1 Prototype Implementation: Matplottoy

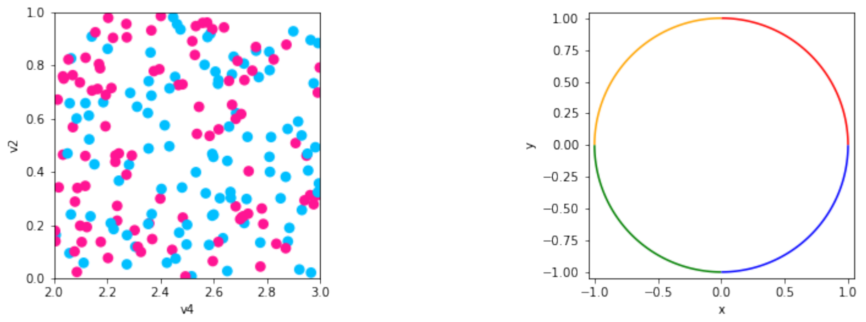


Figure 1: Scatter plot and line plot implemented using prototype artists and data models, building on Matplotlib rendering.

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 [1, 2] so that we can initially focus on the data to graphic transformations. We also exploit the Matplotlib transform stack to transform data coordinates into screen coordinates. To generate the images in figure 1, we instantiate artists that will contain the new primitive objects we implemented based on our topology model.

We then add the and artist that construct the scatter and line graphics. These artists are implemented as the equivalence class A' with the aesthetic configurations factored out into a dictionary that specifies the visual bundle VṪhe equivalence classes A' map well to Python classes since the functional aspects-ν, $\hat{Q}$, and ξ- are completely reusable in a consistent composition, while the visual values in V are what change between different artists belonging to the same class A'. The object is an abstraction of a data bundle E with a specified section τ. Implementing H and ρ are out of scope for this prototype because they are part of the rendering process. We also did not implement any form of ξ because the scatter, line, and bar plots prototyped here directly broadcast from k to s, unlike for example an image which may need to be rotated.

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, is a dictionary of the form 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 object. By binding data and transforms to A' inside , the method is a fully specified artist A.

The data is fetched in section τ via a method on the data because the input to the artist is a section on E. The method takes the attribute because it provides the region in graphic coordinates S that we can use to query back into data to select a subset as discussed in section **??**. The ν functions are then applied to the data to generate the visual section μ that here is the object . The conversion from data to visual space is simplified here to directly show that it is the encoding ν applied to the component. In the full implementation, we allow for fixed visual parameter, such as setting a constant color for all sections, by verifying that the named component is in F before accessing the data. If the data component name is not in F this is interpreted to mean this component is a thickening of Vthat could be pulled back to E via an inverse identity ν.

The components of the visual object, denoted by the Python unpacking convention are then passed into the function that is $\hat{Q}$. This assembly function is responsible for generating a representation such that it could be serialized to recreate a static version of the graphic. Although 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. This artist is not optimized because we prioritized demonstrating the separability of ν and $\hat{Q}$. The last step in the artist function is handing itself off to the renderer. The extra arguments in are artifacts of how these objects are currently implemnted in Matplotlib.

The artist builds on 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. We only show the function here because the are identical the prototype artist. The method repackages the data as a fiber component indexed table of vertices. Even though the is fiber indexed, each vertex at an index $k$has corresponding values in section $\tau(k_i)$. This means that all the data on one vertex maps to one glyph. To ensure the integrity of the section, must be atomic. This means that the values cannot change after the method is called in draw until a new call in draw. We put this constraint on the return of the method so that we do not risk race conditions.

This table is converted to a table of visual variables and is then passed into . In , the μ components are used to construct the vector path of each circular marker with center `(x,y)` and size `x` and set the colors of each circle. This is done via the object. As mentioned in sections **??** and **??**, this assembly function could as easily be implemented such that it was fed one $\tau(k)$ at a time.

The main difference between the and objects is in the function because line has different continuity from scatter and is represented by a different type of graphical mark.

In the artist, 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 , the data is then converted into visual variables. In , this visual representation is composed into a set of line segments, where each segement is the array generated by . 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.

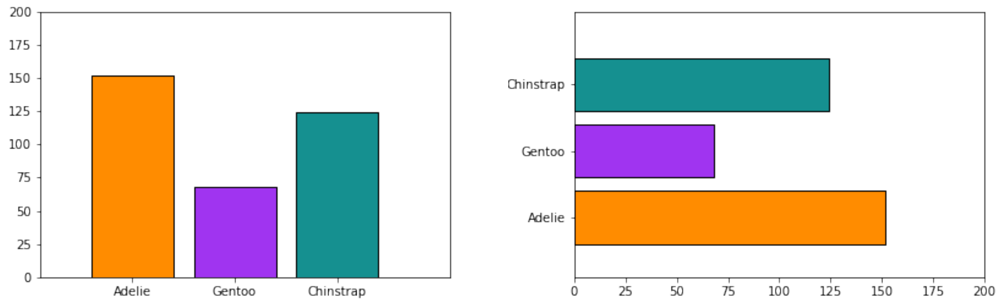


Figure 2: Frequency of Penguin types visualized as discrete bars.

The bar charts in figure 2 are generated with a artist. The artist has required visual parameters P of (position, length), and an additional parameter `orientation` which controls whether the bars are arranged vertically or horizontally. This parameter only applies holistically to the graphic and never to individual data parameters, and highlights how the model encourages explicit differentiation between parameters in V and graphic parameters applied directly to $\hat{Q}$.

The method here has a more complex unpacking of visual encodings to support passing in visual component data directly. This is vastly simplifies building composite objects as the alternative would be higher order functions that take as input the transforms passed in by the user. This construction supports a constant visual parameter, an identity transform where the value is the same in E and V, and setting the visual component directly. The function constructs bars and sets their face and edge colors. The function converts the input position and length to the coordinates of a rectangle of the given width. Defaults are provided for 'width' and 'floor' to make this function more reusable. Typically the defaults are used for the type of chart shown in figure 2, but these visual variables are often set when building composite versions of this chart type as discussed in section 1.4.

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 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 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 are somewhat optimized away in . More complex encoders can be implemented as callable classes, such as

where 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 but is currently factored out of the artist for clarity. In this example, 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.

1.3 Data E

The data input into the will often be a wrapper class around an existing data structure. This wrapper object must specify the fiber components F and connectivity K and have a method that returns an atomic object that encapsulates τ. The object returned by the view must be key valued pairs of where each section is a component as defined in equation **??**. To support specifying the fiber bundle, we define a data class[3]

that asks the user to specify how $K$is triangulated and the attributes of F. Python dataclasses are a good abstraction for the fiber bundle class because the class only stores data. The K is specified as tables because the 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.

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 is a class level attribute since all instances of come from the same fiberbundle. 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 the section method at a time, guaranteeing that all the fiber component values are over the same k. Table has a method as it is a method on Python dictionaries. In contrast, the line in is defined as the functions on each edge.

Unlike scatter, the line section method returns the functions on the edge evaluated on the interval [0,1]. By default these means each returns a list of 1000 x and y points and the associated color. As with scatter, builds a table by calling for each kUnlike 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 functions to bind segments to their visual properties.

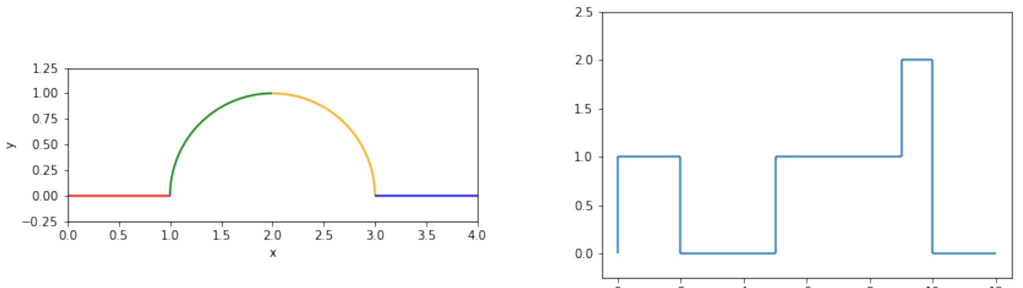
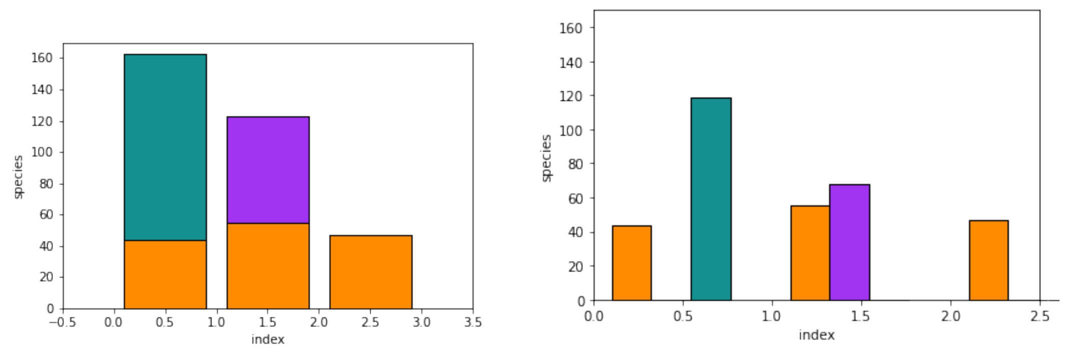


Figure 3: Continuous and discontinuous lines as defined by different data models, but generated with the same A'

The graphics in figure 3 are made using the artist and the data source 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 while the stair chart is discontinuous and only needs to be evaluated at the edges of the interval 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[4, 5] since it is multivariate and has a varying number of penguins. We use a version of the data packaged as a pandas dataframe[6] since that is a very commonly used Python labeled data structure. The wrapper is very thin because there is explicitly only one section. Since the aim for this wrapper is to be very generic, here the fiber is set by querying the dataframe for its metadata. The are a list of column names and the datatype of the values in each column; this is the minimal amount of information the model requires to verify constraints. The pandas indexer is a key valued set of discrete vertices, so there is no need to repackage for the data interface.



(a) (b)

Figure 4: Penguin count disaggregated by island and species

The stacked and grouped bar charts in figure 4 are both out of artists such that the difference between and is specific to the ways in which the are stitched together. These two artists have identical constructors and methods. As with , the orientation is set in the constructor. In both these artists, we separate the transforms applied to only one component and the case where the same transform is applied to multiple components such that V has multiple components that map to the same retinal variable.

Since all the visual transformation is passed through to , the method does not do any visual transformations. In the is used to adjust the for every subsequent bar chart since a stacked bar chart is bar chart area marks concatenated together in the length parameter. In contrast, does not even need the view, but instead keeps track of the relative position of each group of bars in the visual only variable .

Since the only difference between these two glyphs is in the composition of , they take in the exact same transform specification dictionaries. The dictionary dictates the position of the group, in this case by island the penguins are found on.

describes the group, and takes a list of dictionaries where each dictionary is the aesthetics of each group. That and are required parameters is enforced in the creation of the artist. These means that these two artists have identical function signatures

but differ in assembly $\hat{Q}$. 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. While very rough, this API demonstrates that the ideas presented in the math framework are implementable. For example, the function that maps most closely to A is functional, with state only being necessary for bookkeeping the many inputs that the function requires. In choosing a functional approach, if not implementation, we provide a framework for library developers to build reusable encoder ν assembly $\hat{Q}$and artists A. We argue that if these functions are built such that they are equivariant with respect to monoid actions and the graphic topology is a deformation retraction of the data topology, then the artist by definition will be a structure and property preserving map from data to graphic.