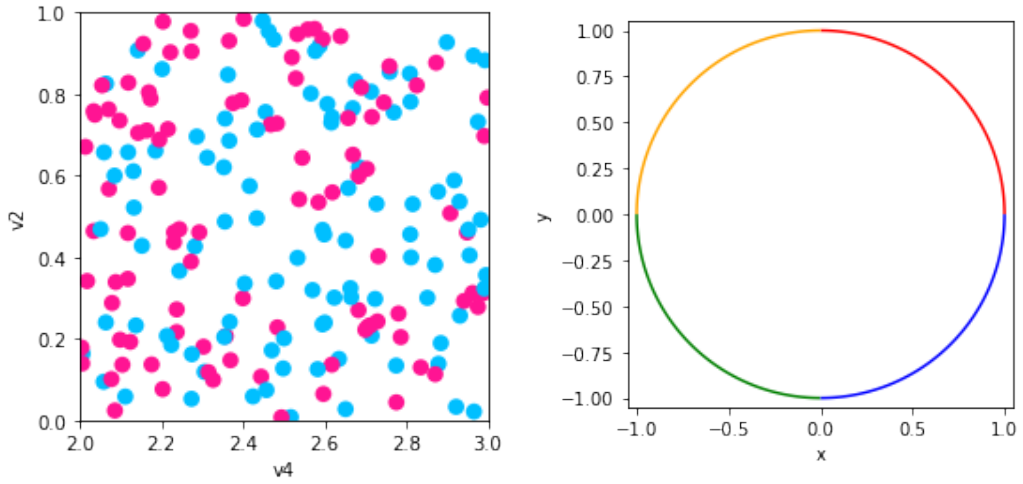
1 Prototype: Matplottoy

To evaluate the feasibility of the model described in **??**, we built prototypes of a `point`, `line`, and `bar` artist. We make use of the Matplotlib figure and axes artists [1, 2] so that we can initially focus on the data to graphic transformations and exploit the Matplotlib transform stack to convert data coordinates into screen coordinates. While the artist is specified in a fully functional manner in **??**, we implement the prototype in a heavily object oriented manner. We do so mostly to more easily manage function inputs, especially parameters that are passed through to methods that are structurally functional.

Building on the curr ent Matplotlib artists which construct an internal representation of the graphic, acts as an equivalence class artist A' as described in **??**. The visual bundle V is specified as the 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 as defined in **??**. 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 can be used to query back into data to select a subset as described in **??**. To ensure the integrity of the section, must be atomic, which means that the values cannot change after the method is called in draw until a new call to draw[3]. We put this constraint on the return of the method so that we do not risk race conditions.

The ν functions are then applied to the data, as describe in **??**, to generate the visual section μthat 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. The function that is $\hat{Q}$, as defined in **??**, is responsible for generating a representation such that it could be serialized to recreate a static version of the graphic. 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 implemented.

1.1 Scatter, Line, and Bar Artists



(a) (b)

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

The figure in Figure 1a is described by **??**. This is implemented via a object where the scatter marker shape is fixed as a circle, and the visual fiber components are x and y position and the facecolor and size 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. 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. To generate Figure 1b, the artist method 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 , described by **??**, this visual representation is composed into a set of line segments, where each segment 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 the view. The current implementation of line in Matplotlib does not have this functionality because it has no notion of an atomic view τ, instead the user keeps track of which values are input into a single call of $\hat{Q}$ by creating a loop where every color is a new call to the line plotting function.

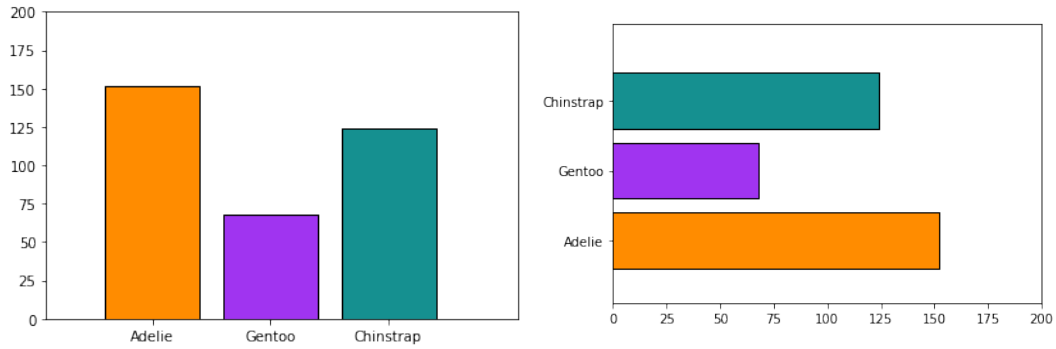


Figure 2: Frequency of Penguin types visualized as discrete bars.

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 $\hat{Q}$.

As with and , function constructs bars and sets their properties, 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 1.2.

1.2 Visual Encoders

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 is an identity ν, maps into P without corresponding τ to set a constant visual value, and is a custom ν implemented as a class for reusability. As described in **??**, a test for equivariance can be implemented trivially. It is currently factored out of the artist for clarity.

1.3 Data Model

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. To support specifying the fiber bundle, we define a data class[4]

that asks the user to specify the the properties of F and the K connectivity as either discrete vertices or continuous data along edges. To generate the scatter plot and the line plot, the distinction is in the method that is the section. The discrete method returns a record of discrete points, akin to a row in a table, while the line returns a sampling of points along an edge k. In both cases, the method packages the data into a data structure that

the artist can unpack via data component name, akin to a table with column names when K is 0 or 1 D.

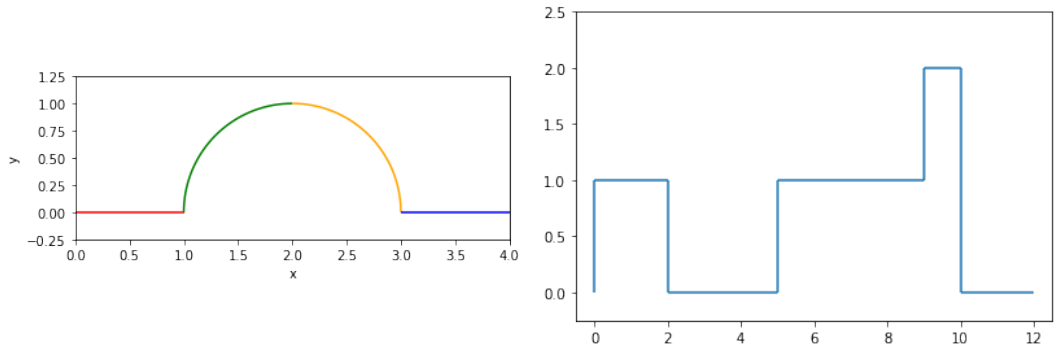
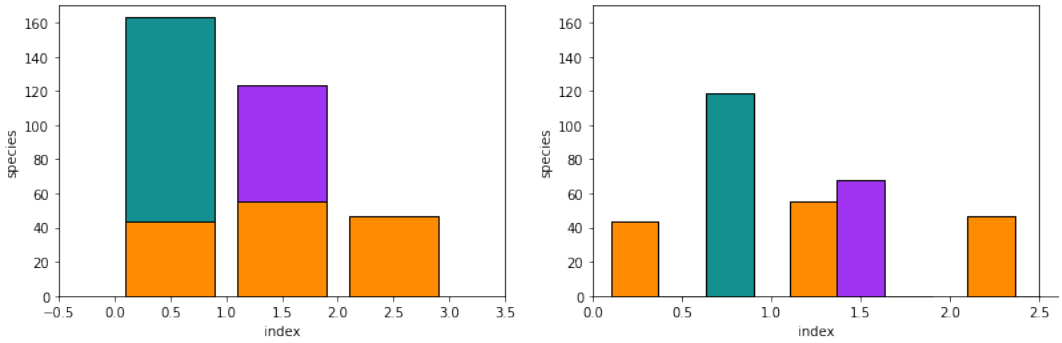


Figure 3: Continuous and discontinuous lines as defined via the same data model, and generated with the same A'

The graphics in figure 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 100 samples, which is an arbitrary choice made to display a smooth curve. The axes can also be used to choose an appropriate number of samples. In contrast, 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

Building on the artist in subsection 1.1, we implement grouped bar charts as these do not exist out of the box in the current version of Matplotlib.Instead, grouped bar charts are often achived via looping over an implementation of bar, which forces the user to keep track which values are mapped to a single visual element and how that is achieved. For this case study, we use the Palmer Penguins dataset[5, 6], packaged as a pandas dataframe[7] 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 composites 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 that are applied to only one component (column) from transforms applied to multiple components (columns). We encode the one component case as dictionary, and the multiple components case as . This convention allows us to, for example, map one column to position, but multiple to length.

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.