Plotting large datasets, Part 2: Making movies

Colin Leach, March 2020

We are trying to follow a simulation through time, so movie files may have an advantage over static images. The simplest approach is to make a series of 2D plots then link them in an mp4 file. The NASA movies are clearly much more sophisticated but we don't know how they were made.

1 Within matplotlib

There are various Animation classes in Matplotlib that can do this: `https://matplotlib.org/api/animation_api.html`

It's a while since I did this so I don't have a simple working example. Google "matplotlib animation" and it will find many blog posts that may be useful.

A personal opinion: this is great for generating simple movies on the fly within Jupyter, for immediate display in the browser. For more serious work I prefer the ffmpeg approach in the next section. You may very well disagree, which is fine.

2 Using ffmpeg

Matplotlib can export the plot to a PNG or JPEG file and it's easy to write a python loop to produce a bunch of these with names like plot_000.png, plot_001.png, etc for the various snaps.

The (HUGE) video editing package ffmpeg can stitch these together into an mp4 movie file. The syntax is something like this (pretty short by ffmpeg standards):

```
ffmpeg -r 10 -start_number 0 -s 1920x1080 -i plot_%03d.png -vcodec libx264 \
  -vf fps=25 -crf 25 -pix_fmt yuv420p my_movie.mp4
```

Steps:

- Install ffmpeg if it's not already on your system. Linux/Mac users can probably get it from a repository, Windows users (or anyone) can get it from `https://ffmpeg.org/download.html`. It works well on Windows.

- Make the plot files with names having **consecutive numbering** and and **zero-padded to a constant length**. So using a list like 000, 001, 002... is fine, 000, 005, 010... will fail (not consecutive) and 8, 9, 10... will fail (not same length). Can you guess I learned that the hard way?

- In Python, the file names are easy to generate with f-strings:

```
          fname = f'plot_{snap:03}.png'
```

The snap then is padded to 3 digits with leading zeros, and this matches the '%03d' in the ffmepg command (after the -i switch, for input). To use non-consecutive snaps, keep a separate counter variable:

```
for i, snap in enumerate(np.arange(0, 20, 5)):
        # get data for snap, then...
        fname = f'plot_{i:03}.png'
```

- The command may take a few attempts to get right. Maybe best assembled in a text editor rather than directly in the command shell. It's really all one line, but line continuation characters are your friend: backslash \ for Linux (and Mac?), carat ˆ for Windows.
- Set start_number for your first file. You can't specify last_number or step_by. It stops automatically when it fails to find the next highest number.
- Other options (-r, -vf, -crf) are in the ffmeg documentation (good luck!) and many StackExchange posts. I resorted to trial and error.

There are other gotchas to fall into. Obviously the x and y limits need to be hard-coded in matplotlib, otherwise the frames will jump about and make you cross-eyed.

Less obviously, the pixel height of every image must be an even number. Matplotlib doesn't think about that but I found a hack online which seems to work.

```
# The simple figsize format is OK for static images:
# fig = plt.figure(figsize=(20,9))

# If saved files are to be used for animations with ffmpeg, row
# count must be an even number. This hack ensures that.
fig = plt.figure()
DPI = fig.get_dpi() # dots per inch of your display
fig.set_size_inches(1200.0/float(DPI),610.0/float(DPI))
```

DPI is 72 on my machine (your result may be different). Make sure your pixel count (1200, 610 in this case) uses even numbers and hope for the best.