Computer Lab 7: Land snow cover

Climate Data Analysis, ATS 301, Fall 2018

As before, we start by specifing that we want plots to be displayed inside the Jupyter Notebook, and load the modules we'll need.

```
In [ ]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        # Set default figure size
        # This is so I don't have to keep specifying figsize later.
        plt.rcParams['figure.figsize'] = (10.0, 8.0)

        import xarray as xr
        import cartopy.crs as ccrs
        from scipy import stats
        import math
```

Introduction to the dataset

```
In [ ]: snow_file='/data/ATS_301/Data/nhsce_v01r01_19661004_20181001.nc'
        dsnow = xr.open_dataset(snow_file)
```

The reference for this dataset is:

Robinson, David A., Estilow, Thomas W., and NOAA CDR Program (2012):NOAA Climate Date Record (CDR) of Northern Hemisphere (NH) Snow Cover Extent (SCE), Version 1. NOAA National Climatic Data Center. doi:10.7289/V5N014G9 [accessed 10/19/2018].

Start, as usual, by looking at the basic information in the file:

```
In [ ]: dsnow
```

Notice the dimensions of the latitude and longitude arrays:

```
In [ ]: dsnow.latitude
```

```
In [ ]: dsnow.longitude
```

This dataset uses a non-regular grid. We'll get back to this in a bit.

Now, look at the three "Data Variables" (ignore coord_system). What data do they each contain?

```
In [ ]: dsnow.land
```

```
In [ ]: dsnow.snow_cover_extent
In [ ]: # This is just an subset of the data so we can see example values.
        # The specific time, lats, and lons aren't important right now.

        dsnow.snow_cover_extent[0,30:35,:].values
```

Note all snow cover extent values are either 0 or 1.

```
In [ ]: dsnow.area
```

```
In [ ]: # quick look at data for one time
        dsnow.snow_cover_extent[0,:,:].plot()
```

Because the values are all either 0 or 1, the plot for a single time only has two colors. Average over all the times to get the snow climatology.

```
In [ ]: # time average at every point. This will produce values other than 0 and 1 to make
        something
        # nicer to plot.

        time_average_snow=dsnow.snow_cover_extent.mean(dim='time')
        time_average_snow.plot()
```

Excluding ocean grid points

This plot doesn't show the difference between land with no snow and ocean. Fortunately, the `land` variable (in the netcdf file) indicates where there is land (where it is =1). 0 corresponds to ocean. We can create a new variable, equal to `snow_cover_extent` when there is land (`dsnow.land>0`), and `nan` where there is ocean.

```
In [ ]: # False if land = 0, True is land = 1 (i.e., >0)
        # Only a slice of the array shown

        dsnow.land[31,:].values > 0
```

```
In [ ]: snow = dsnow.snow_cover_extent.where(dsnow.land > 0)
        snow[0,31,:].values
```

Now we can use this new snow variable for our plot. Fortunately, the `xr.plot` function omits `nan` 's by default.

```
In [ ]: time_average_snow=snow.mean(dim='time')
        time_average_snow.plot()
```

Weird coordinates...

So this looks a little odd. What's going on?

Looking at those latitudes and longitudes again...

```
In [ ]: dsnow.longitude.plot()
In [ ]: dsnow.latitude.plot()
```

Fortunately, python can deal with these weird 2-d lats and lons. We just pass them to the xarray `plot` function.

```
In [ ]: # We have to specify the x and y values, since the coord dimens are col and row,
        # which don't correspond to lat and lon.
        time_average_snow.plot(x='longitude',y='latitude')
```

To make this plot, use `cartopy` and switch to the orthographic projection. We want the N. Pole in the middle. Use `(0,90)` to indicate the center at 0 deg long, 90 deg lat.

```
In [ ]: ax = plt.axes(projection=ccrs.Orthographic(0, 90))

        # We have to specify the x and y values, since the coord dimens are col and row,
        # which don't correspond to lat and lon.
        time_average_snow.plot(x='longitude',y='latitude',ax=ax,transform=ccrs.PlateCarree
        ())
        plt.show()
```

Change the colormap to `Greys_r`, so that white=snow.

```
In [ ]: ax = plt.axes(projection=ccrs.Orthographic(0, 90))
        time_average_snow.plot(x='longitude',y='latitude',ax=ax,transform=ccrs.PlateCarree
        (),cmap='Greys_r')
        plt.show()
```

We can add in a prettier ocean, using the `ax.stock_img()` command. We'll do this first, and then put the snow on top of it. We'll still see the ocean (where `time_average_snow` is `nan`). This is one advantage of using the `land` variable to distinguish land from ocean points.

```
In [ ]: ax = plt.axes(projection=ccrs.Orthographic(0, 90))
        ax.stock_img()
```

```
In [ ]: ax = plt.axes(projection=ccrs.Orthographic(0, 90))

        # This needs to be before the plot function, so that it is at the bottom of the fi
        gure.
        ax.stock_img()
        time_average_snow.plot(x='longitude',y='latitude',ax=ax,transform=ccrs.PlateCarree
        (),cmap='Greys_r')
        plt.show()
```

Probability distributions

We'll start by plotting the histogram for all data points and times.

Note: sometimes it will take a long time calculating the histogram. In these cases, specify `range=(0,1)` to help `plt.hist` out a bit. It gets confused by the fact that there are only 2 possible values.

```
In [ ]: snow.plot.hist()
```

To convert this histogram to a **probability distribution** plot, use the `density=True` option.

```
In [ ]: snow.plot.hist(density=True)
```

`density=True` will make integral of histogram sum to 1. The total count for all bins is 10. Since the bins are each 0.1 wide, this integrates to 1.

What sort of **distribution** is this?

Another way to create this plot is using the `plt.hist` function [note that this goes before the variable]. It doesn't deal with missing values very well, but it will let us include two histograms next to each other later. Since we have missing values, we have to explicitly indicate the range using `range=(0,1)` . We'll still get a warning, but at least it will plot.

The `np.ravel` function converts the 3-d array into a 1-D array. `plt.hist` will not work with arrays that have more than two dimensions.

```
In [ ]: plt.hist(np.ravel(snow),range=(0,1),density=True)
```

Generate random sample from distribution

This is a **Bernoulli** distribution. `scipy.stats` includes a Bernoulli distribution function we can use for comparison.

The `p` shape parameter determines the fraction of 1's versus 0's in the distribution.

To generate a random sample of from this distribution, we use the `rvs` function. The `size` option indicates how many samples to generate.

```
In [ ]: p = 0.3                                # Shape parameter
        r = stats.bernoulli.rvs(p, size=100)    # Sample from the Bernoulli distribution 1
        00 times

        plt.hist(r,density=True,color='r')
```

Plot both distributions for comparison. We have to use the `plt.hist` function with `np.ravel` for the snow variable, which means we'll get warning messages.

```
In [ ]: p = 0.3
        r = stats.bernoulli.rvs(p, size=100)

        plt.hist([np.ravel(snow),r],range=(0,1),density=True,color=('b', 'r'))
```

Repeat the above cell a few times. What happens to the red bars?

In [ ]:
```
# Digression: if you want to get rid of the warning, you have to do two steps:

# convert 3D array to 1D array
snow_1d=np.ravel(snow)

# Make new array with just the land values
land_snow_frac=snow_1d[np.where(np.isfinite(snow_1d))]

# Then, use land_snow_frac in plt.hist.

# Unfortunately, need to do this whenever want to use new array for histogram.
```

Now, increase the number of samples until the distribution doesn't change much when you re-sample from the Bernoulli distribution.

In [ ]:
```
p = 0.3
## Your sample number here
r = stats.bernoulli.rvs(p, size=???)

plt.hist([np.ravel(snow),r],range=(0,1),density=True,color=('b', 'r'))
```

Once you have a sample number that results in a fairly consistent distribution, play around with the `p` parameter until the Bernoulli distribution approx. matches the data. (You can actually calculate this explicitly from the snow data.)

In [ ]:
```
## Your parameter here↔
```

Removing grid points that never have snow

Some grid points never have snow, so their snow fraction always = 0. These just "pile up" on the left-hand side of the plot. To better see how snow cover *changes* with time, we'll create a new array that just has grid points where there's snow at some point (during the entire record).

`np.any` is `True` if *any* value along a given `axis` (0=time, in this case) is `True` for the comparison. We use `where` , similarly to what we did to omit the ocean points, to only keep these grid point values, setting the rest to `nan` .

In [ ]:
```
new_snow = snow.where(np.any(snow > 0,axis=0))
```

In [ ]:
```
#plt.hist((np.ravel(snow),np.ravel(new_snow)),range=(0,1),density=False)

# Don't specify density=True here, since we want the count of values.
snow.plot.hist()
new_snow.plot.hist(color='red',alpha=0.5)
```

Averaging snow cover fraction over different time periods (Central Limit Theorem)

Now, we're going to average snow cover over increasing periods of time to see the Central Limit Theorem in action:

The distribution of the mean of a big enough sample of independent random variables is a **normal (Gaussian) distribution**.

We're starting from **binary** data, and we don't have enough data to actually get to a Gaussian distribution, but we can start to see it occurring.

Part of the reason we eliminated the grid points that never have snow in the previous step is to make the fractions start converging closer to the middle.

Average one month at a time (monthly mean data)

Start by average data from the same month together.

This will average 4 or 5 weeks together. What are possible values?

(Note that, since the behaviors of the different months (as well as the different grid cells) differ, these points don't really come from the same distribution. So we don't actually expect them to have a normal distribution.)

Remind ourselves what new_snow looks like:

In [ ]:
```
print(np.shape(new_snow))
```

In [ ]:
```
print(new_snow.time)
```

To convert from these weekly values, we `resample` into individual `M` onths. This behaves similarly to the `groupby` function. Then, we average these different groups over the `'time'` dimension. (*Note that this is not* *`.mean(dim='time')`* *as before. Unfortunately, this is another place xarray is inconsistent.*)

In [ ]:
```
monthly_ave_snow=new_snow.resample(time='M').mean('time')
print('Shape = ',np.shape(monthly_ave_snow))
print('')
print(monthly_ave_snow.time)
monthly_ave_snow.plot(bins=40,density=True)
```

Averaging over more time steps gives us more possible values.

In [ ]:
```
# Seasonal mean
# About 13 weeks per season
# Smaller bars are b/c some seasons have 14 weeks.

season_mean_snow = new_snow.resample(time='Q').mean('time')
print('Shape = ',np.shape(season_mean_snow))
print('')
print(season_mean_snow.time)
season_mean_snow.plot(bins=40,density=True)
```

Average one year at a time (annual-mean data)

In [ ]:
```
annual_mean_snow = new_snow.groupby('time.year').mean(dim='time')
print('Shape = ',np.shape(annual_mean_snow))
print('')
print(annual_mean_snow.year)
annual_mean_snow.plot(bins=40,density=True)
```

Time-mean

Time-mean snow cover, still calculated for each grid cell separately. This differs from the first time-mean average in that the points that never have snow are excluded.

(The grid point are, again, not expected to have the same distributions themselves, since amount of snow varies with location.)

In [ ]:
```
time_average_snow_new=new_snow.mean(dim='time')
time_average_snow_new.plot.hist(bins=40,density=True)
plt.show()
```

Global-mean

Rather than looking at the distribution of the time mean for each grid point, another way we can examine the data is through the distribution of the "global"-mean time series.

This is just a rough calculation, since it doesn't take into account the areas of the cells.

In [ ]:
```
global_average_snow_new=new_snow.mean(dim='cols').mean(dim='rows')
global_average_snow_new.plot.hist(bins=40,density=True)
plt.show()
```

In [ ]:
```
```