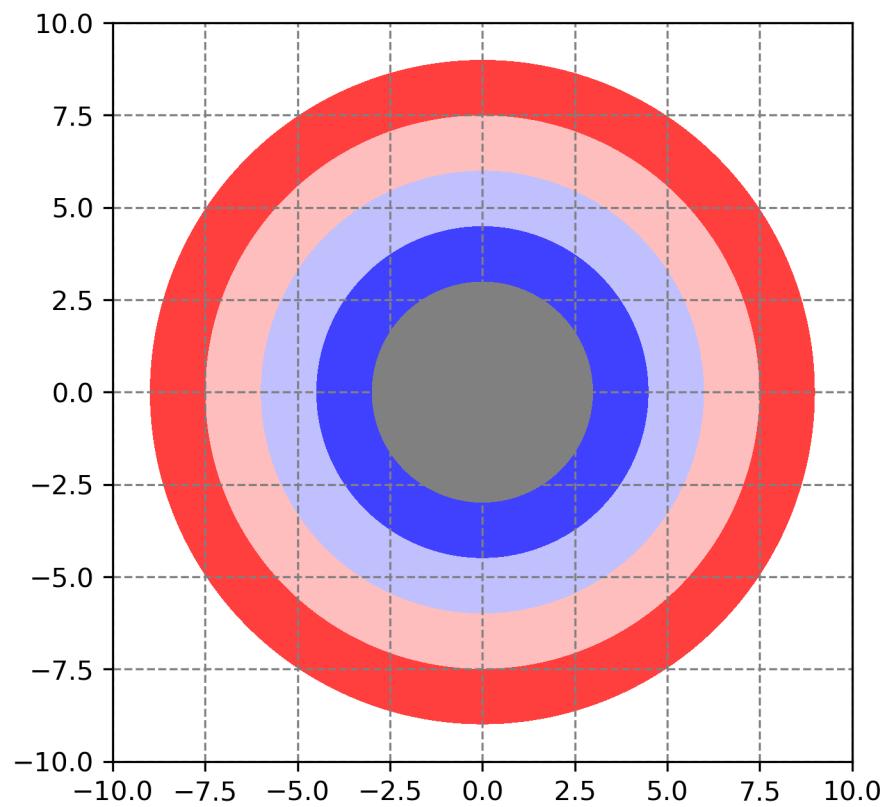


図 5－3－4 下限を下回った場合に灰色、上限を上回った場合に白で塗る

5.3.3 陰影にカラーバーを付ける

陰影だけをつけても値が分からないのでカラーバーを付けてみます (contourf_sample5.py)。カラーバーを付けるには、`plt.colorbar(オプション)` を使います。まずはオプションなしで実行してみます。カラーバーにはラベルを付けることができ、カラーバーを作成した時の戻り値インスタンス `cbar` に対して `cbar.set_label(ラベル, オプション)` の形式でラベルを付けます。ラベルには `plt.text` で利用可能な `fontsize` オプションを指定可能です。カラーバーで 3 ~ 9 まで青～赤に色が変わり、3 より小さい値は灰色、9 より大きい値は白色が表示されました (図 5 - 3 - 5)。

```
cbar = plt.colorbar() # カラーバーを付ける  
cbar.set_label('label', fontsize=14) # カラーバーにラベルを付ける
```

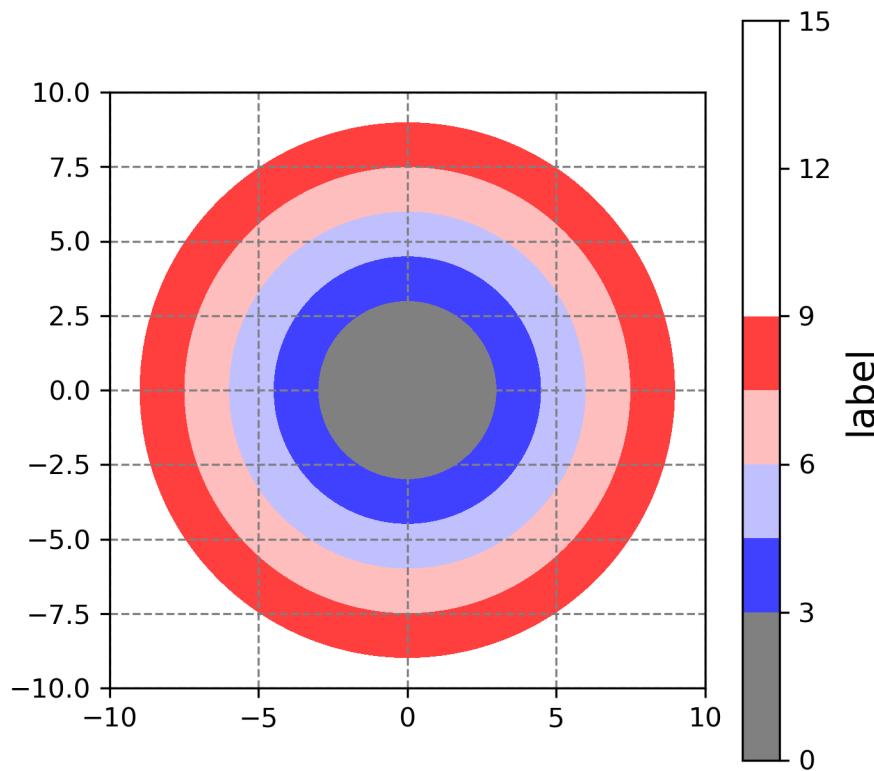


図 5 - 3 - 5 カラーバーを付ける

カラーバーが縦に長すぎるので、カラーバーのサイズを変える `shrink` オプシ

ョンを使って調整してみます (contourf_sample6.py)。shrink=0.8 を指定することでカラーバーのサイズが 0.8 倍になりました (図 5-3-6)。

```
cbar=plt.colorbar(shrink=0.8) # カラーバーのサイズを 0.8 倍に
```

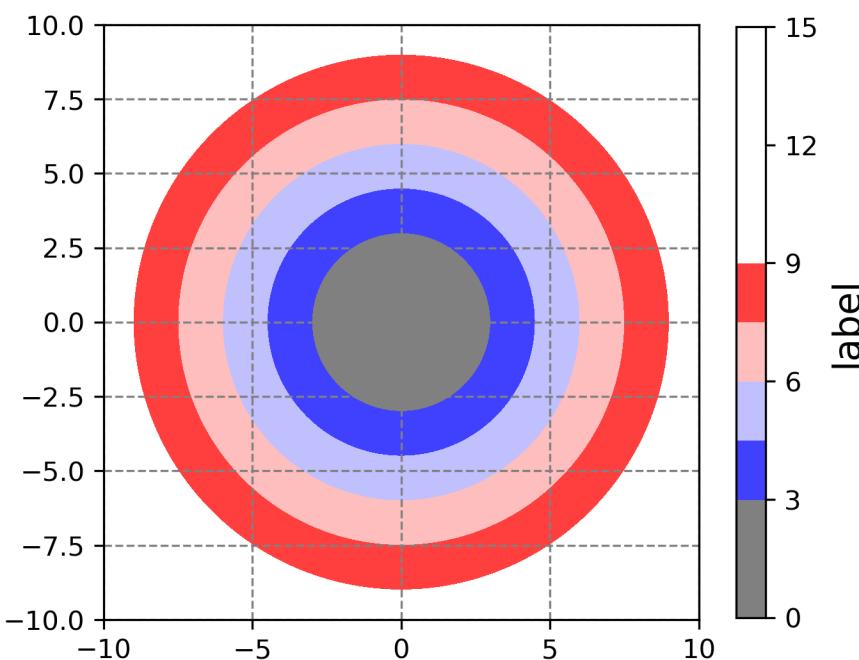


図 5-3-6 カラーバーの大きさを調整

カラーバーの両端を三角にして、カラーバーの範囲外になった色を三角の部分に表示させることも可能です (contourf_sample7.py)。ここでは 3~12 まで値を変化させて、その領域をカラーバーの中に、外れた値を両端の三角の部分に表示することを考えます。値の範囲と間隔は、データの最大値、最小値からの計算ではなく、`np.arange(3, 12.1, 1.5)`で指定しておきます。色を変化させる範囲は、`vmin=3, vmax=12` とします。オプションに `extend='both'` を付けることで、図 5-3-7 のようにカラーバーの両端が変化します。

```
levels = np.arange(3, 12.1, 1.5)
plt.contourf(X, Y, Z, levels=levels, cmap=cmap, \
vmin=3, vmax=12, extend='both')
```

なお、片側だけに三角の部分を表示するには、最小値側に付けたい場合 extend='min'、最大値側に付けたい場合 extend='max'とします。

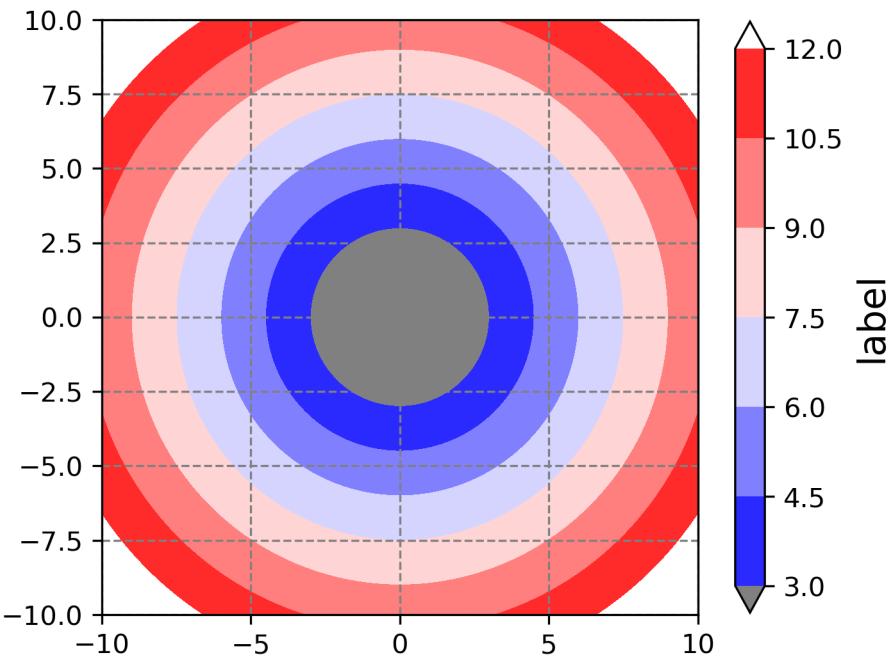


図 5－3－7 カラーバーの両端に三角の表示を付ける

カラーバーには他にも表 5－3－1 のようなオプションがあります。

表 5－3－1 matplotlib の pyplot.colorbar で使用可能なオプション一覧

オプション	説明
orientation	カラーバーの向き、['vertical' 'horizontal']
fraction	元の軸に対する比率、デフォルト値：0.15
pad	カラーバーとグラフとの隙間、デフォルト値：0.05 (vertical) 、 0.15 (horizontal)
shrink	カラーバーのサイズ (倍率) 、デフォルト値：1.0
aspect	短軸に対する長軸の比、デフォルト値：20
anchor	カラーバーの位置 (colorbar axes) 、デフォルト値：verticalで(0.0, 0.5)、horizontalで(0.5, 1.0)
panchor	カラーバーの位置、(colorbar parent axes) 、デフォルト値：verticalで(1.0, 0.5)、horizontalで(0.5, 0.0)
format	カラーバーの目盛り線ラベルの書式 (整数 : format="%0.f"、小数点以下第一位 : format=".1f")

使用方法 : plt.colorbar()、plt.colorbar(オプション)

5.3.4 ハッチを付ける

plt.contourf のオプションに **hatches** というものがありますが、どのようになるか見ていきましょう。hatch ではなく hatches なので注意してください。図 5-3-8 は先ほどの図にハッチを付けたものです。カラーバーにもハッチの情報が渡されるので、自動で変更されています。hatches オプションには表 4-2-3 のハッチのパターンを与えます。次のように `hatches=['//', '..', 'xx']` のように 3 種類だけ与えた場合には、3 種類のパターンの繰り返しとして値が小さい方から順に描かれます。作図には `contourf_sample8.py` を使いました。

```
plt.contourf(X, Y, Z, levels=levels, cmap=cmap, vmin=3, vmax=9, ¥  
             hatches=['//', '..', 'xx'])
```

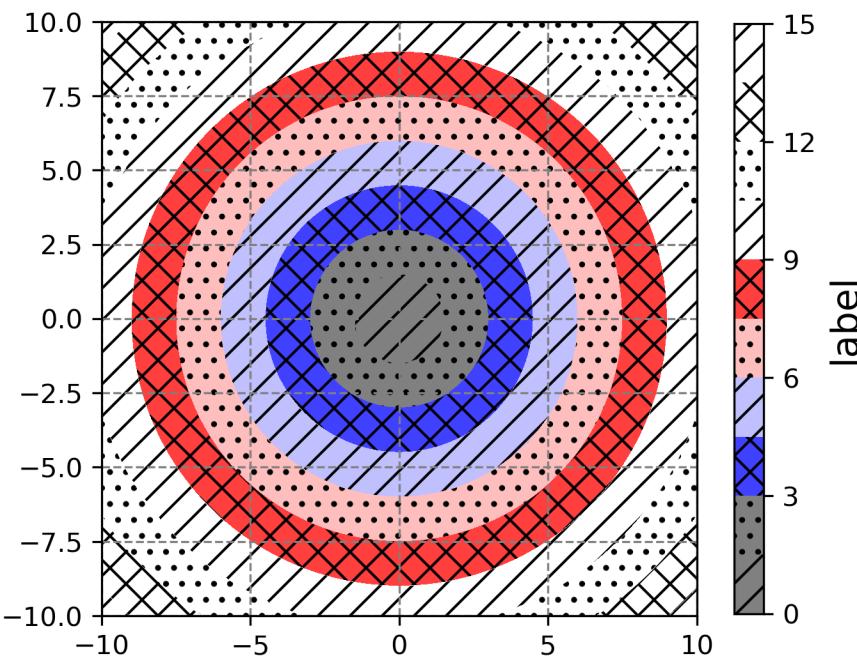


図 5-3-8 ハッチを付ける

5.3.5 ログスケールに変える

`plt.contourf` のオプションに `norm` というものがあり、正規化の行い方を指定することができます。ここではログスケールにしてみます（図5-3-9）。ログスケールに変えるために `matplotlib.colors` から `LogNorm` を import し、`norm=LogNorm()` のように与えます。作図には `contourf_sample9.py` を使いました。

```
from matplotlib.colors import LogNorm  
...  
plt.contourf(X, Y, Z, cmap=cmap, norm=LogNorm())
```

バージョン 3.2～3.4 の `matplotlib` では、図5-3-9（上）のように凡例のラベルに小目盛りが付き、底が 10 で表示されるようになりました。バージョン 3.5 以降では、図5-3-9（下）のように小数点以下の桁数が最も長いものに合わせられます。バージョン 3.2～3.4 で同様の表記としたい場合には、次のようなコードを加えます。

```
from matplotlib.ticker import ScalarFormatter  
cbar.ax.ticklabel_format(style="sci", axis="y", scilimits=(0, 0))  
cbar.ax.yaxis.set_major_formatter(ScalarFormatter(useMathText=True))
```

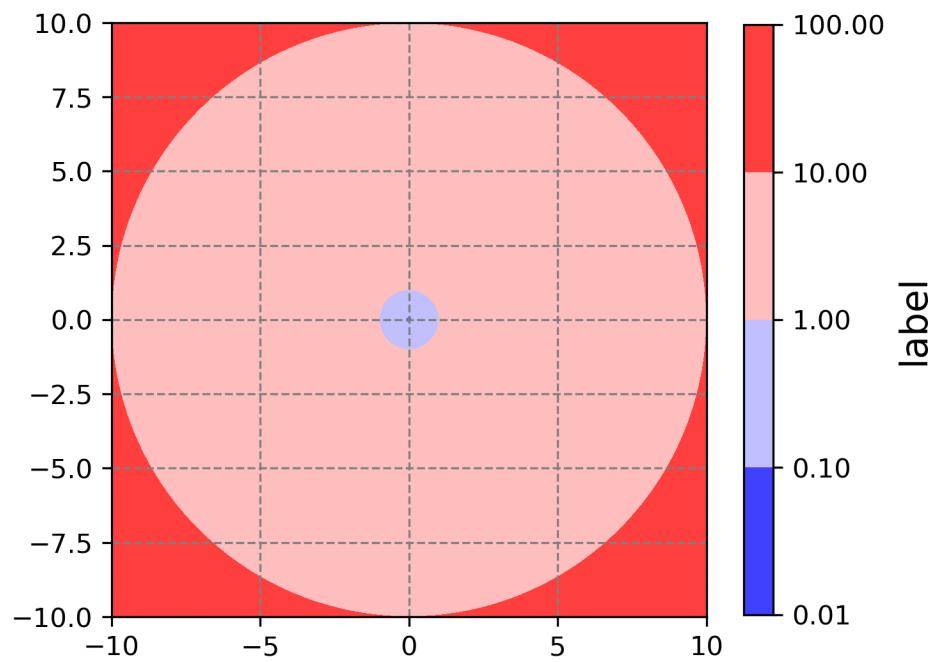
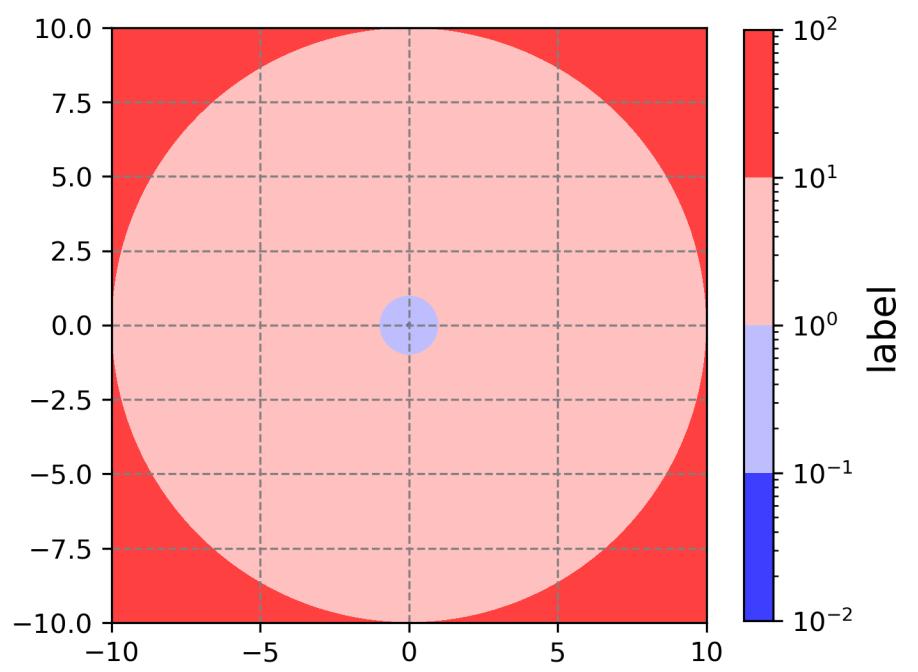


図 5－3－9 LogNorm を使いログスケールに変える（上：3.2～3.4、下：3.5 以降）

バージョン 3.2 より古いバージョンでは、図 5-3-10 のように小目盛りは付きません。3.2~3.4 で凡例に小目盛りを付けないようにしたい場合には、contourf_sample10.py のように凡例の小目盛りの設定を変更します。カラーバーを付けた際の戻り値 cbar に対し、y 軸の小目盛り線の設定は、

cbar.ax.yaxis.set_minor_locator(設定したい内容)

のように行います。ここでは小目盛り線を消したいので、ticker.NullLocator()を使いました。図の x 軸や y 軸であれば ax.xaxis や ax.yaxis に対して設定しますが、cbar に対しても同様の設定を行うことができると思っておいてください。

```
cbar = plt.colorbar(shrink=0.8) # カラーバーを付ける  
cbar.set_label('label', fontsize=14) # カラーバーにラベルを付ける  
cbar.ax.yaxis.set_minor_locator(ticker.NullLocator()) # 小目盛り線を消す
```

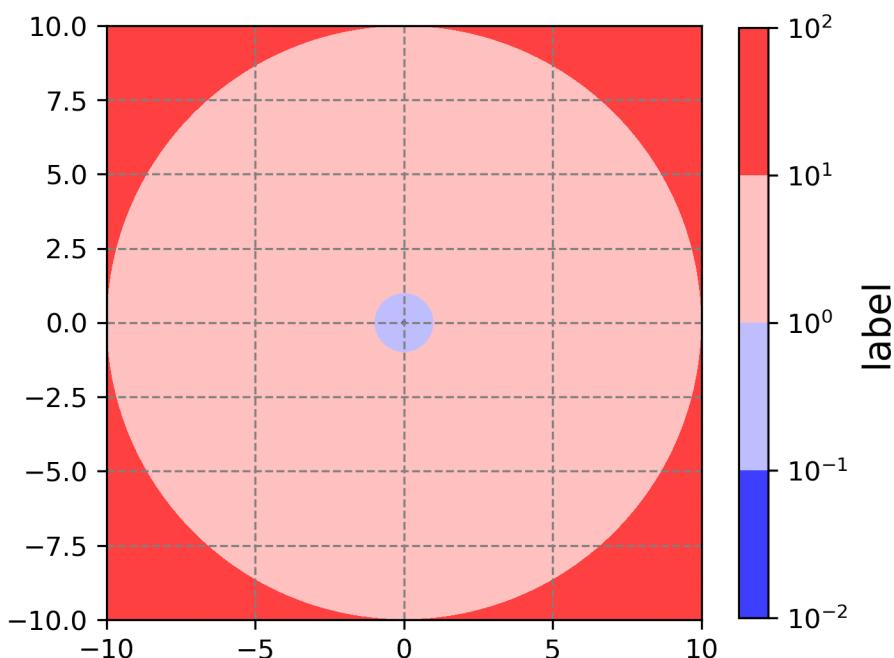


図 5-3-10 凡例の小目盛り線を消す

5.3.6 陰影のまとめ

最後に陰影の作図で利用可能なオプションを表5-3-2にまとめました。

表5-3-2 matplotlib の pyplot.contourf で使用可能なオプション一覧

オプション	説明
X, Y	x軸, y軸に使用する配列、Zデータに対応した大きさの2次元配列で与える
Z (必須)	zデータの配列
levels	陰影を描くZの値のリスト
colors	陰影の色、デフォルト値：None
cmap	カラーマップ、デフォルト値：False
norm	正規化を行う場合に参照するクラス、デフォルト値：None
vmin	陰影の最小値、デフォルト値：None (データの最小値)
vmax	陰影の最大値、デフォルト値：None (データの最大値)
hatches	ハッチのスタイル、デフォルト値：None (ハッチなし)
locator	陰影を描くZの値を決めるtickerLocator、デフォルト値：ticker.MaxNLocator
alpha	不透明度、デフォルト値：1.0

使用方法：plt.contourf(X, Y, Z, オプション)

plt.contourf でも colors を利用でき、塗り潰したい色順のリストを与えると、その順番に塗り潰すことができます。なお plt.contourf に linewidths, linestyles オプションを与えた場合はエラーにはなりませんが無視されます。

5.4 箱ひげ図、バイオリンプロット

matplotlib にはデータの分布を表すためのプロットも含まれており、ここでは、その内の箱ひげ図、バイオリンプロットを紹介しておきます。

5.4.1 基本的な箱ひげ図

中央に長方形の箱、その上下にヒゲが付いた箱ひげ図を見たことがあると思います。箱ひげ図を描くには `plt.boxplot` を使います。`ax.boxplot` でも同じです。まずは、散布図の作図に用いた AO index のデータを使い、箱ひげ図を作成してみます (`boxplot_sample.py`)。

```
plt.boxplot(data_x) # 箱ひげ図
```

図 5-4-1 は作成された箱ひげ図で、中央の箱が第一四分位数（だいいちしぶんいすう、25 パーセンタイル）から第三四分位数（75 パーセンタイル）までを、ヒゲの下端が最小値、ヒゲの上端が最大値を表します。箱に入っている橙色の横線は、第二四分位数（中央値）です。最小値よりも下に 4 つの丸がありますが、これらは外れ値を表していて、デフォルトでは、最小値、最大値側それぞれに箱の 1.5 倍よりも外れた値があった場合、外れ値として処理されます（図 5-4-2）。

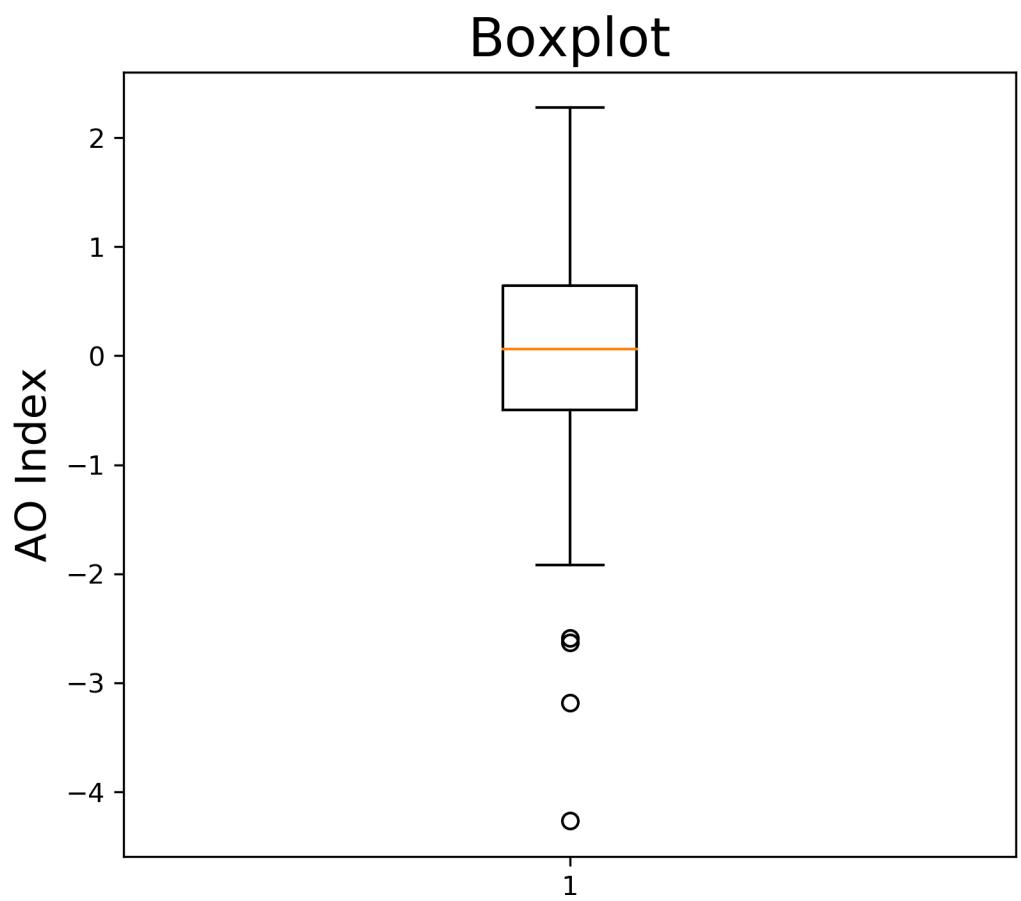


図 5－4－1 2010～2018 年の AO index を使い作図した箱ひげ図。縦軸の 1 は 1 標準偏差に相当

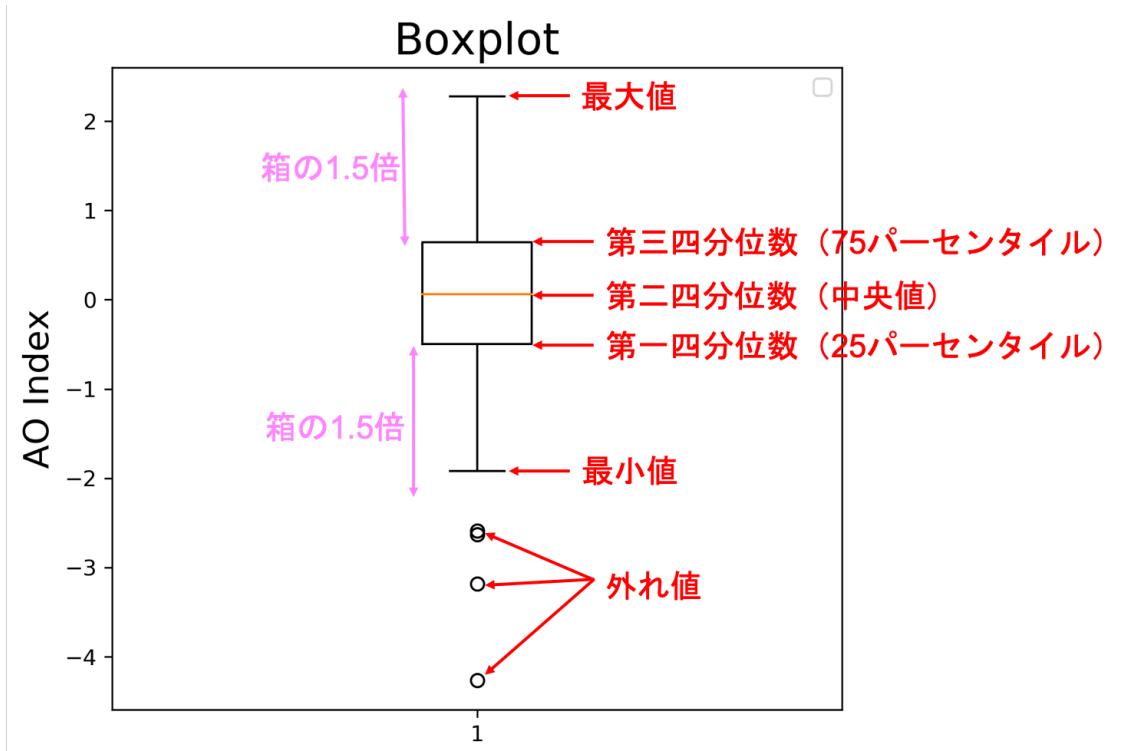


図 5－4－2 箱ひげ図の説明

本来なら外れ値のプロットは測定ミスなどの異常値を除外する目的で使うものなので、AO index のように -3σ や -4σ を下回る値が含まれていてもおかしくないデータでは、外れ値として扱うべきではありません。boxplot にはヒゲをどこまで描くか指定する whis オプションがあり、デフォルトでは whis=1.5 (箱の 1.5 倍まで) です。この値を whis=100 のように非常に大きな値にしておくことで、実際の最小値と最大値までヒゲが描かれます（図 5－4－3）。作図に用いたプログラムは、boxplot_sample2.py です。x 軸の目盛り線ラベルが 1 になっているので、目盛り線ラベルを設定する ax.set_xticklabels を使い、AO index に変えています。また、ax.grid でグリッド線を付けました。

なお、whis=[5, 95]のように範囲をリストで渡すと、5～95%の範囲でヒゲを描くことが可能です。

```
ax.boxplot(data_x, whis=100) # 箱ひげ図（外れ値なし）
ax.set_xticklabels(["AO index"]) # x 軸の目盛り線ラベル
ax.grid(color='gray', ls=':') # グリッド線を描く
```

さらに、y 軸の目盛り線の調整や、x、y 軸の目盛り線の向きの調整を行って図の体裁を整えています。

```
plt.rcParams['xtick.direction'] = 'in' # x 軸目盛線を内側
plt.rcParams['xtick.major.width'] = 1.2 # x 軸大目盛線の長さ
plt.rcParams['ytick.direction'] = 'in' # y 軸目盛線を内側
plt.rcParams['ytick.major.width'] = 1.2 # y 軸大目盛線の長さ
...
ax.yaxis.set_major_locator(ticker.AutoLocator()) # y 軸大目盛り
ax.yaxis.set_minor_locator(ticker.AutoMinorLocator()) # y 軸小目盛り
```

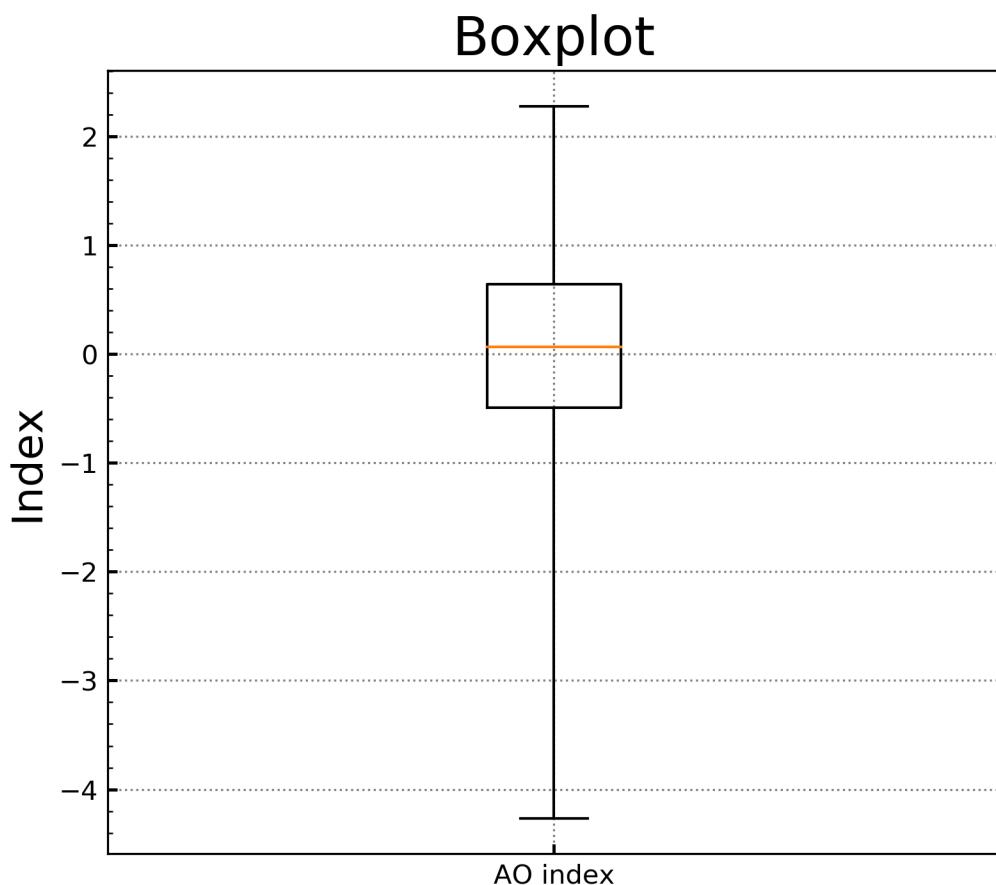


図 5－4－3 外れ値を描かない

5.4.2 複数の箱ひげ図を並べる

複数の箱ひげ図を並べるには、どうしたら良いでしょうか。先ほどの AO index の箱ひげ図に加えて、NAO index、PNA index の箱ひげ図を横に並べて描くプログラムが boxplot_sample3.py です。ax.boxplot の 1 番目の引数には 1 つのデータの配列だけではなく、複数のデータをタプルにして渡すことも可能です。ここでは、1 番目の引数に 3 つのデータをタプルにした(data_x1, data_x2, data_x3)を渡します。目盛り線ラベルを設定する set_xticklabels には、データに対応させて AO index、NAO index、PNA index の 3 種類のラベル与えます。図 5-4-4 のように 3 つの箱ひげ図が横並びに描かれました。箱の中に緑三角が描かれているのに気が付いたでしょうか。緑三角は算術平均値を表していて、showmeans=True のオプションを与えたことで描かれるようになりました (showmeans はデフォルトでは False)。

```
ax.boxplot((data_x1, data_x2, data_x3), whis=100, showmeans=True)
ax.set_xticklabels(["AO index", "NAO index", "PNA index"]) # 目盛り線ラベル
```

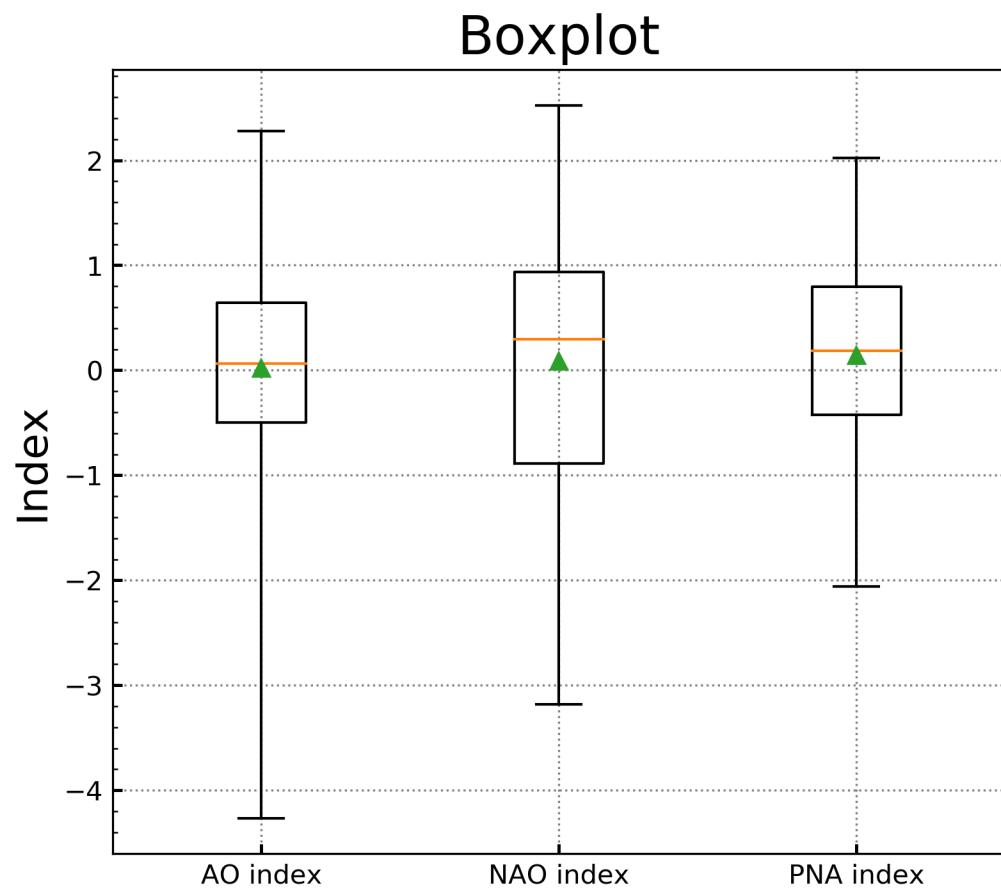


図 5－4－4 3つの箱ひげ図を並べる（2010～2018 年の AO index、NAO index、PNA index）

5.4.3 箱ひげ図のスタイル

箱ひげ図には `patch_artist` というオプションがあり、デフォルトでは `False` です (`False` では `Line2D artist` を使い、`True` では `Patch artists` を使います)。 `patch_artist=True` では、箱ひげ図の見た目が図 5-4-5 のように変わります。大きな違いは箱を塗り潰す部分で、`patch_artist=False` では塗り潰さないスタイル、`patch_artist=True` では青色に塗り潰すスタイルです。作図に用いたプログラムは `boxplot_sample4.py` です。

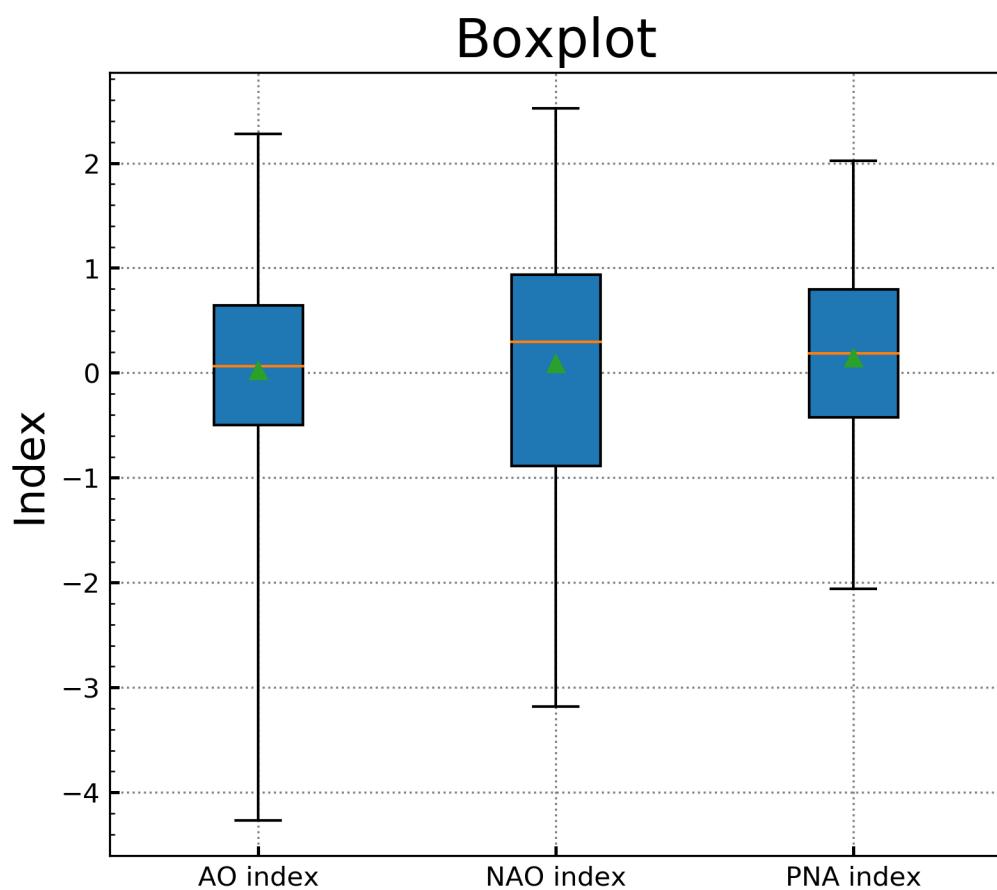


図 5-4-5 `patch_artist=True` で図の見た目を変える

ヒゲの端の `cap`、箱、ヒゲ、外れ値、中央値、平均値について、個別にスタイルを変更することができます。図 5-4-6 の左側は各部分の名称を表していて、ヒゲの端の `cap` 部分のスタイルは `capprops`、箱の部分のスタイルは `boxprops`、ヒゲの部分のスタイルは `whiskerprops`、外れ値の部分のスタイルは `flierprops`、

中央値の部分のスタイルは medianprops、平均値の部分のスタイルは meanprops というオプションにそれぞれ対応します。オプションの値は、変更する色や線の名前などを表す key と value を組み合わせた辞書で与えます。

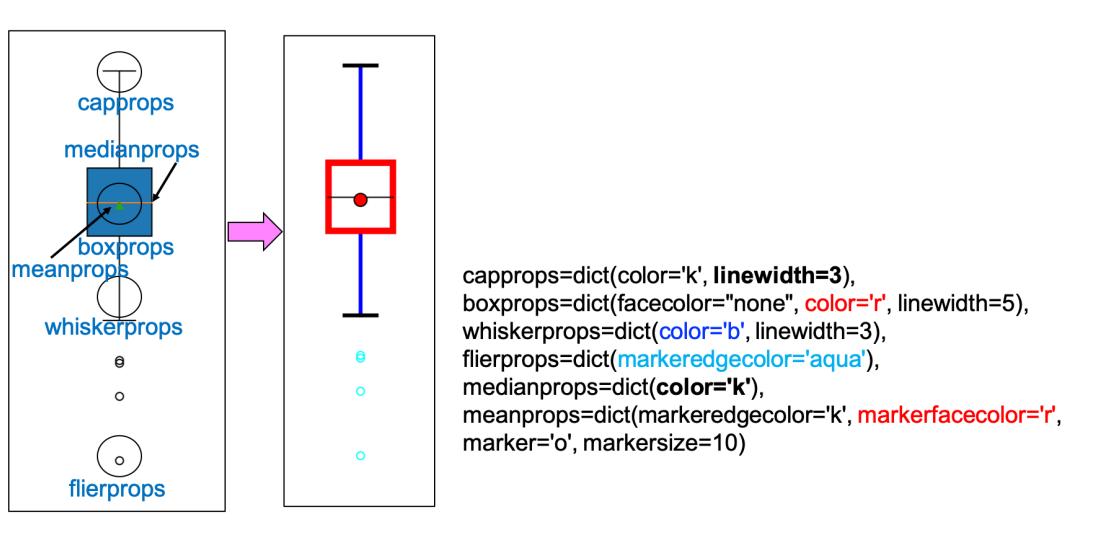


図 5－4－6 箱ひげ図の各部分と設定の変更例（必ずしも見た目良く変更してはいません）

ヒゲの端の cap 部分のスタイル (capprops) のデフォルト値は、黒線 (color='k') で幅は 1 (linewidth=1) です。線の幅を太くするには、

```
capprops=dict(color='k', linewidth=3)
```

のように、key を linewidth、value を 3 としたものを辞書で与えます。図 5－4－6 の右側は変更後で、色はデフォルトと同じなので黒のまま変わりませんが、線の幅が 3 になりました。

箱の部分のスタイル (boxprops) も変更してみます。

```
boxprops=dict(facecolor="none", color='r', linewidth=5)
```

のように与えると、箱の枠線の色が赤、幅が 5 になりました。facecolor は箱に塗る色を表していて、塗り潰さない場合には facecolor="none" とします。紛らわしいですが facecolor=None (色を指定しないのでデフォルトのまま) ではありません。

ヒゲの部分のスタイル (whiskerprops) を変更し、ヒゲの色を青、幅を 3 にしてみます。

```
whiskerprops=dict(color='b', linewidth=3)
```

外れ値の部分のスタイル (flierprops) も変更し、外れ値を表す丸を水色にします。マーカーの端の色は、次のように markeredgecolor で指定します。

```
flierprops=dict(markeredgecolor='aqua')
```

他にもマーカーの大きさ (markersize) やマーカーの種類 (marker) も変更可能です。

中央値の部分のスタイル (medianprops) を変え、中央値を表す横線を黒色にしてみます。

```
medianprops=dict(color='k')
```

平均値の部分のスタイル (meanprops) も変更し、マーカーの種類を丸に、大きさを 10 に変えます。また、マーカーの端の色 (markeredgecolor) を黒色に変え、マーカーを塗り潰す色 (markerfacecolor) を赤色に変えています。

```
meanprops=dict(markeredgecolor='k', markerfacecolor='r',  
marker='o', markersize=10)
```

それでは、実際に図 5-4-5 の箱ひげ図のスタイルを変更してみます。作図に用いたプログラムは、boxplot_sample5.py です。図 5-4-7 では、箱を塗り潰さない、全体の色は黒、中央値は黒線、平均値は黒色のプラス記号で描きました。線の幅は変更すると見栄えが悪いので、デフォルトの 1 のままにしました。平均値のプラス記号のマーカーは marker='+' で表示でき、塗り潰しが必要なマーカーなので、マーカーの端の色 (markeredgecolor) だけを設定します。

```
ax.boxplot((data_x1, data_x2, data_x3), whis=100,  
          patch_artist=True, showmeans=True,  
          capprops=dict(color='k'),  
          boxprops=dict(facecolor="none", color='k'),  
          whiskerprops=dict(color='k'),  
          medianprops=dict(color='k'),  
          meanprops=dict(marker='+', markeredgecolor='k'))
```

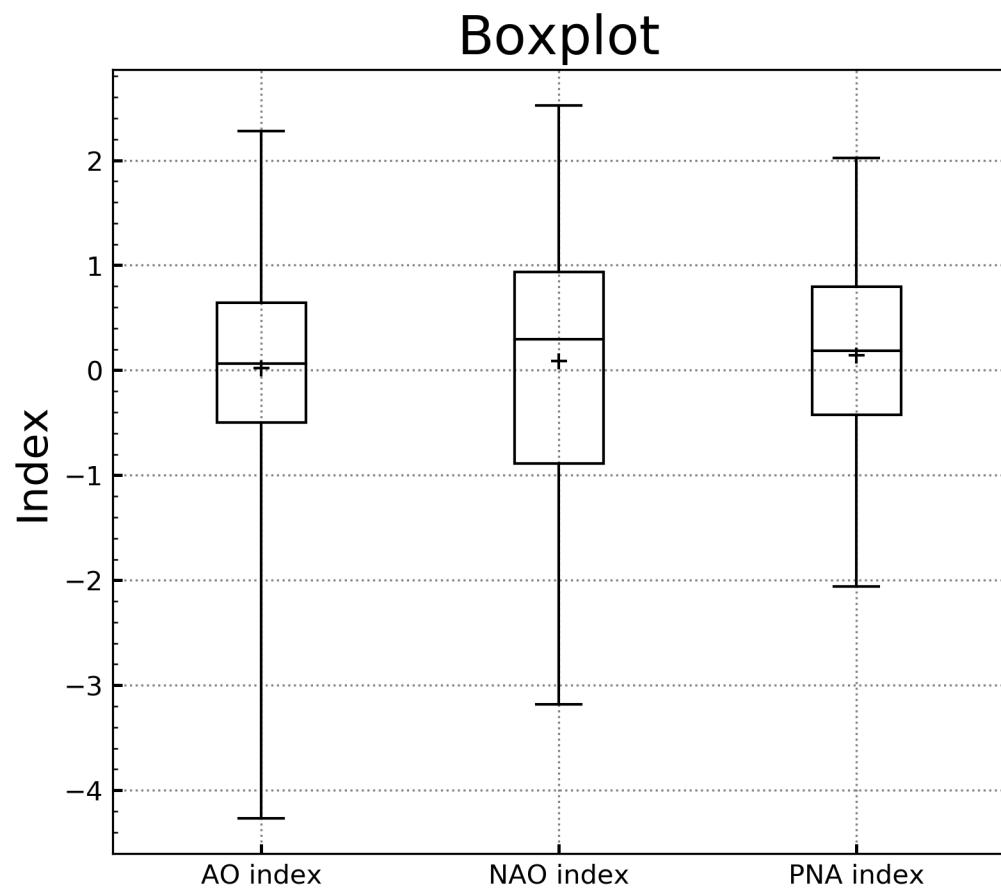


図 5－4－7 箱ひげ図のスタイルを変更

5.4.4 箱ひげ図のまとめ

最後に、箱ひげ図の作成で利用可能なオプションを表5-4-1にまとめておきます。

表5-4-1 matplotlib の pyplot.boxplot で使用可能なオプション一覧

オプション	説明
x	入力データ
notch	notchで表示、{True False}、デフォルト値：False（箱ひげ図）
vert	鉛直にするかどうか、{True False}、デフォルト値：True
whis	ヒゲをどこまで描くか、デフォルト値：1.5（箱の1.5倍まで）
bootstrap	notchにした場合で、中央値周辺の95%の信頼区間をbootstrapで決めるかどうか None : bootstrapを用いない、推奨値：1000～10000
usermedians	中央値を与える場合の配列（xと同じサイズ）、デフォルト値：None（自動計算）
conf_intervals	信頼区間を与える場合の配列、デフォルト値：None（自動計算）
positions	箱の位置を配列
widths	箱の幅を配列、デフォルト値：0.5
patch_artist	Patch artistsで描く、{True False}、デフォルト値：False（Line2D artistで描く）
labels	それぞれのデータセットのラベル（xと同じサイズ）、デフォルト値：None
autorange	25%と75%のパーセンタイルを均等、{True False}、デフォルト値：False
meanline	算術平均値を線で表示、{True False}、デフォルト値：False（点で表示）
zorder	boxplotを描く順序
capprops	ヒゲの端のcapのスタイル、辞書で与える
boxprops	箱のスタイル、辞書で与える
whiskerprops	ヒゲのスタイル、辞書で与える
flierprops	外れ値の部分のスタイル、辞書で与える
medianprops	中央値のスタイル、辞書で与える
meanprops	算術平均値のスタイル、辞書で与える
showcaps	ヒゲの端のcapを描く、{True False}、デフォルト値：True
showbox	中央の箱を描く、{True False}、デフォルト値：True
showfliers	capの外側のoutlierの部分を描く、{True False}、デフォルト値：True
showmeans	算術平均値を描く、{True False}、デフォルト値：False

使用方法：plt.boxplot(x)

5.4.5 基本的なバイオリンプロット

箱ひげ図と同じように最大値、最小値の範囲の情報を持ちながら、確率分布関数全体の形を同時に描くものがバイオリンプロットです。バイオリンプロットを描くには `plt.violinplot` を使います。`ax.violinplot` でも同じです。先ほど箱ひげ図を描くのに用いた 2010～2018 年の AO index を使ってバイオリンプロットを描いてみます (`violinplot_sample.py`)。`plt.violinplot` にオプションなどを与えて描いたものが図 5-4-8 です。最大値と最小値の範囲をヒゲで描き確率分布関数を重ねています。確率分布関数はカーネル密度推定により自動的に計算されます。

```
plt.violinplot(data_x) # バイオリンプロット
```

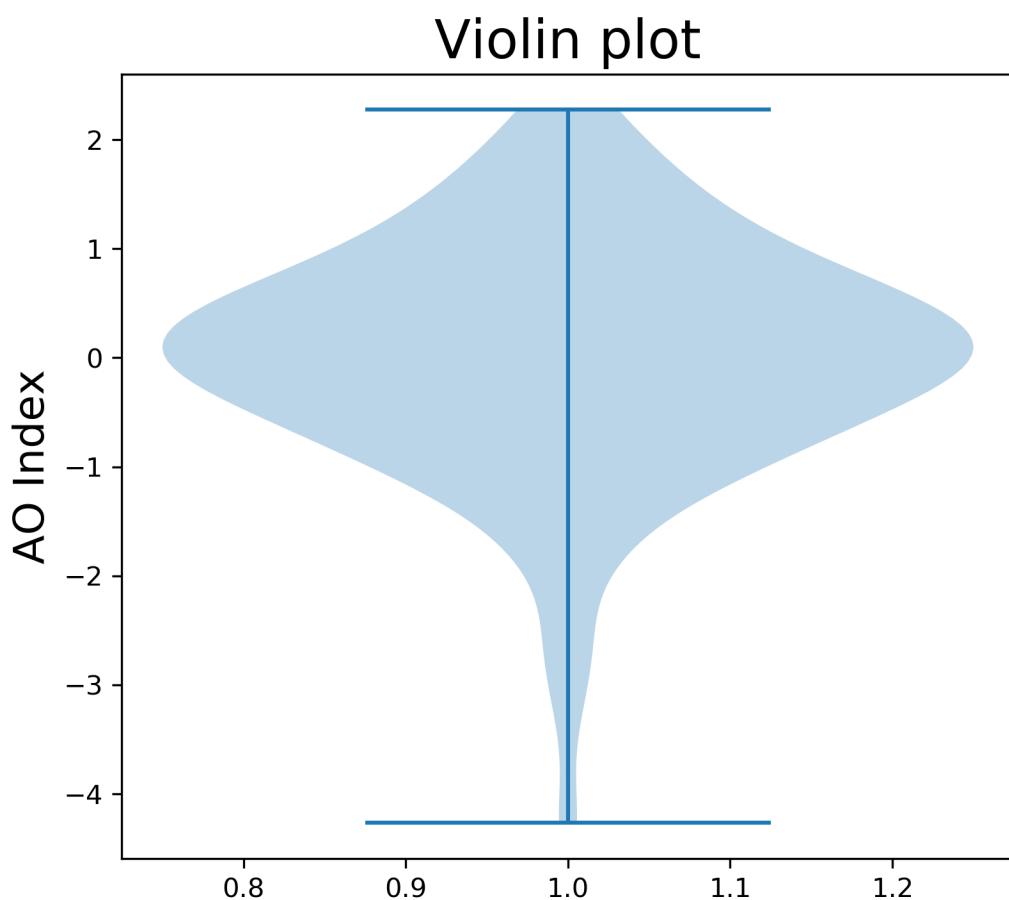


図 5-4-8 2010～2018 年の AO index を使って作成したバイオリンプロット

5.4.6 複数のバイオリンプロットを並べる

複数のバイオリンプロットを並べる場合は、先ほどの箱ひげ図の場合と同じで、1番目の引数に単独のデータ配列 data_x ではなく、3つのデータ配列をタプルにした(data_x1, data_x2, data_x3)を渡します (violinplot_sample2.py)。AO index に加えて、NAO index、PNA index のバイオリンプロットが並びました (図 5-4-9)。

```
ax.violinplot((data_x1, data_x2, data_x3)) # バイオリンプロット  
ax.set_xticks([1, 2, 3]) # 目盛り線  
ax.set_xticklabels(["AO index", "NAO index", "PNA index"]) # 目盛り線ラベル
```

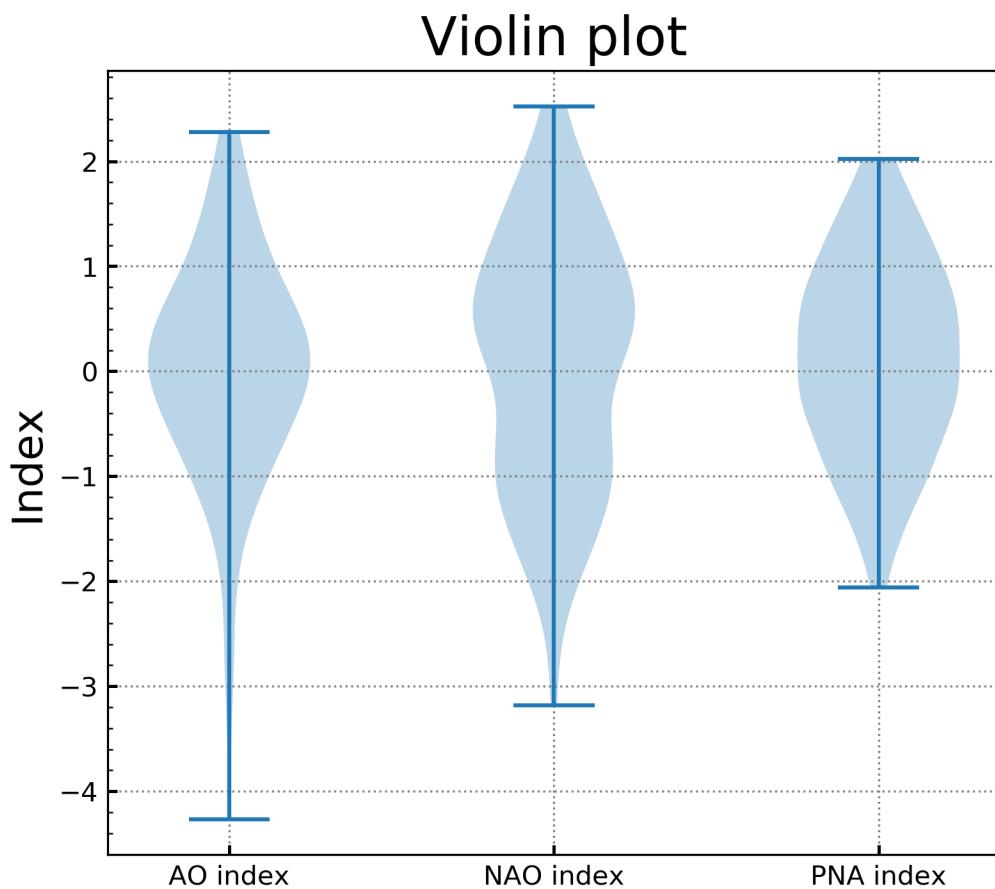


図 5-4-9 3つのバイオリンプロットを並べる

バイオリンプロットでは、算術平均値を表す横線を描くかどうか

(showmeans、デフォルトでは False)、最小値と最大値の範囲を表すヒゲを描くかどうか (showextrema、デフォルトでは True)、中央値を表す横線を描くかどうか (showmedians、デフォルトでは False) をオプションとして指定することができます。またバイオリンプロットの横幅を widths で指定できます (デフォルト値は widths=0.5)。width ではなく widths のことで注意が必要です。

バイオリンプロットの幅を倍にして (widths=1.0)、横線で中央値を描く (showmedians=True) ようにしてみます (図 5-4-10)。

```
ax.violinplot((data_x1, data_x2, data_x3),  
               widths=1.0, showmeans=False, showextrema=True, showmedians=True)
```

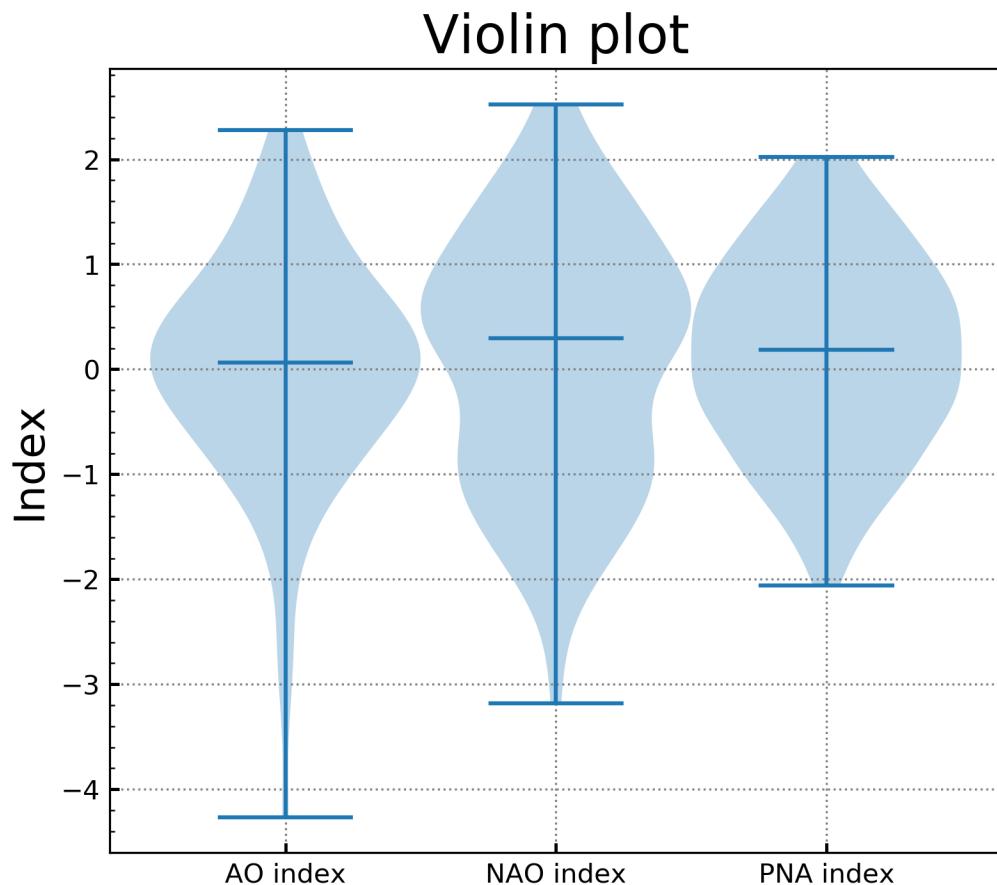


図 5-4-10 バイオリンプロットの幅を倍にして、中央値を横線で付けた

図 5-4-4 で NAO index だけ箱が縦長で平均値と中央値が離れていたのに

気がついていたかもしれません。箱の部分は 25~75 パーセンタイルの範囲に対応するので、50% の標本が含まれる範囲が広いことを表していますが、その範囲の上限と下限が -1σ ～ $+1\sigma$ 付近にまで達しています。正規分布では、68%程度の標本が -1σ ～ $+1\sigma$ の範囲に入るので、正規分布よりもかなり山が低く裾野が広い扁平な分布と推測されます。図 5-4-10 を見ると、NAO index は 2 つ山の分布をしているので、箱ひげ図の箱が縦長になるような扁平な分布がどのように生じていたか理解できるかと思います。また正の側の山に分布が偏っているので、中央値が平均値よりも高くなっていたと考えられます。このように、バイオリンプロットを作成して分布関数も分かるようにしておくと、結果の理解に役立つこともあります。

最後にバイオリンプロットで利用可能なオプションを表 5-4-2 にまとめおきます。バイオリンプロットのデザインを変更するオプションは、今の所実装されていないので、今後の更新が待たれます。

表 5-4-2 matplotlib の pyplot.violinplot で使用可能なオプション一覧

オプション	説明
x	入力データ
positions	バイオリンの位置を配列
vert	鉛直にするかどうか、[True False]、デフォルト値：True
widths	バイオリンの幅を配列、デフォルト値：0.5
labels	それぞれのデータセットのラベル（xと同じサイズ）、デフォルト値：None
showmeans	算術平均値を描く、[True False]、デフォルト値：False
showextrema	最小値、最大値のヒゲを描く、[True False]、デフォルト値：True
showmedians	中央値を描く、[True False]、デフォルト値：False
points	ガウス型のカーネル密度推定用いる点数、デフォルト値：100
bw_method	estimatorのバンド幅を決める方法、['scott' 'silverman']、デフォルト値：None ('scott')

使用方法： plt.violinplot(x)

5.4.7 バイオリンプロットのスタイルを無理やり設定する方法

バイオリンプロットにはスタイルを変更するためのオプションがありません。しかし、バイオリンプロットの各要素は matplotlib のオブジェクトから構成されているので、オブジェクトのスタイルを直接変更していけば、無理やりスタイルを変えることができます。どうしてもスタイルを変更したい時のために、その方法を載せておきます。

バイオリンプロットは、確率分布関数を表すボディの部分 ('bodies') が matplotlib.collections.PolyCollection オブジェクト、最小値 ('cmins')、最大値 ('cmaxes')、間のヒゲ ('cbars')、平均値 ('cmeans')、中央値 ('cmedians') が matplotlib.collections.LineCollection オブジェクトになっており、それらの組み合わせで作られています。violinplot_sample4.py では、平均値も追加したバイオリンプロットを描いた後で、それぞれのオブジェクトのスタイルを変更しています。最小値から最大値の範囲を表すヒゲと中央値は黒の実線、ボディは青の半透明、ボディの枠は黒線に変わり、平均値は青の破線で表示されました（図 5-4-11）。まず、平均値を描く（showmeans=True）ようにしてバイオリンプロットを作成し、戻り値を violin_parts に入れておきます。

```
violin_parts = ax.violinplot((data_x1, data_x2, data_x3),
                             widths=1.0,
                             showmeans=True,
                             showextrema=True,
                             showmedians=True)
```

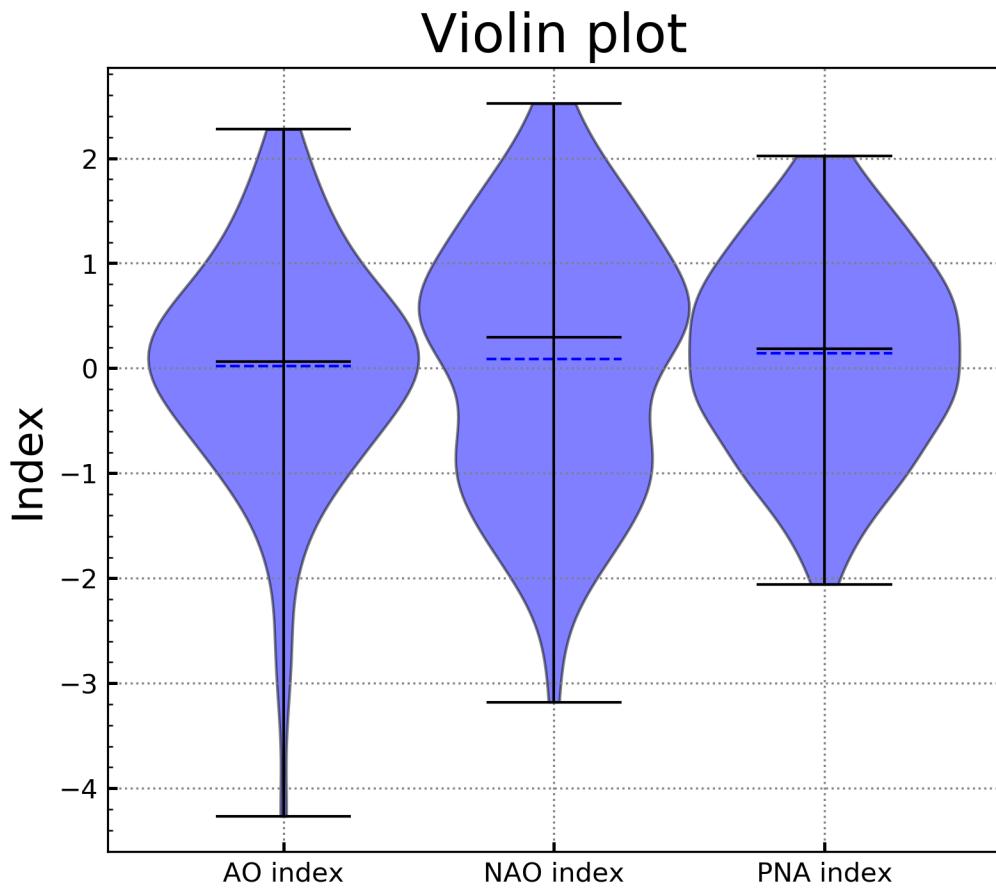


図 5－4－11 バイオリンプロットのスタイルを変更

`violin_parts['部分の名称']`に対してスタイルを設定していきます。まずは `LineCollection` オブジェクトに対して変更します。スタイルを設定する主要なメソッドとして、線の色を設定する `set_edgecolor`、線のスタイルを設定する `set_linestyle`、線の幅を設定する `set_linewidth` があります。

部分の名称をタプルにした `partnames`、対応する線の色をタプルにした `edgecolors`、線のスタイルをタプルにした `linestyles`、線の幅をタプルにした `linewidths` を定義しておきます。ここでは平均値を表す'cmeans'だけ青の破線に、他は黒の実線に設定しました。ループを回す際に `zip` を使い、それらを `partname`、`edgecolor`、`linestyle`、`linewidth` に順に渡していきます。ループ中では最初に `vp = violin_parts[partname]` のように各部に対してインスタンス `vp` を定義しておき、`vp` のメソッドを使用します。例えば、線の色が入っている `edgecolor` を使い、`vp.set_edgecolor(edgecolor)` で各部の線の色を設定します。

```

partnames = ('cbars','cmins','cmaxes','cmmedians', 'cmeans') # 部分の名称
edgecolors = ('k', 'k', 'k', 'k', 'b') # 線の色
linestyles = ('-', '--', ' ', ' ', '--') # 線のスタイル
linewidths = (1, 1, 1, 1, 1) # 線の太さ
for partname, edgecolor, linestyle, linewidth in zip(partnames, edgecolors, linestyles, linewidths):
    vp = violin_parts[partname]
    vp.set_edgecolor(edgecolor) # 線の色を設定
    vp.set_linestyle(linestyle) # 線のスタイルを設定
    vp.set_linewidth(linewidth) # 線の幅を設定

```

次に PolyCollection オブジェクトに対してスタイルを設定します。ボディの部分のインスタンス vp を for vp in violin_parts['bodies'] のように定義します。線の場合には set_edgecolor メソッドで線の色を設定しましたが、ボディの場合には set_edgecolor メソッドが枠線の色の設定に相当します。ボディの色は set_facecolor メソッドで設定します。不透明度も変更したいので set_alpha メソッドも使います。ここではボディを不透明度 0.5 の青色に、枠線を幅 1 の実線に設定しました。

```

edgecolor = 'k' # 枠線の色
facecolor = 'blue' # ボディの色
linewidth = 1 # 枠線の幅
alpha = 0.5 # ボディの不透明度
for vp in violin_parts['bodies']:
    vp.set_facecolor(facecolor) # ボディの色を設定
    vp.set_edgecolor(edgecolor) # 枠線の色を設定
    vp.set_linewidth(linewidth) # 枠線の幅を設定
    vp.set_alpha(alpha) # 不透明度を設定

```

5.5 様々なプロット

matplotlib には他にも様々なプロット機能があり、ここまでに紹介しきれなかったものを簡単に紹介しています。

5.5.1 グラフの中に画像ファイルを表示する

matplotlib にはデータをプロットする他に、画像ファイルを読み込み表示するための `plt.imshow` があります。先ほどの図 5-4-10 を保存した Fig5-4-10.png を使い、グラフ中に画像ファイルから読み込んだイメージを表示してみます (`imshow_sample.py`)。図 5-5-1 のように、図 5-4-10 が枠線の中に入ったものが作成されました。画像をファイルから読み込む際には、Pillow (PIL) の `Image` を利用します。

```
from PIL import Image
```

画像の読み込みは `Image.open("ファイル名")` です。読み込んだものを Numpy の `asarray` を使って `array` に変換し `X` に格納します。`plt.imshow(X)` では、変換した `X` を表示します。オプションの `aspect` はアスペクト比を決める方法を指定するもので、デフォルトではアスペクト比を 1 にする'equal'が設定されています。'auto'に変えると、軸に収まるように自動調整されます。今のケースでは 'equal'を試してみると、図が途中で切れてしまいます。

```
im = Image.open("Fig120.png") # 画像を読み込み  
X = np.asarray(im) # array に変換  
plt.imshow(X, aspect='auto') # 画像を表示
```

デフォルトのままでは枠線に目盛り線やラベルが表示されてしまうので、表示されないように `ticker.NullLocator` を指定します。

```
ax.xaxis.set_major_locator(ticker.NullLocator()) # x 軸の大目盛り  
ax.xaxis.set_minor_locator(ticker.NullLocator()) # x 軸の小目盛り  
ax.yaxis.set_major_locator(ticker.NullLocator()) # y 軸の大目盛り  
ax.yaxis.set_minor_locator(ticker.NullLocator()) # y 軸の小目盛り
```

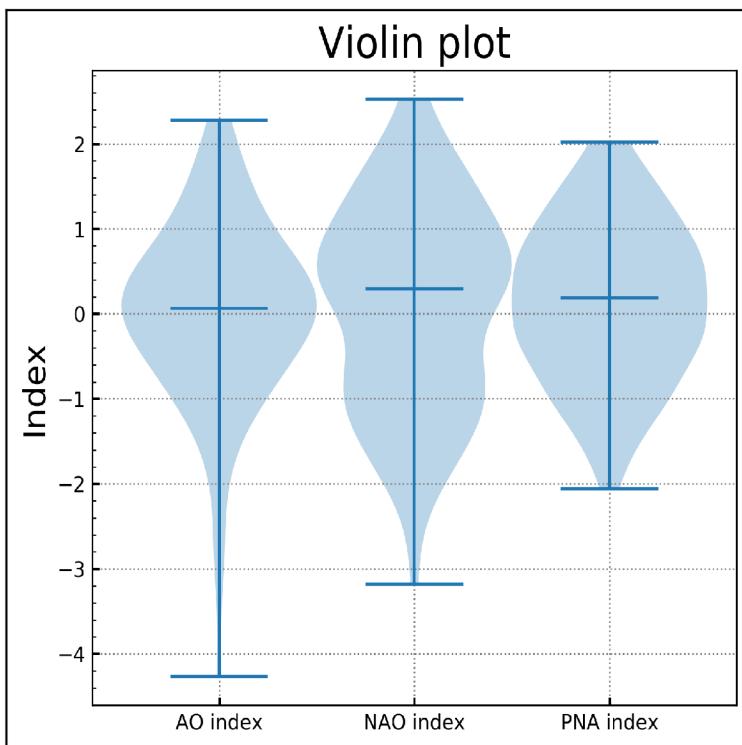
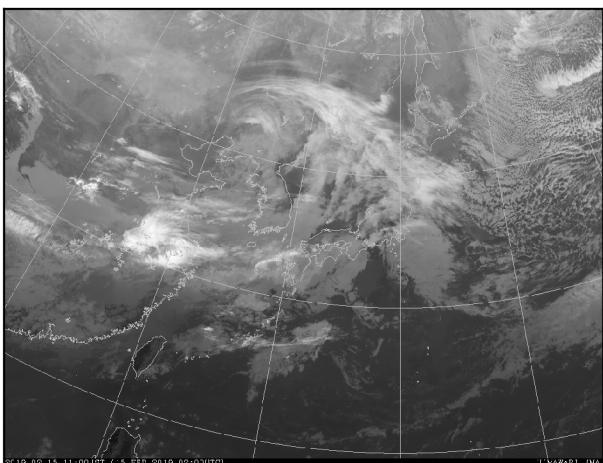


図 5－5－1 保存した画像を読み込み、plt.imshow でイメージを表示

この plt.imshow と urllib.request を組み合わせると、web 上の画像を読み込んで表示させることも可能です (imshow_sample2.py)。図 5－5－2 は 2019 年 2 月 15 日 11 時 00 分 (日本時間) の赤外画像と可視画像を表示したものです。この日には関東地方で降雪がありました。画像は気象庁のページから取得しています。なお、2021 年 3 月から気象庁で衛星画像の提供方法が変わったため、サンプルプログラムで取得するイメージとは少し異なります。現在、プログラム中で取得できる画像は 1 日分しか保存されていないので、サンプルプログラムでは 03UTC (日本時間正午) のものを取得するようにしています。

Infrared (201902151100)



Visible (201902151100)

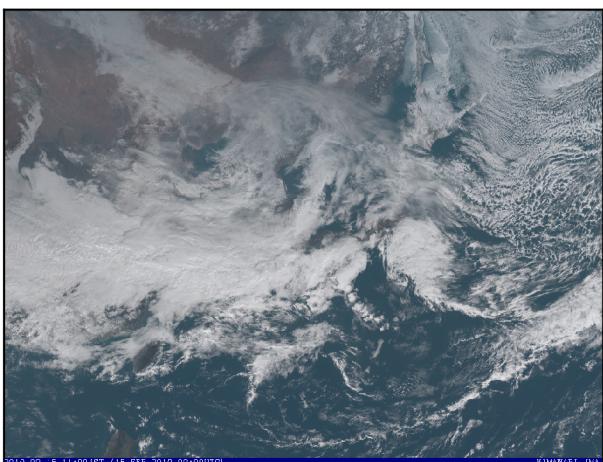


図 5 – 5 – 2 2019 年 2 月 15 日 11 時 00 分（日本時間）の赤外画像、可視画像を表示

まず `urllib.request` と `Pillow` の `Image` を import します。

```
import urllib.request  
from PIL import Image
```

赤外画像、可視画像の URL を設定しておきます (`url_ir`、`url_vis`)。また図の上にタイトルを付けられるように、`title_ir`、`title_vis` も設定します。タイトルに表示する時刻は `time` で指定します。ここでは、2022/3/12 12 時としています。

```
time = "202203121200"
# 赤外画像
title_ir = "Infrared (" + time + ")"
url_ir = ¥
"https://www.data.jma.go.jp/mscweb/data/himawari/img/jpn/jpn_b13_0250.jpg"
# 可視画像
title_vis = "Visible (" + time + ")"
url_vis = ¥
"https://www.data.jma.go.jp/mscweb/data/himawari/img/jpn/jpn_tre_0250.jpg"
```

空のリストを作成し、urllib.request.urlopen('URL')で、赤外画像、可視画像を読み込んでリストに追加していきます。タイトルについても同様です。

```
im = list() # 空のリストを作成
im.append(Image.open(urllib.request.urlopen(url_ir))) # 赤外画像
im.append(Image.open(urllib.request.urlopen(url_vis))) # 可視画像

title = list() # 空のリストを作成
title.append(title_ir) # 赤外画像のタイトル
title.append(title_vis) # 可視画像のタイトル
```

画像を表示する部分では、この im を使いますが、im の要素数だけサブプロットを生成して、サブプロット毎に別の画像を表示していきます。この処理では im の要素数分だけループを回していて、縦にサブプロットを追加しています。今のケースでは、n は 0、1 の値になりますが、サブプロットの番号は 1 から始まるので、add_subplot(len(im), 1, n+1)です。軸の場合は番号が 0 から始まるので ax[n].imshow(im[n], aspect='equal')のような番号になっています。なお、衛星画像の場合にはアスペクト比を元のように保ちたいので、aspect='equal'にしました。最後に、軸の目盛り線を消しタイトルを付けます。

```
ax = list()
for n in np.arange(len(im)):
    ax.append(fig.add_subplot(len(im),1,n+1)) # 軸の追加
    ax[n].imshow(im[n], aspect='equal') # 画像を表示

    ax[n].xaxis.set_major_locator(ticker.NullLocator()) # x 軸の大目盛り
    ax[n].xaxis.set_minor_locator(ticker.NullLocator()) # x 軸の小目盛り
    ax[n].yaxis.set_major_locator(ticker.NullLocator()) # y 軸の大目盛り
    ax[n].yaxis.set_minor_locator(ticker.NullLocator()) # y 軸の小目盛り
    ax[n].set_title(title[n]) # タイトルを付ける
```

5.5.2 流線関数を描く

矢羽や矢印を描くところで紹介し切れませんでしたが、似たようなプロットとして流線関数を描く機能もあります。流線関数を描くには `plt.streamplot` を使います。`ax.streamplot` でも同じです。流線関数は1次元では分かりにくいので、等高線を描いた時に用いた2次元プロットのグリッドを使って説明します。同じ2次元グリッド(X, Y)上に $(U, V) = (-1 - X^2 + Y, 1 + X - Y^2)$ となるような東西風、南北風の2次元データを作成します (`streamplot_sample.py`)。

```
n = 200
x = np.linspace(-10, 10, n) # x 軸データ
y = np.linspace(-10, 10, n) # y 軸データ
X, Y = np.meshgrid(x, y) # x, y 軸メッシュデータ
U = -1 - X**2 + Y # 東西風データ
V = 1 + X - Y**2 # 南北風データ
```

X、Y、U、V データを用いて流線関数を描きます (図 5-5-3)。色の指定が無ければ、自動的にグラディーションが付いた色に設定されますが、ここでは `color='k'` オプションで黒に設定しました。

```
ax.streamplot(X, Y, U, V, color='k') # 流線関数を描く
```

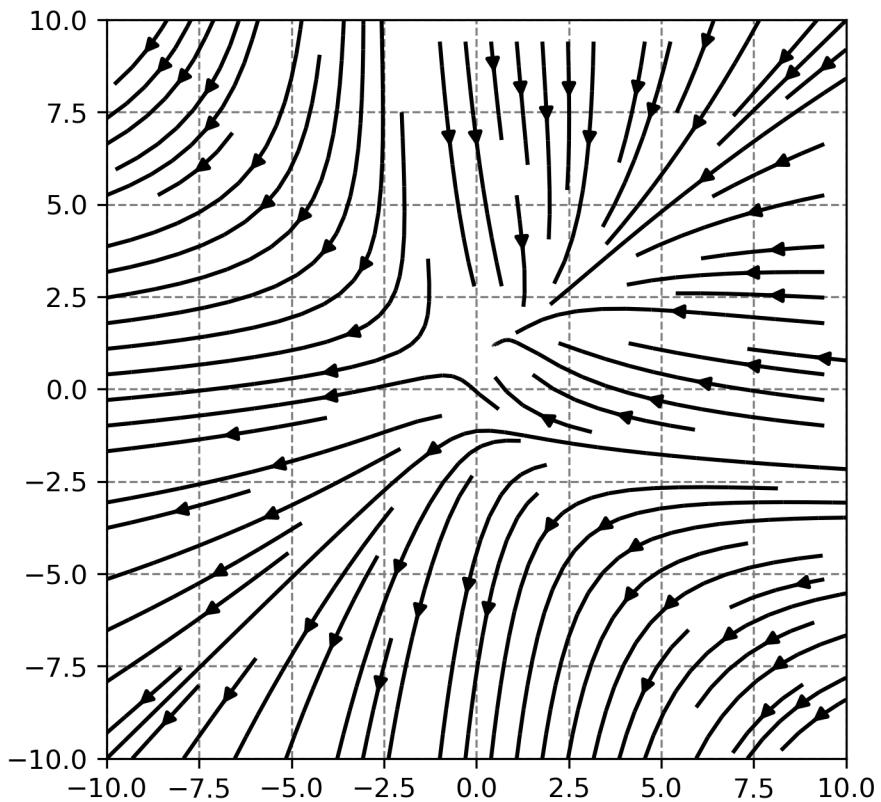


図 5－5－3 流線関数を描く

少し流線が混み合っているように見えるので、density オプションを使い間隔を広くしてみます。density=[x 軸方向の密度, y 軸方向の密度]のように x 軸と y 軸方向の密度を個別に指定することができ、デフォルト値は[1.0, 1.0]です。数字が大きいほど密度が高くなるので、density=[0.5, 1]のように指定すると、x 軸方向のみ間隔を広くすることができます（図 5－5－4）。作図に用いたプログラムは、streamplot_sample2.py です。x 軸、y 軸方向とも同じ値に設定したい場合には、density=0.5 のような指定も可能です。線の幅はデフォルトでは 1 ですが、linewidth オプションで変えることができ、ここでは 2 に設定しました。

```
ax.streamplot(X, Y, U, V, color='k', linewidth=2, density=[0.5, 1])
```

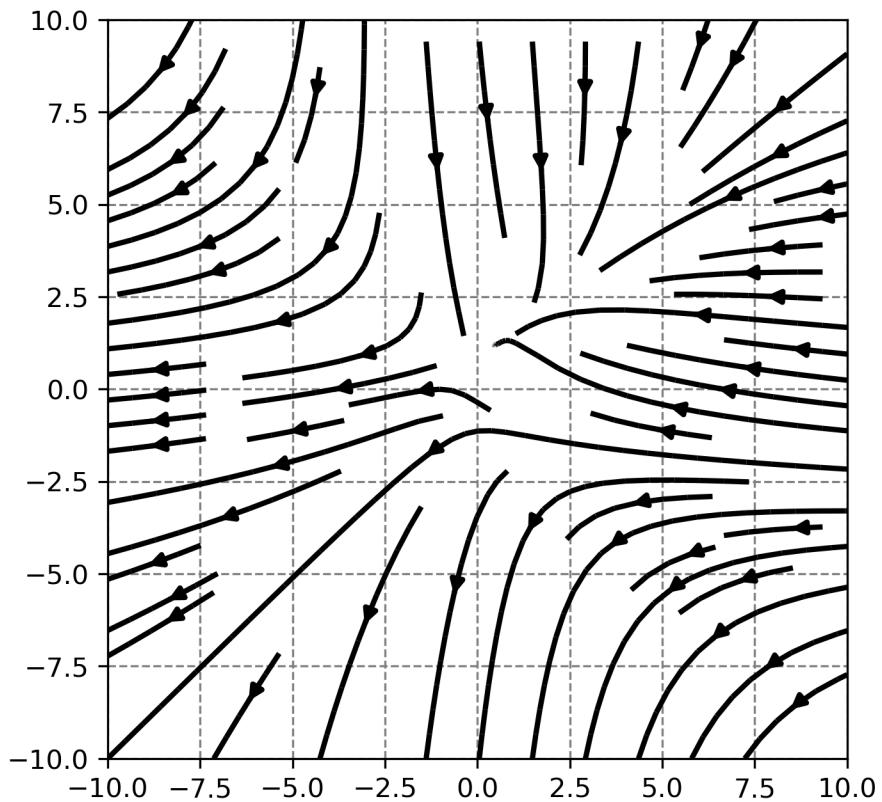


図 5－5－4 流線関数を描く間隔を広くし、線の幅を 2 倍に

流線関数の色を色テーブルで指定し、色の基準として風速を参照するといったプロットの仕方も可能になっています（図 5－5－5）。作図に用いたプログラムは、`streamplot_sample3.py` です。色テーブルの指定には `cmap` オプションを使い、`cmap=plt.cm.色テーブルの名前` のように行います。ここでは、図 4－5－7 の色テーブルの名前のうち `autumn` を使いました。

```
spd = np.sqrt(U**2 + V**2) # 風速
ax.streamplot(X, Y, U, V, color=spd, linewidth=2, cmap=plt.cm.autumn)
```

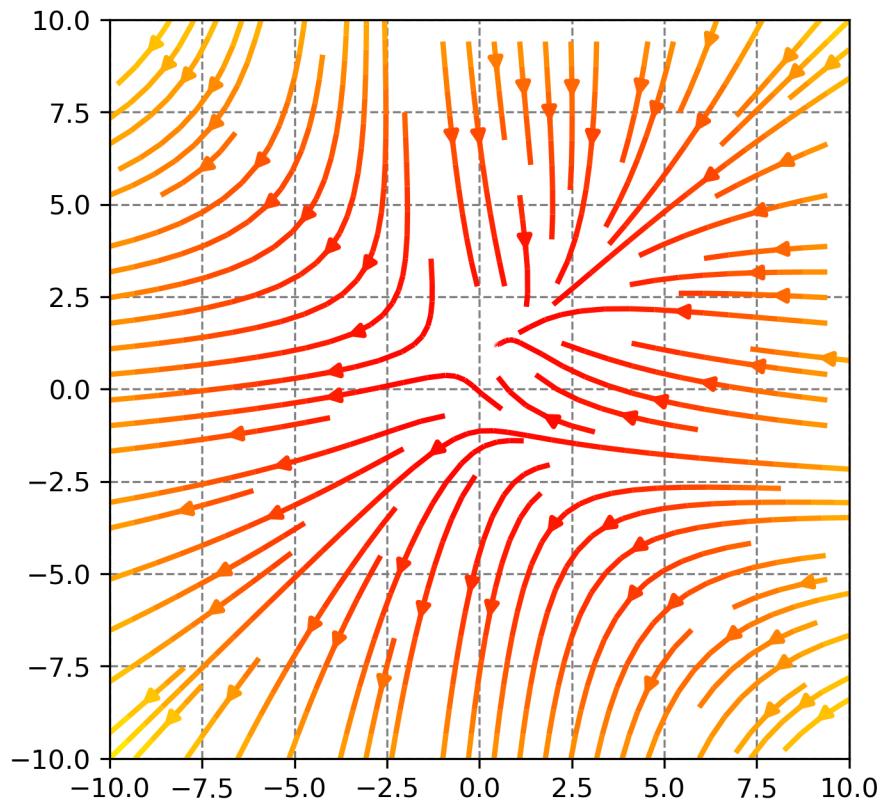


図 5－5－5 色を付けた

流線関数の作図では表 5－5－1 のようなオプションが利用可能です。

表 5－5－1 matplotlib の pyplot.streamplot で使用可能なオプション一覧

オプション	説明
X, Y (必須)	x軸, y軸に使用する配列、Zデータに対応した大きさの2次元配列で与える
U, V (必須)	東西風データ、南北風データの配列
levels	陰影を描くZの値のリスト
color	流線関数の色 ()、デフォルト値 : None (自動設定)
density	流線関数の密度、デフォルト値 : 1.0、例 : density=[0.5, 1]、density=0.5
linewidth	線の幅、デフォルト値 : 1
cmap	カラーマップ、デフォルト値 : None

使用方法 : plt.streamplot(X, Y, Z)

5.5.3 ログスケールのグラフ

折れ線グラフや散布図の所では紹介できませんでしたが、片対数グラフ、両対数グラフを作成したいこともあるかと思います。plt.plot や plt.scatter を使い折れ線グラフや散布図をプロットした後、軸のスケールを変更します。x 軸を対数グラフにする場合は ax.set_xscale("log") 、y 軸を対数グラフにする場合は ax.set_yscale("log")を行います。y 軸が気圧軸のデータを例に作図してみます (logplot_sample.py)。

作図には 2019 年 1 月 26 日 12UTC (21JST) の館野のゾンデデータを使用しており、この時、館野の 500 hPa 気温は 1 月の館野同時刻データのうち歴代 3 位の低さである -40.2°C を記録しました。作図されたものが図 5-5-6 です。500 hPa が -40°C 以下になっていたことが読み取れるかと思います。ちなみに 1 位は 1968 年 1 月 15 日の -40.7°C 、2 位は 1969 年 1 月 3 日の -40.4°C でした。

(<http://www.data.jma.go.jp/obd/stats/etrn/upper/view/urank.php?year=2019&month=1&day=26&hour=21&point=47646&atm=500&view=>)

入力ファイルは 20190126-21UTC_Tateno.csv というファイル名で、csv データになっています。csv データは Pandas の read_csv を使って読み込むことができ、pd.read_csv (ファイル名) の戻り値が Pandas の DataFrame になっています。読み込んだデータから気圧データ、気温データを取り出します。気圧データは 1 列目に入っており、1 列目を index にするために read_csv のオプションで index_col=[0] としています。そのため、dataset.index で気圧データが取り出されます。気温データは Pandas の loc を使い dataset.loc[:, "temp"] のように取り出します。

```
input_file="20190126-21UTC_Tateno.csv" # 入力ファイル名
dataset = pd.read_csv(input_file, index_col=[0]) # データの読み込み
pres = dataset.index # 気圧データ
temp = dataset.loc[:, "temp"] # 気温データ
```

x 軸を気温 (temp)、y 軸を気圧 (prep) として折れ線グラフを作成します。ax.set_yscale("log") で y 軸をログスケールにします。

```
ax.plot(temp, pres, color='k') # 折れ線グラフ  
ax.set_yscale("log") # y 軸をログスケールに
```

y 軸の範囲を指定しない場合、小さな値が下側、大きな値が上側になります。気象データのプロットとしては上空が下になるのは不便なので、y 軸の上下を指定することで上空を上にしています。

```
ymin = 1000  
ymax = 100  
ax.set_ylim([ymin, ymax])
```

現時点では、y 軸の表示を `ticker.AutoLocator()` で自動設定しています。

```
ax.yaxis.set_major_locator(ticker.AutoLocator()) # y 軸大目盛り  
ax.yaxis.set_minor_locator(ticker.AutoLocator()) # y 軸小目盛り
```

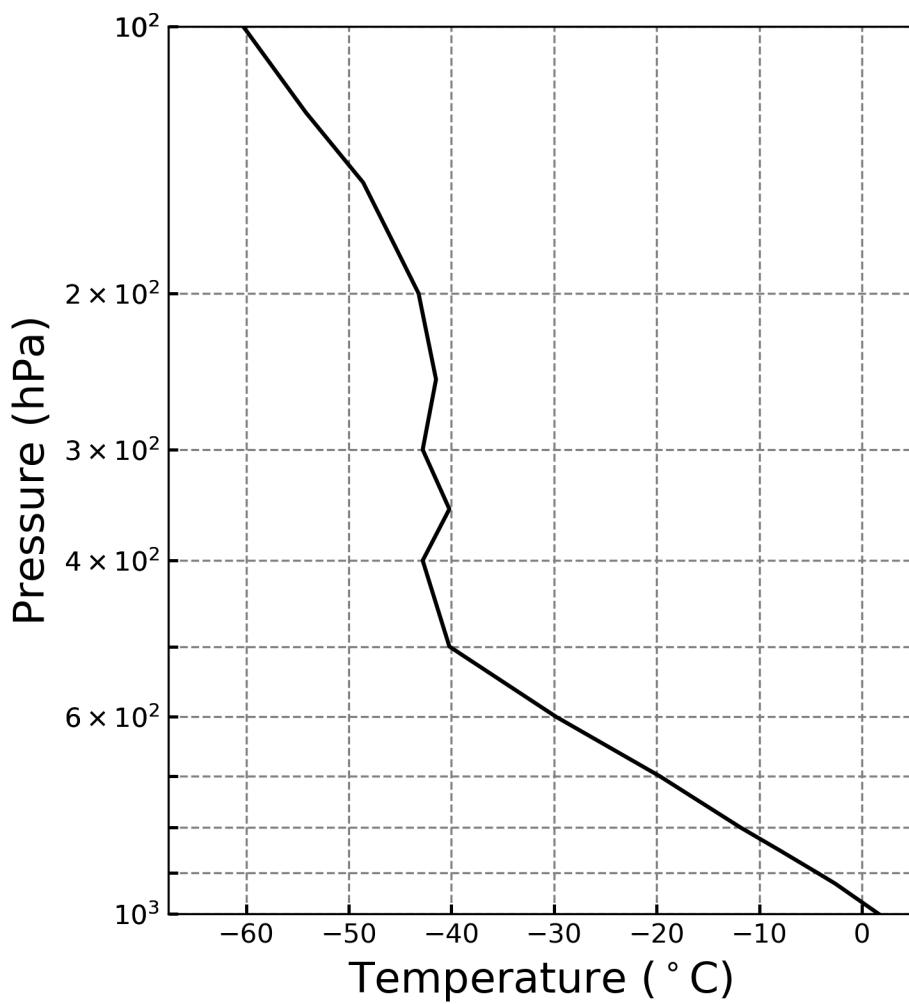


図 5-5-6 2019 年 1 月 26 日 12UTC (21JST) の館野のゾンデデータの気温の鉛直プロファイル。y 軸を対数にした。

自動設定では y 軸が指数表記になるので、図 5-5-7 のように 1000 hPa、100 hPa のような表記に変えてみます。作図には `logplot_sample2.py` を使いました。locator に関しては先ほどと同じ `ticker.AutoLocator()` を使いましたが、formatter を使う時に `ticker.FuncFormatter(関数名)` で書式設定するように工夫しています。まず `major_formatter` 関数を定義し、`return "%.Of" % x` のように入力された `x` に対して小数点以下を表示しないように書式設定したものを返却します。その `major_formatter` 関数を `ticker.FuncFormatter` の引数として与えることで、これまで指数表示だったものを整数で表示させます。もし小数点以下 2 桁まで表示させたければ、`"%.2f" % x` のような戻り値を使います。

```

def major_formatter(x, pos): # Func formatter
    return "%.{0}f" % x
ax.yaxis.set_major_locator(ticker.AutoLocator()) # 大目盛り
ax.yaxis.set_minor_locator(ticker.AutoLocator()) # 小目盛り
ax.yaxis.set_major_formatter(ticker.FuncFormatter(major_formatter))
ax.yaxis.set_minor_formatter(ticker.FuncFormatter(major_formatter))

```

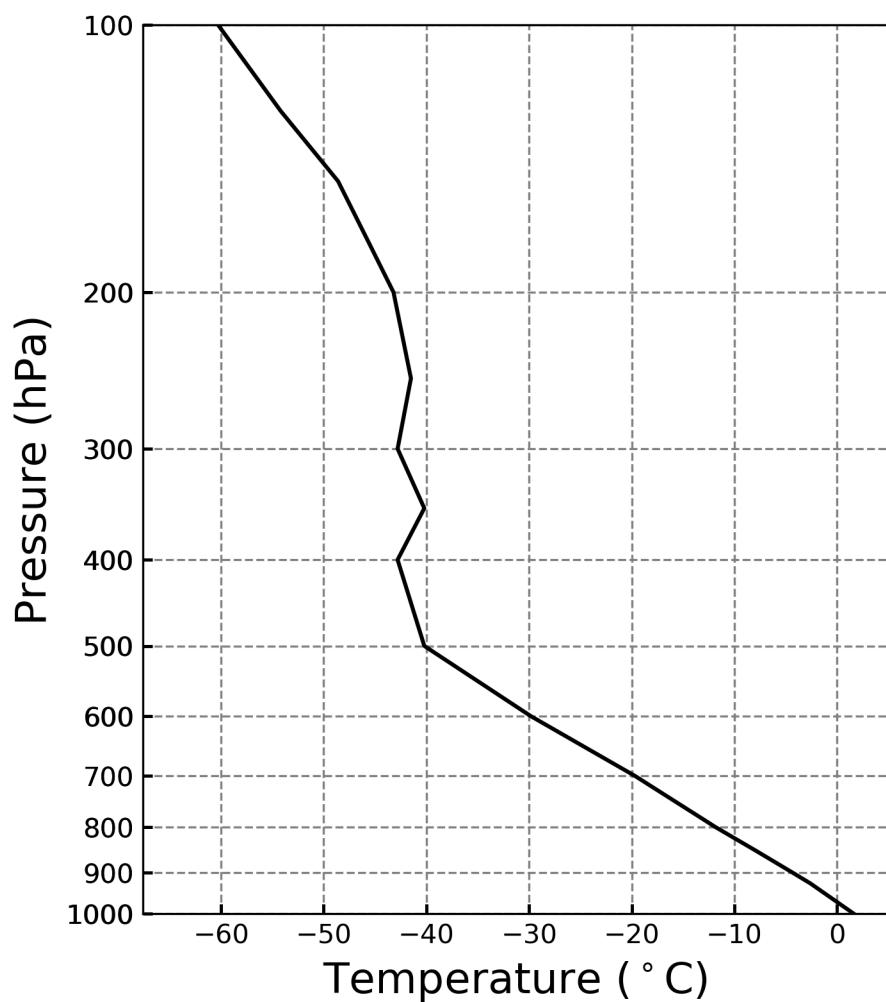


図 5－5－7 y 軸の表示を変更する。

なお plt.plot の代わりに plt.semilogx (x 軸対数)、plt.semilogy (y 軸対数)、plt.loglog (両対数) を使っても片対数グラフ、両対数グラフが作図可能です。

5.5.4 時系列図にラベルを付ける

時系列図に矢印とラベルを付け、その日に起こったイベントなどを図中に示すことも可能です。ラベルを付けるには、`plt.annotate` を使います。ここでは、2018年12月から2019年1月にかけて起こった成層圏突然昇温の事例を用いて折れ線グラフにラベルを描いてみます(`annotate_sample.py`)。この時期には、12月中旬に小昇温 (minor warming) が2度起こった後、12月25日に小昇温が起り、そのまま1月2日には大昇温 (major warming) の基準を満たすまでになりました(図5-5-8)。なおWMOによる小昇温の基準は、10 hPa、90°Nの東西平均した気温が60°Nの東西平均した気温を上回った場合、大昇温の基準はさらに10 hPa、60°Nの東西風が東風となった場合です。

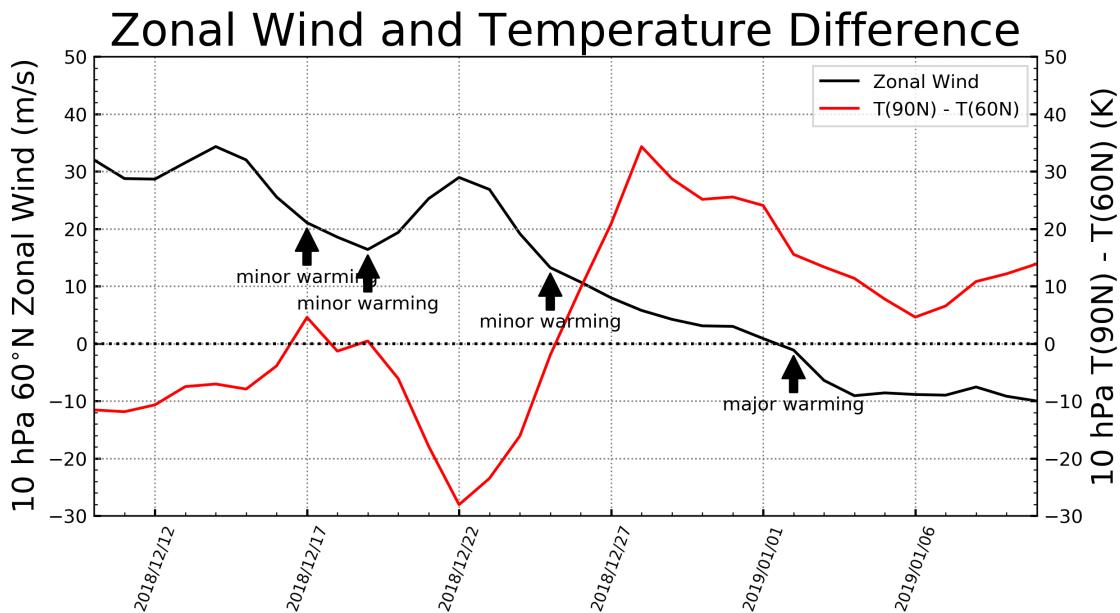


図5-5-8 折れ線グラフにラベルを付ける。JRA-55客観解析データの10 hPa、60°N東西風速（黒線）と90°Nと60°Nの気温差（赤）の時間変化。

作図に用いたデータはJRA-55客観解析データの10 hPa、60°Nの東西平均東西風と90°Nと60°Nで東西平均した気温の差です。入力ファイルはスペースで区切られたヘッダーのないデータになっており、Pandasの`read_fwf`を使うことで`pd.read_fwf(ファイル名, header=None)`のように読み込んでいます。3～5列目が年、月、日のデータになっており、これらのデータを処理して時間軸データとして扱うため、`parse_dates=[[2, 3, 4]]`とします。`parse_dates`オプ

ションは、時刻データとして処理する列番号を与えるもので、単独の列番号でも複数の列番号のリストでも処理可能です。ここでは、時刻データとして扱う列番号をリストで与えています。複数の列番号を並べた場合には、年、月、日順に列番号をリストにしておきます。ここでは3～5列目を使いたいので、[2,3,4]のようにしました。次のindex_col=[0]は、最初の列をindexにするためのオプションです。時刻データの処理を行なった場合、時刻データが最初の列に来ているのでindex_col=[0]は時刻データをindexにすることを意味しています。データは全部で15列からなっているので、15列分の名前を要素にしたタプルcol_namesを定義しておき、names=col_namesで読み込んだデータに列の名前を設定します。最後のkeep_date_col=Trueは、parse_datesで処理した時刻データの他にオリジナルの、year、month、dayの列を残しておくことを意味しています。

```
col_names=('rec_num', 'ref_year', 'year', 'month', 'day', 'ihh', 'time', 'u', ¥
           't', 'tdiff', 'flg1', 'flg2', 'flg3', 'flg4', 'flg5')
dataset = pd.read_fwf(input_file, header=None, parse_dates=[[2, 3, 4]], ¥
                      index_col=[0], names=col_names, keep_date_col=True)
```

読み込んだデータから東西風(dataset.u)、気温差(dataset.tdiff)データを取り出し折れ線グラフを作図します。その際に東西風は左側の軸(ax[0])、気温差は右側の軸(ax[1])の側に描きます。

```

ylab1 = "10 hPa 60°N Zonal Wind (m/s)" # 左側のラベル
ylab2 = "10 hPa T(90N) - T(60N) (K)" # 右側のラベル

ax=list()
fig=plt.figure(figsize=(9, 5)) # プロットエリアの定義
ax.append(fig.add_subplot(1, 1, 1)) # サブプロットの生成 (ax[0])

p0 = ax[0].plot(dataset.index, dataset.u, color='k', label=l0) # 東西風折れ線
ax[0].set_ylabel(ylab1, fontsize=16) # y 軸のラベル
ax[0].set_ylim([-30, 50]) # y 軸の範囲 (左)

ax.append(ax[0].twinx()) # 2 つのプロットを関連付ける
p1 = ax[1].plot(dataset.index, dataset.tdiff, color='r', label=l1) # 気温折れ線
ax[1].set_ylabel(ylab2, fontsize=16) # y 軸のラベル
ax[1].set_ylim([-30, 50]) # y 軸の範囲 (右)

```

2 軸グラフを作成する際に、左側の軸は `ax.append(fig.add_subplot(1,1,1))`、右側の軸は `ax.append(ax[0].twinx())` のような記述をしています。これらは新たに生成した軸のインスタンスをリストに追加している違いがあるだけで、他は `ax0= fig.add_subplot(1, 1, 1)`、`ax1=ax0.twinx()` と同じ処理を行っています。なお生成されたインスタンスはリスト `ax` の要素になっているので、次のようにまとめて目盛り線やグリッド線の指定が行えます。

```

for n in np.arange(2):
    ax[n].xaxis.set_major_locator(ticker.AutoLocator())
    ax[n].xaxis.set_minor_locator(ticker.AutoMinorLocator())
    ...
    ax[n].axhline(y=0, color='k', ls=':') # y=0 の線を付ける
    ax[n].grid(color='gray', ls=':') # グリッド線を描く

```

ラベルと矢印を付ける部分です。まず `ssw_info` というリストを定義して時刻、

ラベルの文字列、矢印を x 軸方向にずらす値の組合せをタプルにし、リストの 1 つの要素として格納しておきます。

```
ssw_info = [  
    (datetime(2018, 12, 17, 0, 0, 0), "minor warming", 0),  
    (datetime(2018, 12, 19, 0, 0, 0), "minor warming", 0),  
    (datetime(2018, 12, 25, 0, 0, 0), "minor warming", 0),  
    (datetime(2019, 1, 2, 0, 0, 0), "major warming", 0)  
]
```

この ssw_info から data、label、hour を取り出し、ax[1].annotate に渡します。1 つ目の引数はラベルの文字列で、xy が矢印の位置、xytext がラベルの位置です。いずれもタプルで与えており、(x 軸上の位置, y 軸上の位置) のように与えます。矢印のプロパティは、arrowprops=dict(facecolor='k') のように与えており、ここでは色を黒に指定しました。水平、鉛直方向に基準に対してどのように配置するのかを horizontalalignment='center'、verticalalignment='top' で指定します。

```
for date, label, hour in ssw_info:  
    ax[1].annotate(label,  
                  xy=(date + timedelta(hours=hour),  
                        dataset.loc[date, "u"] - 1),  
                  xytext=(date + timedelta(hours=hour),  
                          dataset.loc[date, "u"] - 8),  
                  arrowprops=dict(facecolor='k'),  
                  horizontalalignment='center',  
                  verticalalignment='top')
```

5.5.5 時系列データの移動平均

折れ線グラフを作図する際に、移動平均した値を並べてプロットしたい場合もあるかと思います。移動平均は matplotlib そのものではなく、Numpy に含まれている np.convolve を使います。np.convolve(入力データ配列, フィルタ一配列, mode)のように用いますが、1番目の引数は長さ N の配列、2番目の引数は長さ M の配列で $M \leq N$ です。移動平均したいデータを 1番目の引数に、移動平均に用いるフィルターを 2番目の引数とします。mode='same'で出力が長さ N になり、mode='full'（デフォルト値）では元のデータの長さよりも長く $N+M-1$ 、mode='valid'で 1番目と 2番目の配列の範囲が重ならない部分（移動平均に用いるデータが完全に得られない端の部分）は出力から除かれます。

2019年2月の札幌では、8日に日最高気温が -10°C を下回りました。この事例を含むように、ここでは 2019 年 2 月の札幌のアメダス地点データを利用して、折れ線グラフに 5 日移動平均も並べてみます（図 5-5-9）。作図に用いたプログラムは、runmean_amedas_day.py です。

取り出した最高気温データの配列が tmax です。num が移動平均を行う日数で、np.ones(num)/num のように日数分の範囲で要素毎の重みが同じ配列を用意しておきます。ここでは 5 日移動平均（num=5）としたので、隣接する 5 データに対して $1/5$ 毎の重みを掛けた積算が行われます。最後の mode='same'で出力が入力の時系列と同じ長さ N になります。

```
num = 5 # 移動平均の日数
tmax = dat_i.loc[:, 'tmax'] # 日最高気温データの取り出し
y2 = np.convolve(tmax, np.ones(num) / num, mode='same') # 移動平均
```

日最高気温のデータを折れ線グラフにし、さらに 5 日移動平均を重ねます。8 日は日最高気温が -10°C を下回っていますが、大きく下がっていたのはその日だけだったので、5 日移動平均では平坦な時間変化となりました。

```
index = tmax.index # 時間軸データ
ax.plot(index, tmax, color='r', ls='-', label='Max. Temp.') # 日最高気温
ax.plot(index, y2, color='gray', ls='--', label='Max. Temp. (5-days running mean)') # 5 日移動平均値
```

Daily timeseries of Feb. 2019, Sapporo

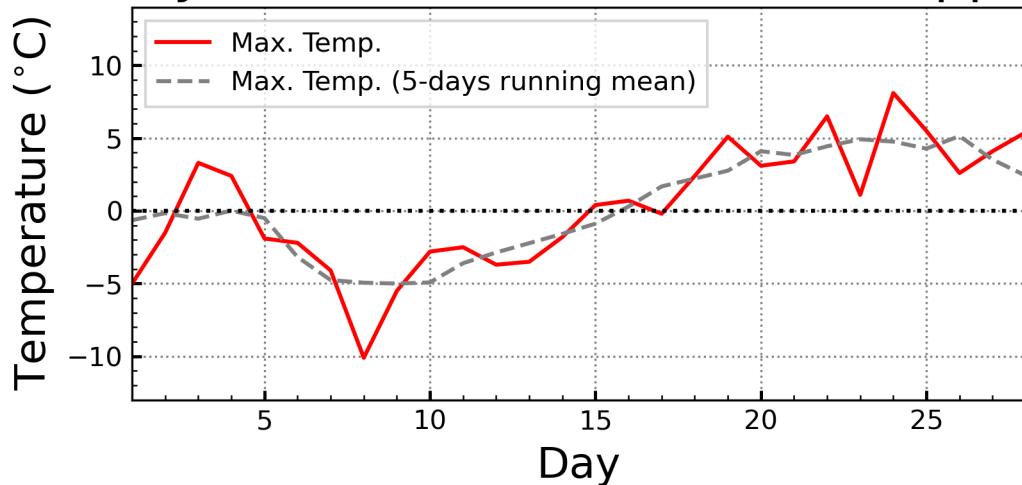


図 5－5－9　日最高気温（赤線）と 5 日移動平均値（灰色線）。2019 年 2 月の札幌のアメダス地点データから作成。

なお np.convolve は、本来は畳み込み積分を行うためのものです。移動平均は、元のデータを表す関数に移動平均を行うフィルター関数（ある時刻の周辺では、移動平均する日数分だけ 0 でない関数）を乗じたものを積分しているため、畳み込み積分の一種であると言えます。なお元のデータを表す関数を $f(t)$ とし、フィルター関数を $g(t)$ として、 $h(t) = \int_{-\infty}^{\infty} d\tau f(t)g(t - \tau)$ のような計算を行うのが畳み込み積分です。 $g(t)$ に移動平均に相当する $g(t) = 1/n$ for $|t - \tau| \leq n/2$, $g(t) = 0$ for elsewhere (t の周囲でのみ値を持ち他は 0) のような関数を定義しておけば、得られる $h(t)$ には t の周囲で平均された値が入るため、移動平均と等価であることが理解できるかと思います。

5.6 プロットの調整

これまでにプロットの調整方法をいくつか紹介していましたが、重要なものについてまとめておきます。

5.6.1 プロット範囲を調整する

まずプロット範囲を調整する方法を紹介します。プロット範囲の調整には、これまでに紹介していた plt.subplots_adjust を用います（表5-6-1）。

表5-6-1 matplotlib の pyplot.subplots_adjust で使用可能なオプション一覧

オプション	説明
left	図の左端、デフォルト : 0.125
right	図の右端、デフォルト : 0.9
bottom	図の下端、デフォルト : 0.1
top	図の上端、デフォルト : 0.9
wspace	サブプロット間の横幅の空き（軸の幅に対する比率）、デフォルト : 0.2
hspace	サブプロット間の縦幅の空き（軸の高さに対する比率）、デフォルト : 0.2

plt.subplots_adjust のオプションのうち図の範囲を決めるものが、図の左端（最も左側にあるサブプロットの左端）の位置を指定する left、図の右端（最も右側にあるサブプロットの右端）の位置を指定する right、図の下端（最も下側にあるサブプロットの下端）を指定する bottom、図の上端（最も上側にあるサブプロットの上端）を指定する top です（図5-6-1）。いずれもプロットエリア全体の大きさに対する比率で指定します。タイトルや目盛り線、目盛り線ラベルなどはサブプロットに含まれるので、図5-6-1のサブプロットの枠線は、それらの外側を表します。サブプロット間の空きは、wspace と hspace で指定します。これらは、サブプロット間の横幅の空きと縦幅の空きに対応しますが、全体に対してではなく軸の幅と高さに対する比率で指定します。なお左右のサブプロットの y 軸同士、上下のサブプロットの x 軸同士を結合して表示したい場合には、それぞれ wspace=0.0、hspace=0.0 です。

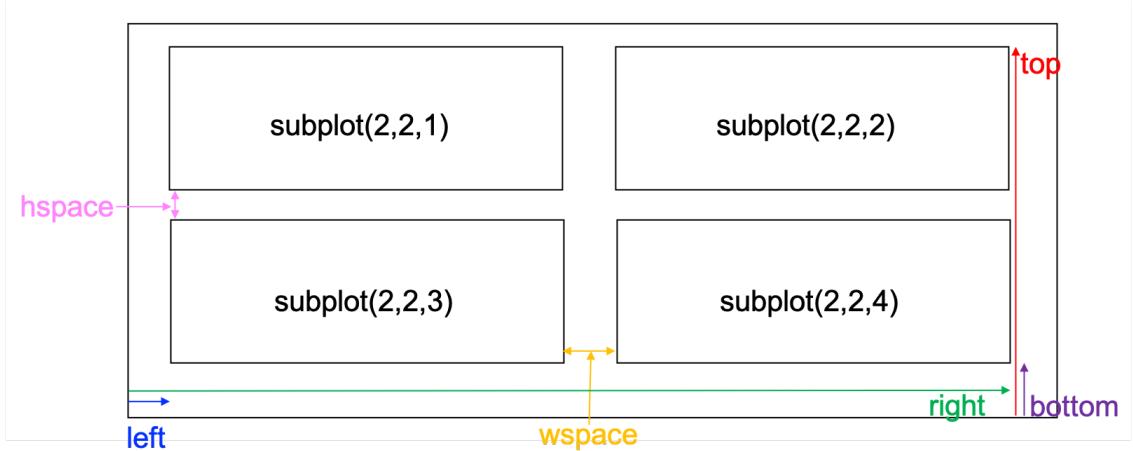


図 5 – 6 – 1 plt.subplots_adjust オプションに対応するもの。サブプロットが 4 つの場合

図 5 – 6 – 1 ではサブプロットが 4 つの場合を示しましたが、図の範囲の指定は単独のサブプロットでも必要になることがあります。例えば、x 軸を時刻として縦長にすると、YYYY-MM-DD のような表示が図の下からはみ出してしまうですが、bottom=0.15 などデフォルトよりも大きな値にすることで、図の中に収めることができます。

5.6.2 凡例の位置を調整する

これまで凡例の位置は loc オプションでしか調整していませんでしたが、手動で位置を指定することも可能です。オプションに bbox_to_anchor があり（表 4 – 3 – 1）、図全体の左下を(0, 0)、右上を(1, 1)として、それに対する位置（アンカー）をタプルで与えることが可能になっています。loc オプションを同時に使い、アンカーに凡例の枠のどこを合わせるかを指定します。例えば、loc='upper left'とした場合には、アンカーに凡例の枠の左上が来るようになります。なお bbox_to_anchor の指定がない場合、loc オプションは凡例を図中のどこに配置するのかを表します（図 3 – 2 – 5）。ややこしいですが、bbox_to_anchor を使う場合には loc オプションの仕様が変わってしまうようです。

図 5 – 5 – 8 の凡例を使い、位置の変更を試してみます (legend_sample.py)。なお図 5 – 5 – 8 の場合には、アンカーが図の右上となる loc='upper right'、アンカーに凡例の枠の右上がくるような bbox_to_anchor=(1, 1) に自動設定され

ていました。ここでは、`bbox_to_anchor=(1, 0.8)`として、アンカーのy軸位置を下側に変更しました（図5-6-2）。

```
plt.legend((p0[0], p1[0]), (l0, l1), loc='upper right', bbox_to_anchor=(1, 0.8))
```

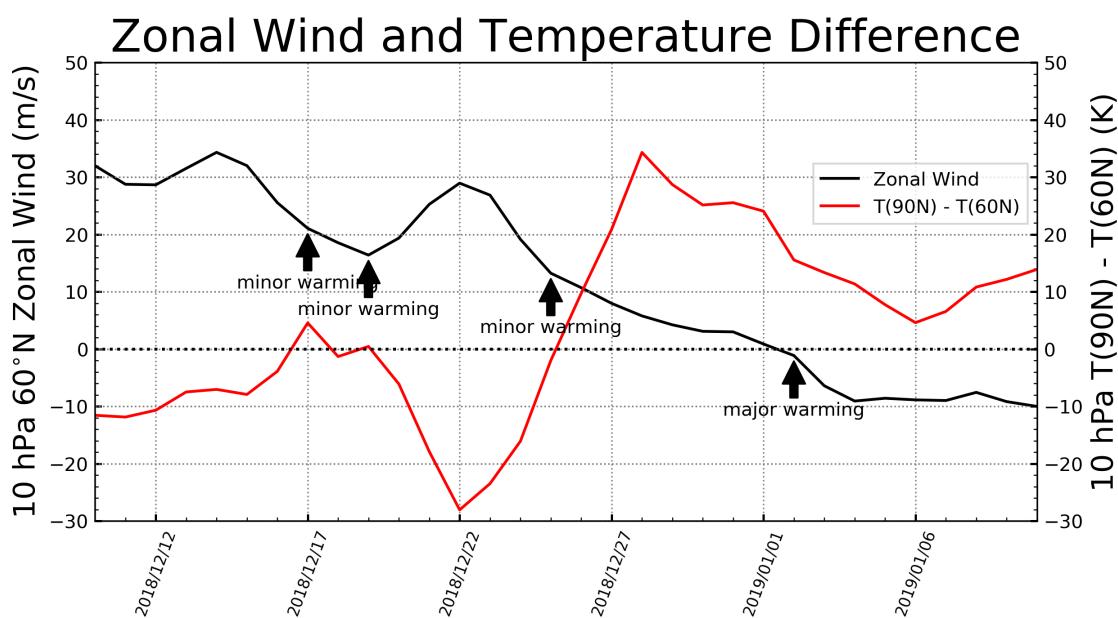


図5-6-2 図5-5-8の凡例の位置を下側に変更

`borderaxespad`というオプションもあり、凡例の枠とアンカーとの隙間を変更することができます。分かりやすくするために、凡例の右端が図の右上に来るような`loc='upper right'`、`bbox_to_anchor=(1, 1)`にしておきます。図の枠に凡例の枠が重なります（図5-6-3）。作図は`legend_sample2.py`で行いました。

凡例枠の右上をアンカーに合わせるため、`borderaxespad=0`にします。なお`borderaxespad`のデフォルト値は、`rcParams["legend.borderaxespad"]`で設定されており、変更しない場合は0.5となっています。

```
plt.legend((p0[0], p1[0]), (l0, l1), loc='upper right', bbox_to_anchor=(1, 1), borderaxespad=0)
```

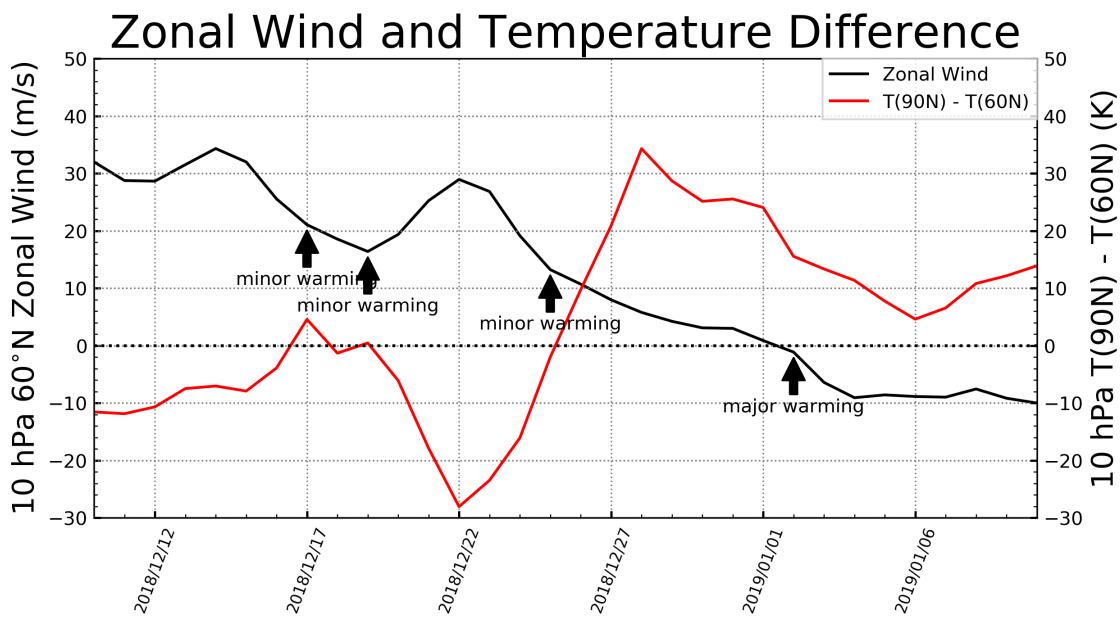


図 5 – 6 – 3 凡例枠を図の右上に合わせる

アンカーの位置は、 $(0, 0) \sim (1, 1)$ の範囲外に設定することもできて、凡例を図の外側に配置することが可能です（図 5 – 6 – 4）。`bbox_to_anchor=(1.05, 1.1)`のように図全体の右側、上側にアンカーを設定し、`loc='upper left'`、`borderaxespad=0`で凡例の左上がちょうどアンカーに重なるようにしています。作図には、`legend_sample3.py` を用いました。

```
plt.legend((p0[0], p1[0]), (l0, l1), loc='upper left', bbox_to_anchor=(1.05, 1.1),
           borderaxespad=0)
```

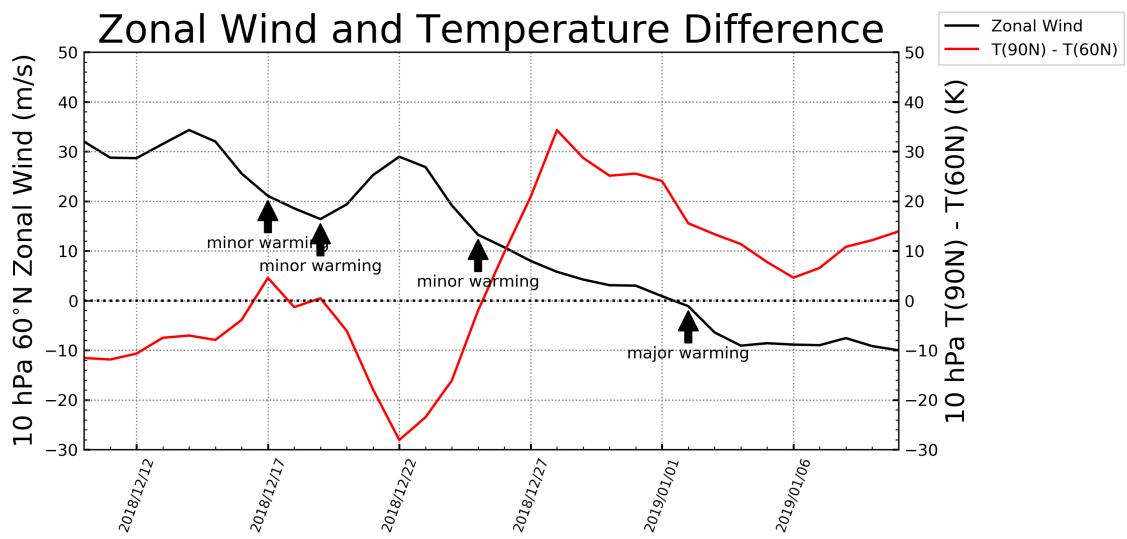


図 5 – 6 – 4 凡例を図の外側に配置する