30 May 2013

Speeding up Matplotlib

For the record, Matplotlib is awesome! Its output looks amazing, it is extremely configurable and very easy to use. What more could you want?

Well... speed. If there is one thing I could criticize about Matplotlib, it is its relative slowness. To measure that, lets make a very simple line plot and draw some random numbers as quickly as possible:

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

fig, ax = plt.subplots()

tstart = time.time()
num_plots = 0
while time.time()-tstart < 1:
    ax.clear()
    ax.plot(np.random.randn(100))
    plt.pause(0.001)
    num_plots += 1
print(num_plots)
```

On my machine, I get about 11 plots per second. I am using `pause()` here to update the plot without blocking. The correct way to do this is to use `draw()` instead, but due to a bug in the Qt4Agg backend, you can't use it there. If you are not using the Qt4Agg backend, `draw()` is supposedly the correct choice.

For a single plot, ten plots per second is not terrible. But then, this is really the simplest case possible, so ten frames per second in the simplest case probably means bad things for not so simple cases.

One thing that really takes time here is creating all the axes and text labels over and over again. So let's not do that.

Instead of calling `clear()` and then `plot()`, thus effectively deleting everything about the plot, then re-creating it for every frame, we can keep an existing plot and only modify its data:

```
fig, ax = plt.subplots()
line, = ax.plot(np.random.randn(100))

tstart = time.time()
num_plots = 0
while time.time()-tstart < 1:
    line.set_ydata(np.random.randn(100))
    plt.pause(0.001)
    num_plots += 1
print(num_plots)
```

which yields about 26 plots per second. Not bad for a simple change like this. The downside is that the axes are not re-scaled any longer when the data changes. Thus, they won't change their limits based on the data any more.

Profiling this yields some interesting results:

```
ncalls tottime percall cumtime percall filename:lineno(function)
15     0.167   0.011   0.167   0.011   {built-in method sleep}
```

The one function that uses the biggest chunk of runtime is `sleep()`, of all things. Clearly, this is not what we want. Delving deeper into the profiler shows that this is indeed happening in the call do `pause()`. Then again, I *was* wondering if using *pause* really was a great idea for performance...

As it turns out, `pause()` internally calls `fig.canvas.draw()`, then `plt.show()`, then `fig.canvas.start_event_loop()`. The default implementation of `fig.canvas.start_event_loop()` then calls `fig.canvas.flush_events()`, then sleeps for the requested time. To add insult to injury, it even insists on sleeping at least one hundredth of a second, which actually explains the profiler output of 0.167 seconds of `sleep()` for 15 calls very well.

Putting this all together now yields:

```
fig, ax = plt.subplots()
line, = ax.plot(np.random.randn(100))

tstart = time.time()
num_plots = 0
while time.time()-tstart < 1:
    line.set_ydata(np.random.randn(100))
    fig.canvas.draw()
    fig.canvas.flush_events()
    num_plots += 1
print(num_plots)
```

which now plots about 40 frames per second. Note that the call to `show()` mentioned earlier can be omitted since the figure is already on screen. `flush_events()` just runs the Qt event loop, so there is probably nothing to optimize there.

The only thing left to optimize now is thus `fig.canvas.draw()`. What this really is doing is drawing all the artists contained in the `ax`. Those artists can be accessed using `ax.get_children()`. For a simple plot like this, the artists are:

- the background `ax.patch`
- the line, as returned from the `plot()` function
- the spines `ax.spines`
- the axes `ax.xaxis` and `ax.yaxis`

What we can do here is to selectively draw only the parts that are actually changing. That is, at least the background and the line. To only redraw these, the code now looks like this:

```
fig, ax = plt.subplots()
line, = ax.plot(np.random.randn(100))
plt.show(block=False)

tstart = time.time()
num_plots = 0
while time.time()-tstart < 5:
    line.set_ydata(np.random.randn(100))
    ax.draw_artist(ax.patch)
    ax.draw_artist(line)
    fig.canvas.update()
    fig.canvas.flush_events()
    num_plots += 1
print(num_plots/5)
```

Note that you have to add `fig.canvas.update()` to copy the newly rendered lines to the drawing backend.

This now plots about 500 frames per second. Five hundred times per second! Frankly, this is quite amazing!

Note that since we are only redrawing the background and the line, some detail in the axes will be overwritten. To also draw the spines, use `for spine in ax.spines.values(): ax.draw_artist(spine)`. To draw the axes, use `ax.draw_artist(ax.xaxis)` and `ax.draw_artist(ax.yaxis)`. If you draw all of them, you get roughly the same performance as `fig.canvas.draw()`. The axes in particular are quite expensive.

There is also a way of drawing the complete figure once and copying the complete but empty background, then reinstating that and only plotting a new line on top of it. This is equally fast as the code above without any visual artifacts, but breaks if you resize the plot.

In conclusion, I am quite impressed with the flexibility of Matplotlib. Matplotlib by default values quality over performance. But if you really need the performance at some point, it is flexible and hackable enough to let you tweak it to your hearts content. Really, an amazing piece of technology!

EDIT: As it turns out, `fig.canvas.blit(ax.bbox)` is a bad idea since it leaks memory like crazy. What you should use instead is `fig.canvas.update()`, which is equally fast but does not leak memory.

Tags: python

Other posts