1 Prototype Implementation: Matplottoy

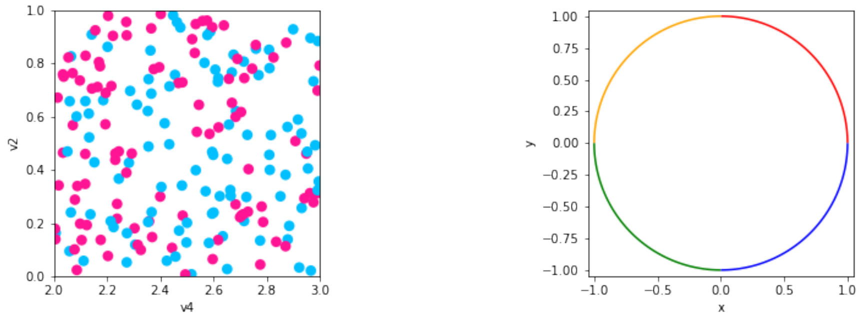

Figure 1: Scatter plot and line plot implemented using prototype artists and data models, building on Matplotlib rendering.

To prototype our model, we implemented the artist classes for the scatter and line plots shown in figure 1 because they differ in every attribute: different visual channels ν that composite to different marks Q with different continuities ξẆe make use of the Matplotlib figure and axes artists [3, 4] so that we can initially focus on the data to graphic transformations.

To generate the images in figure 1, we instantiate `fig, ax` artists that will contain the new `Point, Line` primitive objects we implemented based on our topology model.

```
fig, ax = plt.subplots()
artist = Point(data, transforms)
ax.add_artist(artist)
```

```
fig, ax = plt.subplots()
artist = Line(data, transforms)
ax.add_artist(artist)
```

We then add the A'=`Point` and A'=`Line` artists that construct the scatter and line graphics. The arguments to the artist are the data E=`data` that is to be plotted and the aesthetic configuration ν=`transforms`. We implement the artists as equivalence classes A' because it would be impractical to implement a new artist for every aesthetic setting, such as one artist for red lines and another for green.

1.1 Artist Class A'

The artist is the piece of the matplotlib architecture that constructs an internal representation of the graphic that the render then uses to draw the graphic. In the prototype artist, `transform` is a dictionary of the form `{parameter:(variable, encoder)}` where parameter is a component in P, variable is a component in F, and the ν encoders are passed in as functions or callable objects. The data bundle E is passed in as a `data` object. By binding data and transforms to A' inside `__init__`, the `draw` method is a fully specified artist A.

```
class ArtistClass(matplotlib.artist.Artist):
    def __init__(self, data, transforms, *args, **kwargs):
        # properties that are specific to the graphic but not the channels
        self.data = data
        self.transforms = transforms
        super().__init__(*args, **kwargs)

    def assemble(self, visual):
        # set the properties of the graphic

    def draw(self, renderer, *args, **kwargs):
        # returns K, indexed on fiber then key
        view = self.data.view()
        # visual channel encoding applied fiberwise
        visual = {p: encoder(view.get(f, None)) for
                     p, (f, encoder) in self.transforms.items()}
        self.assemble(visual)
        # pass configurations off to the renderer
        super().draw(renderer, *args, **kwargs)
```

The data is fetched in section τ via a `view` method on the data because the input to the artist is a section on E. The return view object has a `get` method to support querying for components that are not in F which we exploit to support parameters in the visual fiber that are not bound to fiber components in F. The ν functions are then applied to the data to generate the μ=`visual` input to Q. An explicit ξ is not implemented since that would mean copying a single μon $k$to all the associated s, as illustrated in figure **??**, and that is unnecessary overhead for these scatter and line plots. In $\hat{Q}$=`assemble` the artist generates instructions for the render by setting the attributes that are related to the graphic. These are the settings that would have to be serialized in order to recreate a static version of the graphic. Although `assemble` could be implemented outside the class such that it returns an object the artist could then parse to set attributes, the attributes are directly set here to reduce indirection. The ν functions could be evaluated in this function to avoid passing over Ktwice but are not done so here to demonstrate the seperability of ν and $\hat{Q}$The last step in the artist function is handing itself off to the renderer.

The `Point` artist builds on `collection` artists because collections are optimized to efficiently draw a sequence of primitive point and area marks. In this prototype, the scatter marker shape is fixed as a circle, and the only visual fiber components are x and y position, size, and the facecolor of the marker.

```python
class Point(mcollections.Collection):
    def __init__(self, data, transforms, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.data = data
        self.transforms = transforms

    def assemble(self, visual):
        # construct geometries of the circle marks in visual coordinates
        self._paths = [mpath.Path.circle(center=(x,y), radius=s)
                       for (x, y, s) in zip(visual['x'],visual['y'], visual['s'])]
        # set attributes of marks, these are vectorized
        # circles and facecolors are lists of the same size
        self.set_facecolors(visual['facecolors'])

    def draw(self, renderer, *args, **kwargs):
        # query data for a vertex table K
        view = self.data.view()
        visual = {p: encoder(view.get(f, None)) for
                      p, (f, encoder) in self.transforms.items()}
        self.assemble(visual)
        # call the renderer that will draw based on properties
        super().draw(renderer, *args, **kwargs)
```

The `view` method repackages the data as a fiber component indexed table of vertices, as described in section **??**; even though the `view` is fiber indexed, each vertex at an index $k$has corresponding values in section $\tau(k_i)$ such that all the data on one vertex maps to one marker. To ensure the integrity of the section, `view` must be atomic, meaning that the values cannot change after the method is called in draw until a new call in draw. This table is converted to a table of visual variables. It is then passed into `assemble`, where it is used to individually construct the vector path of each circular marker with center `(x,y)` and size `x` and set the colors of each circle. Since `view` returns a τ all these operations could be applied on a section on one k or a subset of K.

The only difference between the `Point` and `Line` objects is in the `view` and `assemble` function because line has different continuity from scatter and is represented by a different type of graphical mark.

```python
class Line(mcollections.LineCollection):
    def assemble(self, visual):
        #assemble line marks as set of segments
        segments = [np.vstack((vx, vy)).T for vx, vy
                        in zip(visual['x'], visual['y'])]
        self.set_segments(segments)
        self.set_color(visual['color'])

    def draw(self, renderer, *args, **kwargs):
        # query data source for edge table
        view = self.data.view()
        visual = {p: encoder(view.get(f, None)) for
                         p, (f, encoder) in self.transforms.items()}
        self.assemble(visual)
        super().draw(renderer, *args, **kwargs)
```

In the `Line` artist, `view` returns a table of edges. Each edge consists of (x,y) points sampled along the line defined by the edge and information such as the color of the edge. As with `Point`, the data is then converted into visual variables. In `assemble`, this visual representation is composed into a set of line segments and then the colors of each line segment are set. The colors are guaranteed to correspond to the correct segment because of the atomicity constraint on view.

1.2 Encoders ν

As mentioned above, the encoding dictionary is specified by the visual fiber component, the corresponding data fiber component, and the mapping function. The visual parameter serves as the dictionary key because the visual representation is constructed from the encoding applied to the data $\mu = \nu \circ \tau$. For the scatter plot, the mappings for the visual fiber components $P = (x, y, facecolors, s)$ are defined as

```python
cmap =  color.Categorical({'true':'deeppink', 'false':'deepskyblue'})
transforms = {'y': ('v1', lambda x: x),
              'x': ('v3', lambda x: x),
             'facecolors': ('v2', cmap),
             's':(None ,lambda _: itertools.repeat(.02))}
```

where the position *(x,y)* ν transformers are identity functions. The size s transformer is not acting on a component of F, instead it is a ν that returns a constant value. While size could be embedded inside the `assembly` function, it is added to the transformers to illustrate user configured visual parameters that could either be constant or mapped to a component in F. The identity and constant ν are explicitly implemented here to demonstrate their implicit role in the visual pipeline, but they could be optimized away. More complex encoders can be implemented as callable classes, such as

```
class Categorical:
    def __init__(self, mapping):
        # check that the conversion is to valid colors
        assert(mcolors.is_color_like(color) for color in mapping.values())
        self._mapping = mapping

    def __call__(self, value):
        # convert value to a color
        return [mcolors.to_rgba(self._mapping[v]) for v in values]
```

where `__init__` can validate that the output of the ν is a valid element of the P component the ν function is targeting. Creating a callable class also provides a simple way to swap out the specific (data, value) mapping without having to reimplement the validation or conversion logic.

A test for equivariance can be implemented trivially such that it is independent of data or encoder.

```
def test_nominal(values, encoder):
    m1 = list(zip(values, encoder(values)))
    random.shuffle(values)
    m2 = list(zip(values, encoder(values)))
    assert sorted(m1) == sorted(m2)
```

In this example, `is_nominal` checks for equivariance of permutation group actions by applying the encoder to a set of values, shuffling values, and checking that (value, encoding) pairs remain the same. This equivariance test can be implemented as part of the artist or encoder, but for minimal overhead, the equivariant it is implemented as part of the library tests.

1.3 Data E

The data input into the will often be a wrapper class around an existing data structure, but must meet the following criteria:

1. specify the fiber components F and connectivity K

2. have a that returns an atomic object that encapsulates τ
3. the view object must have that returns a fiber component

To support specifying the fiber bundle, we define an optional `FiberBundle` class

```
class FiberBundle:
    def __init__(self, base, fiber):
        """
        base:  {'tables': ['vertex', 'edge', 'face']}
        fiber: {'component name': {'type':, 'monoid':,  'range':}}
        """
        self.base = base
        self.fiber = fiber

    def is_section(self, section):
        """checks if a section is from a given fiber bundle:
        are values in F, are keys in K"""
```

that asks the user to specify how $K$is triangulated and the attributes of F. The `assembly` functions expect tables that match the continuity of the graphic; scatter expects a vertex table because it is discontinuous, line expects an edge table because it is 1D continuous. The fiber informs appropriate choice of ν therefore it is a dictionary of attributes of the fiber components. I've basically stripped this out of the artists above so should I just ditch this section?

To generate the scatter plot in figure 1, we fully specify a dataset with random keys and values in a section chosen at random form the corresponding fiber component. The fiberbundle `FB` is a class level attribute since all instances of `code`VertexSimplex come from the same fiberbundle.

```
class VertexSimplex: #maybe change name to something else
    """Fiberbundle is consistent across all sections
    """
    FB = FiberBundle({'tables': ['vertex']},
            {'v1': {'type': float,'monoid':'interval','range': [0,1]},
             'v2': {'type': str, 'monoid':'nominal', 'range':['true', 'false']},
             'v3': {'type': float, 'monoid':'interval', 'range':[2,3]}})

    def __init__(self, sid = 45, size=1000, max_key=10**10):
        # create random list of keys
    def tau(self, k):
        # e1 is sampled from F1, e2 from F2, etc...
```

```
        return (k, (e1, e2, e3, e4))

    def view(self):
        table = defaultdict(list)
        for k in self.keys:
            table['index'] = k
            # on each iteration, add one (name, value) pair per component
            for (name, value) in zip(self.FB.fiber.keys(), self.tau(k)[1]):
                table[name].append(value)
        return table
```

The view method returns a dictionary where the key is a fiber component name and the value is a list of values in the fiber component. The table is built one call to `tau` at a time, guaranteeing that all the fiber component values are over the same k. Table has a `get` method as it is a method on Python dictionaries. In contrast, the line in `EdgeSimplex` is defined as the functions `_color,_xy` on each edge.

```
class EdgeSimplex:
    # assign a class level FB attribute
    def __init__(self, num_edges=4, num_samples=1000):
        self.keys = range(num_edge) #edge id
        # distance along edge
        self.distances = np.linspace(0,1, num_samples)
        # half generlized representation of arcs on a circle
        self.angle_samples = np.linspace(0, 2*np.pi, len(self.keys)+1)

    @staticmethod
    def _color(edge):
        colors = ['red','orange', 'green','blue']
        return colors[edge%len(colors)]

    @staticmethod
    def _xy(edge, distances, start=0, end=2*np.pi):
        # start and end are parameterizations b/c really there is
        angles = (distances *(end-start)) + start
        return np.cos(angles), np.sin(angles)

    def tau(self, k): #will fix location on page on revision
        x, y = self._xy(k, self.distances,
                        self.angle_samples[k], self.angle_samples[k+1])
        color = self._color(k)
```

```python
        return (k, (x, y, color))

    def view(self, simplex):
        table = defaultdict(list)
        for k in self.keys:
            table['index'].append(k)
            # (name, value) pair, value is [x0, ..., xn] for x, y
            for (name, value) in zip(self.FB.fiber.keys(), self.tau(k, simplex)[1]):
                table[name].append(value)
```

Unlike scatter, the line **tau** method returns the functions on the edge evaluated on the interval [0,1]. By default these means each **tau** returns a list of 1000 x and y points and the associated color. As with scatter, **view** builds a table by calling **tau** for each $k\dot{U}$nlike scatter, the line table is a list where each item contains a list of points. This bookkeeping of which data is on an edge is used by the **assembly** functions to bind segments to their visual properties.

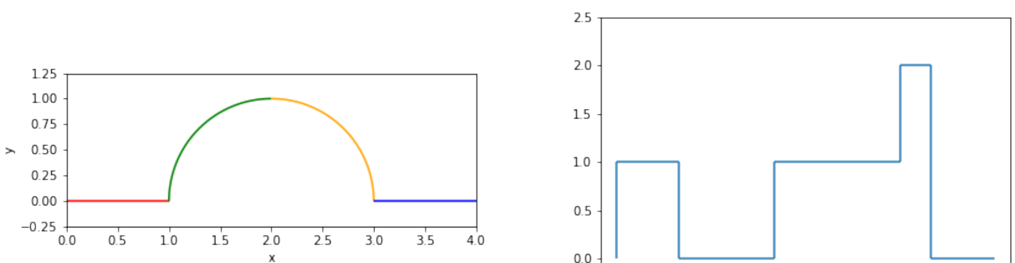

Figure 2: Continuous and discontinuous lines as defined by different data models, but generated with the same A'=**artist**

The graphics in figure 2 are made using the **Line** artist and the **Graphline** data source

```python
class GraphLine:
    def __init__(self, FB, edge_table, vertex_table, num_samples=1000, connect=False):
        #s set args as attributes and generate distance
        if connect: # test connectivity if edges are continuous
            assert edge_table.keys() == self.FB.F.keys()
            assert is_continuous(vertex_table)

    def tau(self, k, simplex='edge'):
        # evaluates functions defined in edge table
```

```
        return(k, (self.edges[c][k](self.distances) for c in self.FB.F.keys()))

    def view(self, simplex='edge'):
        """walk the edge_vertex table to return the edge function
        """
        table = defaultdict(list)
        #sort since intervals lie along number line and are ordered pair neighbors
        for (i, (start, end)) in sorted(zip(self.ids, self.vertices), key=lambda v:v[1][0]):
            table['index'].append(i)
            # same as view for line, returns nested list
            for (name, value) in zip(self.FB.F.keys(), self.tau(i, simplex)[1]):
                table[name].append(value)
        return table
```

where if told that the data is connected, the data source will check for that connectivity by constructing an adjacency matrix. The multicolored line is a connected graph of edges with each edge function evaluated on 1000 samples

```
simplex.GraphLine(FB, edge_table, vertex_table, connect=True)
```

while the stair chart is discontinuous and only needs to be evaluated at the edges of the interval

```
simplex.GraphLine(FB, edge_table, vertex_table, num_samples=2, connect=False)
```

such that one advantage of this model is it helps differentiate graphics that have different artists from graphics that have the same artist but make different assumptions about the source data.

1.4 Case Study: Penguins

For this case study, we use the Palmer Penguins dataset[1, 2] since it is multivariate and has a varying number of penguins. We use a version of the data packaged as a pandas dataframe[5, 6] since that is a very commonly used Python labled data structure. The wrapper is very thin since here there is explicitly only one section.

```
class DataFrameSection:
    def __init__(self, dataframe):
        self._tau = dataframe.iloc
        self._view = dataframe
    def view(self):
        return self._view
```

The pandas indexer is a key valued set of discrete vertices, so there is no need to repackage for triangulation. As with the previous examples, there is no need to implement an explicit `get` method since the `dataframe` object has a get method.

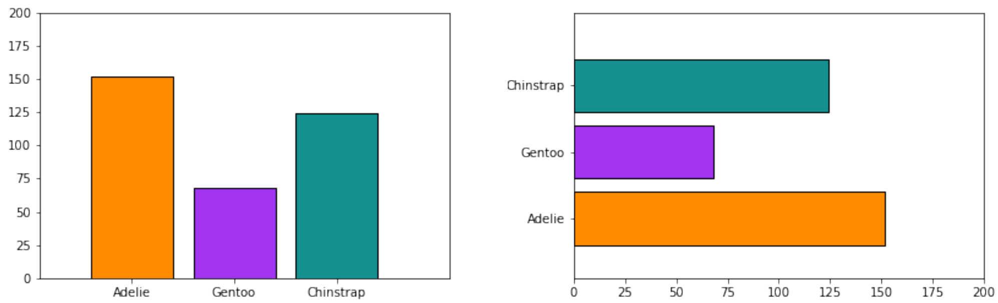

Figure 3: Frequency of Penguin types visualized as discrete bars.

The bar charts in figure 3 are generated with a `Bar` artist. The have the same required P components of (position, length). In of `Bar` an additional parameter is set, `orientation` which only applies holistically to the graphic and never to individual data parameters. Explicitly differentiate between parameters in V and ones that are only in $\hat{Q}$ is another way this model allows for cleaner separation of roles in the code.

```python
class Bar(mcollections.Collection):
    def __init__(self, data, transforms, *args, **kwargs):
        # parameter of the graphic
        self.orientation = kwargs.pop('orientation', 'v')

        super().__init__(*args, **kwargs)
        self.data = data
        self.transforms = transforms

    @staticmethod
    def _make_bars(orientation, position, width, floor, length):
        if orientation in {'vertical', 'v'}:
            xval, xoff, yval, yoff = position, width, floor, length
        elif orientation in {'horizontal', 'h'}:
            xval, xoff, yval, yoff = floor, length, position, width
        return [[(x, y), (x, y+yo), (x+xo, y+yo), (x+xo, y), (x, y)]
                 for (x, xo, y, yo) in zip(xval, xoff, yval, yoff)]


```

```python
    def assemble(self, visual):
        #set some defaults
        visual['width'] = visual.get('width', itertools.repeat(0.8))
        visual['floor'] = visual.get('floor', itertools.repeat(0))
        visual['facecolors'] = visual.get('facecolors', 'C0')
        #build bar glyphs based on graphic parameter
        verts = self._make_bars(self.orientation, visual['position'],
                     visual['width'], visual['floor'], visual['length'])
        self._paths = [mpath.Path(xy, closed=True) for xy in verts]
        self.set_edgecolors('k')
        self.set_facecolors(visual['facecolors'])

    def draw(self, renderer, *args, **kwargs):
        view = self.data.view()
        visual = utils.convert_transforms(view, self.transforms)
        self.assemble(visual)
        super().draw(renderer, *args, **kwargs)
        return
```

The `draw` method identical to the ones above, but here the visual transformations are factored out into a separate function. The `assemble` function sets some defaults, constructs bars, and sets their edge color to black. The `_make_bars` function is somewhat factored out because this is an operation that may be used by other bar making functions that may not be able to make use of bars assemble or draw.

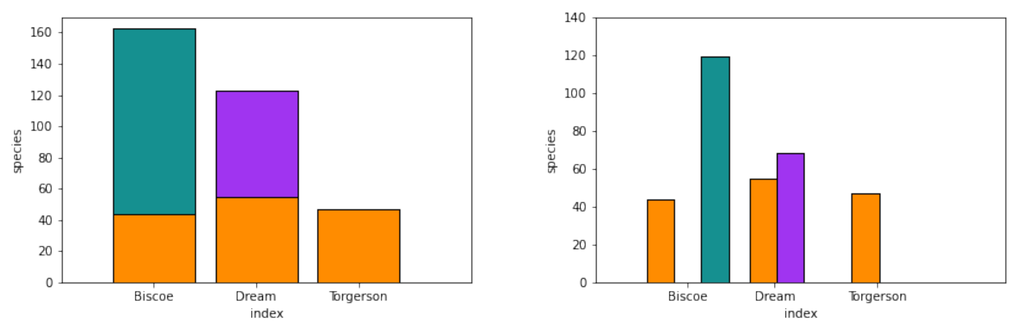

Figure 4: Penguin count disaggregated by island and species

For example, the `MultiBar` artist that makes figure 4 reuses `_make_bars` but does not reuse the assemble function because the composition of elements forces fundemental differences in glyph construction. As demonstrated in the `init`, the composite bar chart has orientation and whether it is stacked or not. While the stacked bar chart and the grouped bar chart could be seperate artists, as demonstrated they share so much overlapping code that it is far less redundant to implement them together. looking at the mess that is this code, I'm a) not convinced these should be combined b) no longer convinced this provides anything over just bar if it isn't rewritten to use bar more

```
class MultiBar(mcollections.Collection):
    def __init__(self, data, transforms, *args, **kwargs):
        #set the orientation of the graphic
        self.orientation = kwargs.pop('orientation', 'v')
        # set how the bar glyphs are put together to create the graphic
        self.stacked = kwargs.pop('stacked', False)
        # rest is same as other artist __init__s

        #this needs to be factored out but just want to finish now
        self.width = kwargs.pop('width', .8)

    def assemble(self, visual, view):
        (groups, gencoder) = self.transforms['length']
        ngroups = len(np.atleast_1d(groups))
        visual['floor'] = visual.get('floor', np.empty(len(view[groups[0]])))
        visual['facecolors'] = visual.get('facecolors', 'C0')
        # make equal width stacked columns
        if 'width' not in visual and self.stacked:
            visual['width'] = itertools.repeat(self.width)

        # make equal with groups
        if not self.stacked:
            visual['width'] = itertools.repeat(self.width/ngroups)
            offset = (np.arange(ngroups) /ngroups) * self.width
        else:
            offset = itertools.repeat(0)

        # make the bars and arrange them
        verts = []
        for group, off in zip(groups, offset):
            verts.extend(Bar._make_bars(self.orientation, visual['position'] + off,
                            visual['width'], visual['floor'], view[group]))
            if self.stacked: #add stacked bar to previous bar
                visual['floor'] += view[group]

        # convert lengths after all calculations are made and reorient if needed
        # here or in transform machinery?
```

```python
        if self.orientation in {'v', 'vertical'}:
            tverts = [[(x, gencoder(y)) for (x, y) in vert]
                            for vert in verts]
        elif self.orientation in {'h', 'horizontal'}:
            tverts = [[(gencoder(x), y) for (x, y) in vert]
                            for vert in verts]
        self._paths = [mpath.Path(xy, closed=True) for xy in tverts]
        #flatted columms of colors to match list of bars
        self.set_facecolor(list(itertools.chain.from_iterable(visual['facecolors'])))
        self.set_edgecolors('k')

    def draw(self, renderer, *args, **kwargs):
        view = self.data.view()
        #exclude converting the group visual length, special cased in assemble
        visual = utils.convert_transforms(view, self.transforms, exclude=['length'])
        # pass in view because nu is not distributable so may need to apply it
        # after visual assembly
        self.assemble(visual, view)
        super().draw(renderer, *args, **kwargs)
        return
```

In the `__draw__`, a utility function is used for conversions, but the length transforms are held until after assembly because the length is computed by adding the current length to the previous and many transforms are not distributable such that $\nu(x_0 + x_1 + x_2) = \nu(x_0) + \nu(x_1) + \nu(x_2)$. Inside `assemble`, the glyphs are either shifted vertically (`stacked`) or horizontally (`grouped`) such that the positions are recorded and added to with the next group. This function allows multiple columns to be mapped to a visual parameter, but it must be equal numbers of columns

```python
{'position': ('island', lambda x: {'Biscoe':0, 'Dream':1, 'Torgersen':2}[x]),
 'length':(['Adelie', 'Chinstrap', 'Gentoo'], lambda x: x),
 'facecolors': (['Adelie_s', 'Chinstrap_s', 'Gentoo_s'],
        color.Categorical({'Adelie':'#FF8C00',
                           'Gentoo':'#159090',
                           'Chinstrap':'#A034F0'}))}
```

such as in this example where for each column contributing to a segment of the bar there is a corresponding column of colors for this segment. The reason the multibar can work with such a trasnformer is because it is relying on the data model to do most of the bookkeeping of which values get mapped to which bars. This also yields a much simpler function call to the artist

```python
fig, ax = plt.subplots()
artist = bar.MultiBar(table, trans, orientation='h', stacked=True)
ax.add_artist(artist)
```

where `trans` is the same dictionary for both stacked and grouped version, as is the `DataFrameSection` object `table`. The only difference between the two versions is the `stacked` flag, and the only difference between figures 3 is the `orientation` argument. By decomposing the architecture into data, visual encoding, and assembly steps, we are able to build components that are more flexible and also more self contained than the existing code base.

This API may want to be redesigned such that there's a way to clearly couple the columns when doing multindex broadcasting