animatplot Documentation

Release 0.2.2

Tyler Makaro

Aug 08, 2018

User Documentation

version 0.2.2

Source Code Github

animatplot is a library for producing interactive animated plots in python built on top of matplotlib.

Contents

CHAPTER 1

Installation

Using pip:

```
pip install animatplot
```

Warning: If matplotlib was installed with anaconda, please upgrade matplotlib to >= 2.2 with anaconda before installing animatplot with pip. Otherwise, pip may butcher your environment(s).

If you are using jupyter lab, then install jupyter-matplotlib.

CHAPTER 2

Tutorial

2.1 Getting Started

Animatplot is built on the concept of blocks. We'll start by animating a Line block.

First we need some imports.

Note: Interactivity is not available in the static docs. Run the code locally to get interactivity.

2.1.1 Basic Animation

```
In [1]: %matplotlib notebook
        import numpy as np
        import matplotlib.pyplot as plt
        import animatplot as amp
```

We will animate the function:

$y = \sin(2\pi(x + t))$ over the range $x = [0, 1]$, and $t = [0, 1]$

Let's generate the data:

```
In [2]: x = np.linspace(0, 1, 50)
        t = np.linspace(0, 1, 20)

        X, T = np.meshgrid(x, t)
        Y = np.sin(2*np.pi*(X+T))
```

In order to tell animatplot how to animate the data, we must pass it into a block. By default, the Line block will consider each of the rows in a 2D array to be a line at a different point in time.

We then pass a list of all our blocks into an Animation, and show the animation.

```
In [3]: block = amp.blocks.Line(X, Y)
        anim = amp.Animation([block])
```

```
        anim.save_gif('images/line1') # save animation for docs
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.1.2 Adding Interactivity

We'll use the same data to make a new animation with interactive controls.

```
In [4]: block = amp.blocks.Line(X, Y)
        anim = amp.Animation([block])

        anim.controls() # creates a timeline_slider and a play/pause toggle
        anim.save_gif('images/line2') # save animation for docs
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.1.3 Displaying the Time

The above animation didn't display the time properly because we didn't tell animatplot what the values of time are. Instead it displayed the frame number. We can simply pass our values of time into our call to `Animation`.

```
In [5]: block = amp.blocks.Line(X, Y)
        anim = amp.Animation([block], t) # pass in the time values

        anim.controls()
        anim.save_gif('images/line3') # save animation for docs
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.1.4 Controlling Time

Simply passing in the values of time into the call to `Animation` doesn't give us much control. Instead we use a `Timeline`.

```
In [6]: timeline = amp.Timeline(t, units='s', fps=20)
```

The `units` argument will set text to be displayed next to the time number.

The `fps` argument gives you control over how fast the animation will play.

```
In [7]: block = amp.blocks.Line(X, Y)
        anim = amp.Animation([block], timeline) # pass in the timeline instead

        anim.controls()
```

```
        anim.save_gif('images/line4') # save animation for docs
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.1.5 Built on Matplotlib

Since animatplot is build on matplotlib, we can use all of our matplotlib tools.

```
In [8]: block = amp.blocks.Line(X, Y, marker='.', linestyle='-', color='r')
        anim = amp.Animation([block], timeline)

        # standard matplotlib stuff
        plt.title('Sine Wave')
        plt.xlabel('x')
        plt.ylabel('y')

        anim.controls()
        anim.save_gif('images/line5') # save animation for docs
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.2 Using multiple blocks

Here we are going to use 2 different blocks in our animation.

First we need some imports:

```
In [1]: %matplotlib notebook
        import numpy as np
        import matplotlib.pyplot as plt
        import animatplot as amp
```

We are going to plot a pcolormesh and a line on 2 different axes.

Let's use: $z = \sin(x^2 + y^2 - t)$ for the pcolormesh, and a cross-section of $y = 0$: $z = \sin(x^2 - t)$ for the line.

First, we generate the data.

```
In [2]: x = np.linspace(-2, 2, 41)
        y = np.linspace(-2, 2, 41)
        t = np.linspace(0, 2*np.pi, 30)

        X, Y, T = np.meshgrid(x, y, t)

        pcolormesh_data = np.sin(X*X+Y*Y-T)
        line_data       = pcolormesh_data[20,:,:] # the slice where y=0
```

We need to be careful here. Our time axis is the last axis of our data, but animatplot assumes it is the first axis by default. Fortunately, we can use the `t_axis` argument.

We use the `axis` argument to attached the data to a specific subplot.

```
In [3]: # standard matplotlib stuff
        # create the different plotting axes
        fig, (ax1, ax2) = plt.subplots(1, 2)

        for ax in [ax1, ax2]:
            ax.set_aspect('equal')
            ax.set_xlabel('x')

        ax2.set_ylabel('y', labelpad=-5)
        ax1.set_ylabel('z')
        ax1.set_ylim([-1.1,1.1])

        fig.suptitle('Multiple blocks')
        ax1.set_title('Cross Section: $y=0$')
        ax2.set_title(r'$z=\sin(x^2+y^2-t)$')

        # animatplot stuff
        # now we make our blocks
        line_block       = amp.blocks.Line(X[0,:,:], line_data,
                                           axis=ax1, t_axis=1)
        pcolormesh_block = amp.blocks.Pcolormesh(X[:,:,0], Y[:,:,0], pcolormesh_data,
                                                 axis=ax2, t_axis=2, vmin=-1, vmax=1)
        plt.colorbar(pcolormesh_block.quad)
        timeline = amp.Timeline(t, fps=10)

        # now to contruct the animation
        anim = amp.Animation([pcolormesh_block, line_block], timeline)
        anim.controls()

        anim.save_gif('images/multiblock')
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

There is a lot going on here so lets break it down.

Firstly, the `standard matplotlib stuff` is creating, and labeling all of our axes for our subplot. This is exactly how one might do a static, non-animated plot.

When we make the Line block, we pass in the data for our lines as 2D arrays (`X[0,:,:]` and `line_data`). We attached that line to the first axis `axis=ax1`. We also specifify that the time axis is the last axis of the data `t_axis=1`.

When we make the Pcolormesh block, we pass in the x, y data as 2D arrays (`X[:,:,0]` and `Y[:,:,0]`), and the z data as a 3D array. We attached the pcolormesh to the second axis `axis=ax2`. We also specifify that the time axis is the last axis of the data `t_axis=2`.

Additional, we told the Pcolormesh blocks what the minimum and maximum values will be (`vmin=-1` and `vmax=1`), so that the colorscale will be proper. The keywords `vmin`, and `vmax` get passed to the underlaying called to matplotlib's pcolormesh.

`plt.colorbar` does not recognize the Pcolormesh block as a mappable, so we pass in a mappable from the block to get the colorbar to work. In the future, animatplot may have a wrapper around this.

The rest simply brings all of the blocks, and the timeline together into an animation.

2.3 Custimizing the Controls

Here we'll like how to manipulate the `timeline_slider` and the `toggle` button.

The interactive controls can be make using the `controls()` method of the animation class, as in the getting started tutorial, but this method is a wrapper around the `toggle` and `timeline_slider` methods.

First, we need from imports and data to animate.

```
In [1]: %matplotlib notebook
        import numpy as np
        import matplotlib.pyplot as plt
        import animatplot as amp
```

```
In [2]: x = np.linspace(0, 1, 50)
        t = np.linspace(0, 1, 20)

        X, T = np.meshgrid(x, t)
        Y = np.sin(2*np.pi*(X+T))
```

`Animation.`**`toggle`**(*axis=None*)

Creates a play/pause button to start/stop the animation

Parameters **`axis`** (*optional*) – A matplotlib axis to attach the button to.

`Animation.`**`timeline_slider`**(*axis=None*, *valfmt='%1.2f'*, *color=None*)

Creates a timeline slider.

Parameters

- **`axis`** (*optional*) – A matplotlib axis to attach the slider to
- **`valfmt`** (*str, optional*) – a format specifier used to print the time Defaults to '%1.2f'
- **`color`** – The color of the slider.

`Animation.`**`controls`**(*timeline_slider_args={}*, *toggle_args={}*)

Creates interactive controls for the animation

Creates both a play/pause button, and a time slider at once

Parameters

- **`timeline_slider_args`** (*Dict, optional*) – A dictionary of arguments to be passed to timeline_slider()
- **`toggle_args`** (*Dict, optional*) – A dictionary of argyments to be passed to toggle()

Now to make the animation

By specifying the `axis` parameter, we can change the position of either the toggle or the timeline_slider.

We use `color` to change the color of the slider, and `valfmt` to change how the time is displayed.

Let's create our block, then create the controls at the top of the animation.

```
In [3]: block = amp.blocks.Line(X, Y)

        plt.subplots_adjust(top=0.8) # squish the plot to make space for the controls
        slider_axis = plt.axes([.18, .89, .5, .03]) # the rect of the axis
        button_axis = plt.axes([.78, .87, .1, .07]) # x, y, width, height

        anim = amp.Animation([block])

        anim.toggle(button_axis)
```

```
        anim.timeline_slider(slider_axis, color='red', valfmt='%1.0f')
        # equivalent to:
        # anim.controls({'axis':slider_axis, 'color':'red', 'valfmt': '%1.0f'},
        #               {'axis':button_axis})

        anim.save_gif('images/controls')
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.4 Using Jupyter

In order to display interactive animations in jupyter notebook or lab, use one of the following line magics:

```
%matplotlib notebook  # notebook only
%matplotlib ipympl  # notebook or lab
%matplotlib widget  # notebook or lab (equivalent to ipympl)
```

CHAPTER 3

API

Animatplot is build on top of three main classes:

- Animation
- Block
- Timeline

A `Timeline` holds the information and logic to actually control the timing of all animations.

A `Block` represent any "thing" that is to be animated.

An `Animation` is a composition of a list of blocks and a timeline. This class builds the final animation.

3.1 Animation

Animation(blocks[, timeline, fig])	The foundation of all animations.

3.1.1 animatplot.Animation

class animatplot.**Animation**(*blocks*, *timeline=None*, *fig=None*)

The foundation of all animations.

Parameters

- **blocks** (*list of animatplot.animations.Block*) – A list of blocks to be animated
- **timeline** (*Timeline or 1D array, optional*) – If an array is passed in, it will be converted to a Timeline. If not given, a timeline will be created using the length of the first block.
- **fig** (*matplotlib figure, optional*) – The figure that the animation is to occur on

animation
a matplotlib animation returned from FuncAnimation

Methods

__init__	Initialize self.
controls	Creates interactive controls for the animation
save	Saves an animation
save_gif	Saves the animation to a gif
timeline_slider	Creates a timeline slider.
toggle	Creates a play/pause button to start/stop the animation

__init__(*blocks*, *timeline=None*, *fig=None*)
Initialize self. See help(type(self)) for accurate signature.

controls(*timeline_slider_args={}*, *toggle_args={}*)
Creates interactive controls for the animation

Creates both a play/pause button, and a time slider at once

Parameters

- **timeline_slider_args** (*Dict, optional*) – A dictionary of arguments to be passed to timeline_slider()
- **toggle_args** (*Dict, optional*) – A dictionary of argyments to be passed to toggle()

save(**args*, ***kwargs*)
Saves an animation

A wrapper around `matplotlib.animation.Animation.save()`

save_gif(*filename*)
Saves the animation to a gif

A convience function. Provided to let the user avoid dealing with writers.

Parameters **filename** (*str*) – the name of the file to be created without the file extension

timeline_slider(*axis=None*, *valfmt='%1.2f'*, *color=None*)
Creates a timeline slider.

Parameters

- **axis** (*optional*) – A matplotlib axis to attach the slider to
- **valfmt** (*str, optional*) – a format specifier used to print the time Defaults to '%1.2f'
- **color** – The color of the slider.

toggle(*axis=None*)
Creates a play/pause button to start/stop the animation

Parameters **axis** (*optional*) – A matplotlib axis to attach the button to.

3.2 Timeline

Timeline(t[, units, fps, log])	An object to contain and control all of the time

3.2.1 animatplot.Timeline

class animatplot.**Timeline**(*t*, *units=''*, *fps=10*, *log=False*)

An object to contain and control all of the time

Parameters

- **t** (*array_like*) – gets converted to a numpy array representing the time at each frame of the animation
- **units** (*str, optional*) – the units the time is measured in.
- **fps** (*float, optional*) – indicates the number of frames per second to play
- **log** (*bool, optional*) – Displays the time scale logarithmically (base 10). Defaults to False.

Methods

__init__	Initialize self.

__init__(*t*, *units=''*, *fps=10*, *log=False*)

Initialize self. See help(type(self)) for accurate signature.

3.3 blocks

Blocks handle the animation of different types of data. The following blocks are available in `animatplot.blocks`.

Block([axis, t_axis])	A base class for blocks
Line(x, y[, axis, t_axis])	Animates lines
Quiver(X, Y, U, V[, axis, t_axis])	A block for animated quiver plots
Pcolormesh(*args[, axis, t_axis])	Animates a pcolormesh
Imshow(images[, axis, t_axis])	Animates a series of images
Nuke(func, axis, length[, fargs])	For when the other blocks just won't do

3.3.1 animatplot.blocks.Block

class animatplot.blocks.**Block**(*axis=None*, *t_axis=None*)

A base class for blocks

Methods

__init__	Initialize self.

__init__(*axis=None*, *t_axis=None*)
Initialize self. See help(type(self)) for accurate signature.

3.3.2 animatplot.blocks.Line

class animatplot.blocks.**Line**(*x*, *y*, *axis=None*, *t_axis=0*, ***kwargs*)
Animates lines

Parameters

- **x** (*list of 1D numpy arrays or a 2D numpy array*) – The x data to be animated.
- **y** (*list of 1D numpy arrays or a 2D numpy array*) – The y data to be animated.
- **axis** (*matplotlib.axes.Axes, optional*) – The axis to attach the block to. Defaults to matplotlib.pyplot.gca()
- **t_axis** (*int, optional*) – The axis of the numpy array that represents time. Defaults to 0. No effect if x, y are lists of numpy arrays.

  The default is chosen to be consistent with: X, T = numpy.meshgrid(x, t)

Notes

This block accepts additional keyword arguments to be passed to matplotlib.axes.Axes.plot()

Methods

__init__	Initialize self.

__init__(*x*, *y*, *axis=None*, *t_axis=0*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

3.3.3 animatplot.blocks.Quiver

class animatplot.blocks.**Quiver**(*X*, *Y*, *U*, *V*, *axis=None*, *t_axis=0*, ***kwargs*)
A block for animated quiver plots

Parameters

- **X** (*1D or 2D numpy array*) – The x positions of the arrows. Cannot be animated.
- **Y** (*1D or 2D numpy array*) – The y positions of the arrows. Cannot be animated.
- **U** (*2D or 3D numpy array*) – The U displacement of the arrows. 1 dimension higher than the X, Y arrays.
- **V** (*2D or 3D numpy array*) – The V displcement of the arrows. 1 dimension higher than the X, Y arrays.

- **axis** (*matplotlib axis, optional*) – The axis to the block to
- **t_axis** (*int, optional*) – The axis of the array that represents time. Defaults to 0. No effect if U, V are lists.

Notes

This block accepts additional keyword arguments to be passed to `matplotlib.axes.Axes.quiver()`

Methods

__init__	Initialize self.

__init__(*X*, *Y*, *U*, *V*, *axis=None*, *t_axis=0*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

3.3.4 animatplot.blocks.Pcolormesh

class animatplot.blocks.**Pcolormesh**(**args*, *axis=None*, *t_axis=0*, ***kwargs*)
Animates a pcolormesh

Parameters

- **X** (*2D np.ndarray, optional*) –
- **Y** (*2D np.ndarray, optional*) –
- **C** (*list of 2D np.ndarray or a 3D np.ndarray*) –
- **axis** (*matplotlib axis, optional*) – an axis to attach the block to.
- **t_axis** (*int, optional*) – The axis of the array that represents time. Defaults to 0. No effect if C is a list.

Notes

All other keyword arguments get passed to `axis.pcolormesh` see `matplotlib.axes.Axes.pcolormesh()` for details.

Methods

__init__	Initialize self.

__init__(**args*, *axis=None*, *t_axis=0*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

3.3.5 animatplot.blocks.Imshow

class animatplot.blocks.**Imshow**(*images*, *axis=None*, *t_axis=0*, ***kwargs*)
Animates a series of images

Parameters

- **images** (*list of 2D/3D arrays, or a 3D or 4D array*) – matplotlib considers arrays of the shape (n,m), (n,m,3), and (n,m,4) to be images. Images is either a list of arrays of those shapes, or an array of shape (T,n,m), (T,n,m,3), or (T,n,m,4) where T is the length of the time axis (assuming `t_axis=0`).
- **axis** (*matplotlib axis, optional*) – The axis to attach the block to
- **t_axis** (*int, optional*) – The axis of the array that represents time. Defaults to 0. No effect if images is a list.

Notes

This block accepts additional keyword arguments to be passed to `matplotlib.axes.Axes.imshow()`

Methods

__init__	Initialize self.

__init__(*images*, *axis=None*, *t_axis=0*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

3.3.6 animatplot.blocks.Nuke

class animatplot.blocks.**Nuke**(*func*, *axis*, *length*, *fargs=[]*)
For when the other blocks just won't do

The block will clear the axis and redraw using a provided function on every frame. This block can be used with other blocks so long as other blocks are attached to a different axis.

Only use this block as a last resort. Using the block is like nuking an ant hill. Hence the name.

Parameters

- **func** (*callable*) – The first argument to this function must be an integer representing the frame number.
- **axis** (*a matplotlib axis, optional*) –
- **length** (*int*) – the number of frames to display
- **fargs** (*list, optional*) – a list of arguments to pass into func

Methods

__init__	Initialize self.

__init__(*func*, *axis*, *length*, *fargs=[]*)
Initialize self. See help(type(self)) for accurate signature.

CHAPTER 4

Example Gallery

Warning: For the purpose of these documents, animations are rendered as gifs and with a lower framerate and fewer frames to make them smaller.

If you run these animations locally, then they will be interactive.

Interactivity is available in Jupyter Notebook with following cell magic.

```
%matplotlib notebook
```

4.1 Nuke

Sometimes matplotlib just doesn't give us the tools we need to animate stuff. This block is a way to work around that.

Matplotlib.axes.Axes.quiver does not have a way to dynamically set the location of arrows, only the angle. In this example, we work around that.

```
In [1]: %matplotlib notebook
        import numpy as np
        import matplotlib.pyplot as plt
        import animatplot as amp
```

Lets first construct our data.

```
In [2]: E0 = np.array([1, 2])
        E0 = E0 / np.linalg.norm(E0)

        phi = np.array([0, np.pi/7])

        f = 3
        t = np.linspace(0,2*np.pi,100)

        ES = E0[:, np.newaxis]*np.exp(1j*(t+phi[:, np.newaxis])) # fancy array boardcasting
```

Now, we animate the data.

```
In [3]: fig, ax = plt.subplots()

        def animate(i):
            ax.set_title('Polarization')
            ax.set_aspect('equal')
            ax.set(xlim=(-1.2, 1.2), ylim=(-1.2, 1.2))

            E = E0*np.exp(1j*(f*t[i]+phi))

            xx = np.array([0,E[0].real,0])
            yy = np.array([0,0,0])
            uu = np.array([E[0].real,0,E[0].real])
            vv = np.array([0,E[1].real,E[1].real])

            plax = ax.plot(ES[0].real, ES.real[1])
            qax = ax.quiver(xx,yy,uu,vv,[0,55,200], scale_units='xy', scale=1.)

        animate(0) # initialise the plot with the animate function

        timeline = amp.Timeline(t, units='ns', fps=10)
        block = amp.blocks.Nuke(animate, axis=ax, length=len(timeline))
        anim = amp.Animation([block], timeline)

        anim.controls()
        anim.save_gif('nuke')
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

4.2 Imshow

```
In [1]: %matplotlib notebook
        import numpy as np
        import matplotlib.pyplot as plt
        import animatplot as amp
```

First we focus on creating the data. A ising model is used to make this data.

```
In [2]: # Define LxL matrix
        L = 55
        # Initialize as random spin
        M = 2*(np.random.rand(L,L)>.5)-1
        J = 1
        b = 2.5

        nPer = 100

        images = [M]
        for i in range(100):
            M = M.copy()
            for dm in range(nPer):
                jj = int(np.random.rand()*L - 1)
                kk = int(np.random.rand()*L - 1)
```

```
                dE = 2*J*(M[jj+1,kk] + M[jj-1,kk] + M[jj,kk+1] + M[jj,kk-1])*M[jj,kk]
                if dE <= 0:
                    M[jj,kk]*=-1
                else:
                    if(np.random.rand()<np.exp(-b*dE)):
                        M[jj,kk]*=-1
            images.append(M)
            M[:,-1] = M[:,0]
            M[-1,:] = M[0,:]
```

Now we plot it.

```
In [3]: block = amp.blocks.Imshow(images)
        anim = amp.Animation([block])

        anim.controls()
        anim.save_gif('ising')
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

4.3 Logarithmic Timescales

Simply pass the keyword argument `log=True` to the Timeline, to get logarithmic timescales.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import animatplot as amp


        x = np.linspace(0, 1, 20)
        t = np.logspace(0, 2, 30)

        X, T = np.meshgrid(x, t)
        Y = np.sin(X*np.pi)*np.log(T)

        timeline = amp.Timeline(t, log=True)
        block = amp.blocks.Line(X, Y)
        anim = amp.Animation([block], timeline)

        plt.xlim([0,1])
        plt.ylim([0,Y.max()+1])

        anim.controls()
        anim.save_gif('logtime')
        plt.show()
```

```
<Figure size 640x480 with 3 Axes>
```

4.4 Parametric

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from matplotlib.animation import PillowWriter
        import animatplot as aplt


        def psi(t):
            x = t
            y = np.sin(t)
            return x, y


        t = np.linspace(0, 2*np.pi, 25)
        x, y = psi(t)
        X, Y = aplt.util.parametric_line(x, y)

        timeline = aplt.Timeline(t, 's', 24)

        ax = plt.axes(xlim=[0, 7], ylim=[-1.1, 1.1])
        block1 = aplt.blocks.Line(X, Y, ax)
        # or equivalently
        # block1 = aplt.blocks.ParametricLine(x, y, ax)

        anim = aplt.Animation([block1], timeline)

        # Your standard matplotlib stuff
        plt.title('Parametric Line')
        plt.xlabel('x')
        plt.ylabel(r'y')

        # Create Interactive Elements
        anim.toggle()
        anim.timeline_slider()

        anim.save('parametric.gif', writer=PillowWriter(fps=5))
        plt.show()
```

```
<Figure size 640x480 with 3 Axes>
```

4.5 pcolormesh

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import animatplot as amp


        x = np.linspace(-2, 2, 50)
        y = np.linspace(-2, 2, 50)
        t = np.linspace(0, 2*np.pi, 40)

        X, Y, T = np.meshgrid(x, y, t)

        Z = np.sin(X*X+Y*Y-T)
```

```
        block = amp.blocks.Pcolormesh(X[:,:,0], Y[:,:,0], Z, t_axis=2, cmap='RdBu')
        plt.colorbar(block.quad)
        plt.gca().set_aspect('equal')

        anim = amp.Animation([block], amp.Timeline(t))

        anim.controls()

        anim.save_gif('pcolormesh')
        plt.show()
```

```
<Figure size 640x480 with 4 Axes>
```

4.6 Polarization

```
In [1]: %matplotlib notebook
        import numpy as np
        import matplotlib.pyplot as plt
        import animatplot as amp
```

Let's create the data.

```
In [2]: E0 = np.array([1, 2])
        E0 = E0 / np.linalg.norm(E0)

        phi = np.array([0, np.pi/7])

        f = 3
        t = np.linspace(0,2*np.pi,50)

        # The Electric Field
        E = E0[:, np.newaxis]*np.exp(1j*(t+phi[:, np.newaxis])) # fancy array boardcasting

        # Converting the Electric field into animatable arrows.
        X = np.zeros(3) # x location of the arrow tails
        Y = np.zeros(3) # y location of the arrow tails

        zeros = np.zeros_like(E[0,:]) # padding
        U = np.array([E[0,:], zeros, E[0,:]]).real
        V = np.array([zeros, E[1,:], E[1,:]]).real
```

Now to animate it.

```
In [3]: plt.plot(E[0].real, E.real[1])

        timeline = amp.Timeline(t, units='ns', fps=20)
        block = amp.blocks.Quiver(X, Y, U, V, t_axis=1, scale_units='xy', scale=1)
        anim = amp.Animation([block], timeline)

        block.ax.set_aspect('equal')
        block.ax.set_xlim([-1,1])
        block.ax.set_ylim([-1,1])

        anim.controls()
        anim.save_gif('polarization')
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

4.7 Quiver Plot

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from matplotlib.animation import PillowWriter
        import animatplot as aplt


        x = np.linspace(0, 2*np.pi, 10)
        y = np.linspace(0, 2*np.pi, 5)
        t = np.linspace(0, 4.9, 25)

        timeline = aplt.Timeline(t)

        X, Y, T = np.meshgrid(x, y, t)

        U = np.cos(X+T)
        V = np.sin(Y+T)

        ax = plt.axes(xlim=[-1, 7], ylim=[-1, 7])
        block1 = aplt.blocks.Quiver(X[:,:,0], Y[:,:,0], U, V, axis=ax, t_axis=2, units='inches', pivo
        anim = aplt.Animation([block1], timeline)

        anim.toggle()
        anim.timeline_slider()

        anim.save('quiver.gif', writer=PillowWriter(fps=10))
        plt.show()
```

```
<Figure size 640x480 with 3 Axes>
```

4.8 Square Well

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from matplotlib.animation import PillowWriter
        import animatplot as aplt


        def psi(x, t):
            return (2**-.5*np.exp(t*1j)*np.sin(np.pi*x)
                    + .5*np.exp(t*4j)*np.sin(2*np.pi*x)
                    + .5*np.exp(t*9j)*np.sin(3*np.pi*x))


        x = np.linspace(0, 1, 20)
        t = np.linspace(0, 10, 20)
```

```
        X, T = np.meshgrid(x, t)
        Y1 = psi(X, T).real
        Y2 = psi(X, T).imag

        timeline = aplt.Timeline(t, 's', 24)

        ax = plt.axes(xlim=[0, 1], ylim=[-2, 2])
        block1 = aplt.blocks.Line(X, Y1, ax)
        block2 = aplt.blocks.Line(X, Y2, ax)

        anim = aplt.Animation([block1, block2], timeline)

        # Your standard matplotlib stuff
        plt.title(r'Particle in a Box: $|\Psi\rangle = \frac{1}{\sqrt{2}}'
                  r'|E_1\rangle + \frac{1}{2}|E_2\rangle + \frac{1}{2}|E_3\rangle$',
                  y=1.03)
        plt.xlabel('position')
        plt.ylabel(r'$\Psi$')
        plt.legend(['Real', 'Imaginary'])

        anim.toggle()
        anim.timeline_slider()

        anim.save('sq_well.gif', writer=PillowWriter(fps=5))
        plt.show()
```

```
<Figure size 640x480 with 3 Axes>
```

CHAPTER 5

Developer Setup

5.1 Requirements

The following are required to build the docs.

```
sphinx>=1.5.1
ipykernel
nbsphinx
matplotlib>=2.2
numpy
```

5.2 Install

Clone and install the repository:

```
git clone https://github.com/t-makaro/animatplot.git
cd animatplot
pip install -e .
```

5.3 Testing

From the root animatplot directory simply run:

```
pytest
```

Warning: Tests are currently very limited. Please run examples to ensure everything works.

5.4 Linting

This project currently uses `pycodestyle` for linting.

CHAPTER 6

Changes to animatplot

6.1 0.2.2

- Fix .animations and .blocks subpackages not being distributed properly.

6.2 0.2.0

- Complete and total overhaul of animatplot using with the idea of `blocks` as a foundation
- Chuck all previous attempts to support python 2 in the dumpster

6.3 0.1.0.dev3

This is the original release.

Index

Symbols

A

B

C

I

L

N

P

Q

S

T