Topological Artist Model

Hannah Aizenman

A dissertation proposal submitted to
the Graduate Faculty in Computer Science in partial fulfillment of the
requirements for the degree of Doctor of Philosophy,
The City University of New York

Committee Members:
Dr. Michael Grossberg (Advisor), Dr. Robert Haralick, Dr. Lev Manovich,
Dr. Huy Vo, Dr. Marcus Hanwell

June 2021

Abstract

This work presents a functional model of the structure-preserving maps from data to visual representation to guide the development of visualization libraries. Our model, which we call the topological equivariant artist model (TEAM), provides a means to express the constraints of preserving the data continuity in the graphic and faithfully translating the properties of the data variables into visual variables. We formalize these transformations as actions on sections of topological fiber bundles, which are mathematical structures that allow us to encode continuity as a base space, variable properties as a fiber space, and data as binding maps, called sections, between the base and fiber spaces. This abstraction allows us to generalize to any type of data structure, rather than assuming, for example, that the data is a relational table, image, data cube, or network-graph. Moreover, we extend the fiber bundle abstraction to the graphic objects that the data is mapped to. By doing so, we can track the preservation of data continuity in terms of continuous maps from the base space of the data bundle to the base space of the graphic bundle. Equivariant maps on the fiber spaces preserve the structure of the variables; this structure can be represented in terms of monoid actions, which are a generalization of the mathematical structure of Stevens' theory of measurement scales. We briefly sketch that these transformations have an algebraic structure which lets us build complex components for visualization from simple ones. We demonstrate the utility of this model through case studies of a scatter plot, line plot, and image. To demonstrate the feasibility of the model, we implement a prototype of a scatter and line plot in the context of the Matplotlib Python visualization library. We propose that the functional architecture derived from a TEAM based design specification can provide a basis for a more consistent API and better modularity, extendability, scaling and support for concurrency.

Contents

1 Introduction

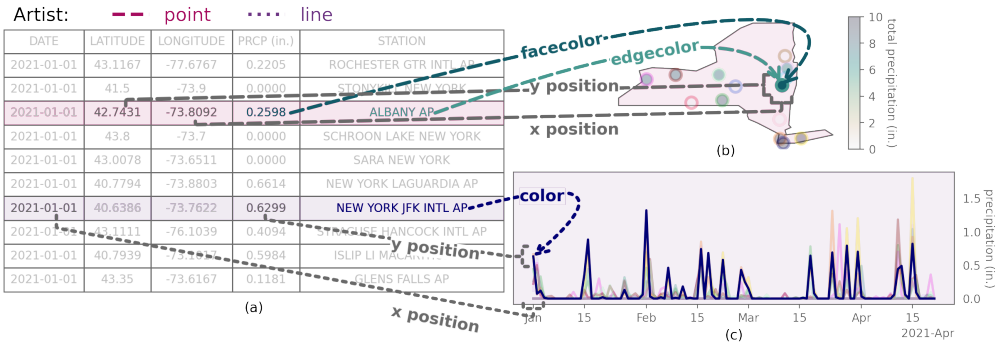


Figure 1: Building block visualization libraries implement independent reusable functions that map data to visual representations. For example, position encoding functions map the latitude and longitude values in the table (a) to locations on the map (b). These same encoding functions map dates and precipitation (a) to x and y positions in the timeseries plot (c). Color encoding functions map the precipitation (a) to the color of the point in the map (a) and color encoding functions map names of weather stations (a) to colors in both the map (a) and timeseries (b). Functions, which we call *artists*, compose these encoding functions into attributes of visual elements. The `point` artist transforms the output of the position, facecolor, and edgecolor maps into attributes of each point in the map (a), while the `line` artist transforms the output of the position and color maps into attributes of each line in the timeseries plot (c).

Visualizations, by definition, reflect something of the underlying structure and semantics[1] of the data, whether through direct mappings from data into visual elements or via figurative representations that have meaning due to their similarity in shape to external concepts [2]. We define visualization components to be structure preserving maps from data to visual representations. As illustrated in Figure 1, visualizations map weather station precipitation data (a) into graphical elements such as points (b) and lines (c) and map individual data components (columns) into components of the visualization such as position or color. For example, the gray position encoder function converts the latitude and longitude to x and y positions, and the color encoders map temperature and station name to colors. A `point` function composites these encodings into attributes of a point in the map (a). The `line` function composites encoders that convert station name to color and date and precipitation to x and y positions into a piece of a line in the timeseries plot (c). We identify the structure

these maps must preserve as the *continuity* of the data and the *equivariance* of the data and visual components.

We introduce a model of visualization components based on these *continuity* and *equivariance* constraints and use this model to develop a design specification. Our specification is targeted at building block libraries, as defined by Wongsuphasawat[3], because they are the visualization tools that provide independent functions that map components of data to visual representations. Just as the number of ways to arrange physical building blocks is solely constrained by the size and shape of the blocks, we propose that the only restriction on how building block library components can be composed are that the compositions preserve *continuity* and *equivariance*. Independent, modular, components are inherently functional[4], so we propose a functional architecture for our model. Doing so means the functional architecture can be evaluated for correctness, the resulting code is likely to be shorter and clearer, and the architecture is well suited to distributed, concurrent, and on demand tasks[5].

This work is strongly motivated by the needs of the Matplotlib[6, 7] visualization library. One of the most widely used visualization libraries in Python, new components and features have been added in an organic, sometimes hard to maintain, manner since 2002. In Matplotlib, every component carries its own implicit notion of how it believes the data is structured-for example if the data is a table, cube, image, or network - that is expressed in the API for that component. This leads to an inconsistent API for interfacing with the data, for example when updating streaming visualizations or constructing dashboards[8]. This entangling of data model with visual transform also yields inconsistencies in how visual component transforms, e.g. shape or color, are supported. We propose a redesign of the functions that convert data to graphics, named *Artists* in Matplotlib, in a manner that reliably enforces *continuity* and *equivariance* constraints. We evaluate our functional model by implementing new artists in Matplotlib that are specified via *equivariance* and *continuity* constraints. We then use the common data model introduced by the model to demonstrate how plotting functions can be consolidated in a way that makes clear whether the difference is in expected data structure, visual component encoding, or the resulting graphic.

2 Background

There are many formal models of visualization as structure preserving maps from data to visual representation, and many implementations of visualization libraries that preserve this structure in some manner. This work bridges the formalism and implementation in a functional manner with a topological approach at a building blocks library level. We propose a data structure agnostic model of the constraints visual transformations must satisfy such that they can be composed to produce *equivariant* and *continuity preserving* visual representations.

2.1 Structure

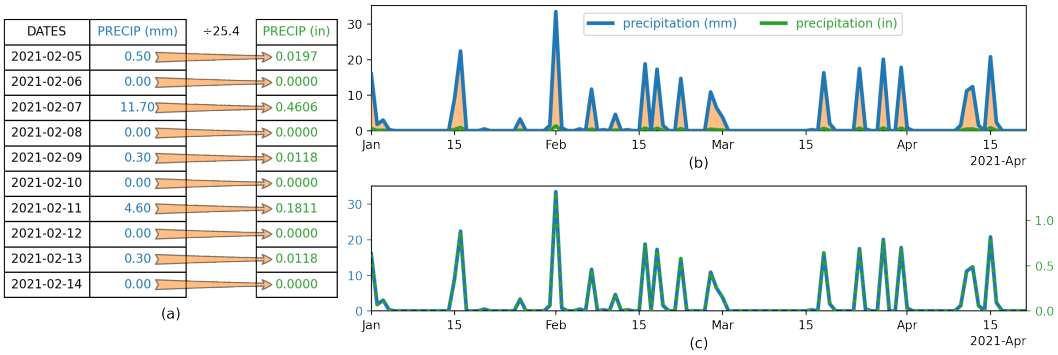


Figure 2: The table (a) lists precipitation in millimeters in blue. These values are converted to inches, in green, by dividing the millimeter values by 25.4. This conversion is a scaling action, represented by the orange arrows in (a). The *equivariant* action is the scaling in the timeseries (b), which is represented by the orange fill between the green and blue lines. In (c), the scaling is illustrated in the different millimeter and inches y axis labeling of data that appears identical since the distances were scaled by a constant factor that is now incorporated in the labeling.

The components of a visual representation were first codified by Bertin[9], who introduced a notion of structure preservation that we formally describe in terms of *equivariance* and *continuity*. Bertin proposes that there are categories of visual encodings-such as position, shape, color, and texture-that preserve the properties of the measurement type, quantitative or qualitative, of the encoded data. For example, in Figure 2, the blue precipitation data in millimeters in the table (a) is converted to inches. This scaling action is represented by

the green arrows that translate the precipitation into the scaled values in green. For this visualization to be equivariant, this same scaling factor must be present in the timeseries representation of the data (b). The precipitation in green is an *equivariant* scaling of the precipitation in blue, meaning that the y-values of the green line is $\frac{1}{25.4}$ that of the y-values of the blue precipitation timeseries. The orange fill in the timeseries (b) is the scaling equivalent to the arrows in the table (a). By definition, the distances in the millimeter data were scaled equally [10] when converted to inches, which means the shapes of the graphs are equivalent when plotted against y-axes that preserve relative distance (c). This visualization is also equivariant to the table (a), but this is indicated through labeling rather than the line plots. The idea of equivariance is formally defined as the mapping of a binary operator from the data domain to the visual domain in Mackinlay's *A Presentation Tool*(APT) model[11, 12]. The algebraic model of visualization proposed by Kindlmann and Scheidegger uses equivariance to refer generally to invertible binary transformations[13], which are mathematical groups [14]. Our model defines *equivariance* in terms of monoid actions, which are a more restrictive set than all binary operations and more general than groups. As with the algebraic model, our model also defines structure preservation as commutative mappings from data space to representation space to graphic space, but our model uses topology to explicitly include continuity.

Station	Precipitation (in.)
ALBANY AP	7.3819
BINGHAMTON	8.2874
BUFFALO	7.5157
GLENS FALLS AP	6.3071
ISLIP LI MACARTHUR AP	12.7874
NEW YORK JFK INTL AP	12.1260
NEW YORK LAGUARDIA AP	11.3582
ROCHESTER GTR INTL AP	7.5551
SYRACUSE HANCOCK INTL AP	7.1220

(a)

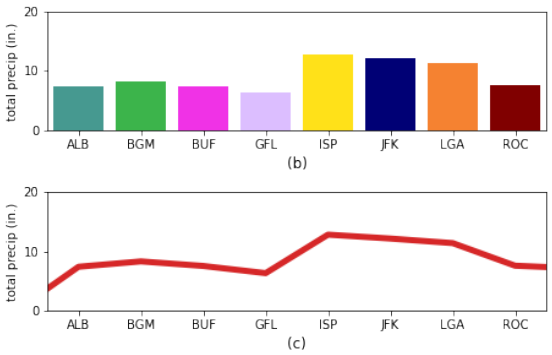


Figure 3: The bar chart (b) and line plot (c) are different visual encodings of the precipitation data in the table (a). The bar chart preserves the *continuity* of the stations by encoding the discrete station data as discrete bars (b). In contrast, the line plot (c) does not preserve this discrete *continuity* because it connects the total temperature at each station by drawing a line through each point. Doing so implies that the total temperatures are points along a 1D continuous line, whereas there is no connectivity between the stations or their corresponding rows in the table.

The notion that data is equivalent to visual representations when structure is preserved serves as the basis for visualization best practices. When *continuity* is preserved, as in the bar chart (b) in Figure 3, then the graphic has not introduced new structure into the data. In Figure 3, the line plot (c) does not preserve *continuity* because the line connecting the total precipitation of the stations implies that the stations are connected to each other along a 1D continuous line. Bertin asserted that *continuity* was preserved by choosing graphical marks that match the *continuity* of the data - for example discrete data is a point, 1D continuous is the line, and 2D data is the area mark. Informally, Norman's Naturalness Principal[15] states that a visualization is easier to understand when the properties of the visualization match the properties of the data. This principal is made more concrete in Tufte's concept of graphical integrity, which is that a visual representation of quantitative data must be directly proportional to the numerical quantities it represents (Lie Principal), must have the same number of visual dimensions as the data, and should be well labeled and contextualized, and not have any extraneous visual elements[16]. Expressivity, as defined by Mackinlay, is a measure how much of the mathematical structure in the data can be expressed in the visualizations; for example that ordered variables can be mapped into ordered visual elements. We generalize these different codifications of structure preserv-

ing encodings, proposing that a graphic is an equivalent representation of the data when *continuity* and *equivariance* are preserved.

Structure

continuity How elements in the dataset are connected to each other, e.g. discrete points, networked nodes, points on a continuous surface

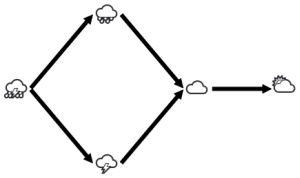
equivariance if an action is applied to the data or the graphic–e.g. a rotation, permutation, translation, or rescaling– there must be an equivalent action applied on the other side of the transformation.

2.2 Tools

DATE	LATITUDE	LONGITUDE	TEMP	NAME
2021-01-01	42.1	-77.1	27.86	GANG MILLS NEW YORK
2021-01-01	41.5	-73.9	29.48	STONYKILL NEW YORK
2021-01-01	43.8	-73.7	24.08	SCHROON LAKE NEW YORK
2021-01-01	43.35	-73.6167	25.52	GLENS FALLS AP
2021-01-01	43.1111	-76.1039	33.80	SYRACUSE HANCOCK INTL AP
2021-01-01	43.1167	-77.6767	31.64	ROCHESTER GTR INTL AP
2021-01-01	40.6386	-73.7622	35.96	NEW YORK JFK INTL AP
2021-01-01	42.1997	-75.985	28.76	BINGHAMTON
2021-01-01	40.7939	-73.1017	35.78	ISLIP LI MACARTHUR AP
2021-01-01	43.0078	-73.6511	27.68	SARA NEW YORK

(a)

(b)



(c)

Figure 4: Visualization libraries, especially ones tied to specific domains, tend to be architectured around a core data structure, such as tables Figure 4a, images Figure 4b, or networks Figure 4c.

Most information visualization software design patterns are tuned to specific data structures and domains, as categorized by Heer and Agrawala[17]. For users who generally work in one domain–such relational tables (Figure 4a), images (Figure 4b), or graphs (Figure 4c)– well defined data space (and corresponding visual space[18]) often yields a coherent user experience[19]. This coherant experience is often not extensible; developers who want to build new visualizations on top of these libraries must work around the existing assumptions, sometimes in ways that break the architecture model of the libraries. For example tools influenced by APT that assume that data is a relational table integrate computation into

the visualization pipeline. This is a wide array of tools, including Tableau[20–22] and the Grammar of Graphics[23] inspired ggplot[24], protovis[25], vega[26] and altair[27]. Since these libraries represent data as a table, and computations on tables are well defined[28], these libraries implement computation as part of the visualization process. This is also true of tools that support images, such as the biology oriented ImageJ[29] and Napari[30] or the digital humanities ImageJ macro ImagePlot[31]. While these tools often have some support for visualizing non image components of the data, the architecture is oriented towards building plugins into the existing system [32] where the image is the core data structure. Tools like Gephi[33], Graphviz[34], and Networkx[35] are used to visualize and manipulate graphs. As with tables and images, extending network libraries to work with other types of data either requires breaking the libraries internal model of how data is structured and what data transformations are allowed or implementing a model for other types of data structures alongside the network model. Our model aims to identify which computations are manipulations of the data and which are necessary for the visual encoding; for example data is often aggregated for a bar chart but it is not required, while a boxplot by definition requires computing distribution statistics[36]. Disentangling the computation from the visual transforms allows us to determine whether the visualization library should implement computations or if they should be implemented in data space.

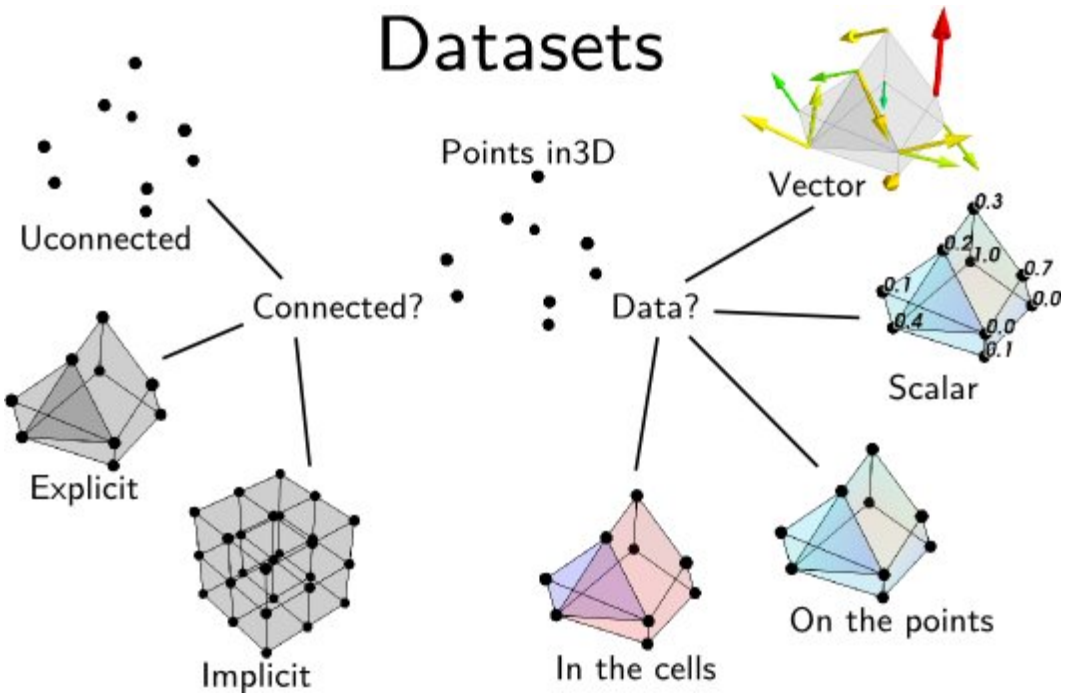


Figure 5: The *continuity* of the data is often used to classify data structures. For example, a relational database often consists of discrete unconnected points, while an image is an implicitly connected 2D grid. This image is from the Data Representation chapter of the MayaVi 4.7.2 documentation.[37]

Visualizations assume the structure of the input data, as described in Tory and Möller's taxonomy [38], which leads to many building block libraries implementing multiple models of data to support these different visualizations. For example, in Matplotlib, D3[39] VTK [40, 41] and MayaVi[42], every plot is defined in terms of the *continuity* of the data it expects as input. VTK has explicitly codified this in terms of *continuity* based data representations, as illustrated in figure 5. Downstream library developers impose some coherency by writing domain specific libraries with assumed data structures on top of the building block libraries–for example Seaborn[43] and Titan[44] assume a relational database, xarray[45] and ParaView[46] assume a data cube–but must work around the incoherencies in the building block libraries to do so. Our model navigates the tradeoff between coherency and extensibility by proposing functional composable well constrained visual components

that take as input a structure aware data abstraction general enough to provide a common interface for many different types of data continuities.

2.3 Data

Fiber bundles were proposed by Butler as a core data structure for visualization because they encode data *continuity* separately from the components of the dataset[47, 48]. Butler's model lacks a robust way of describing variables; therefore we encode a schema like description of the data in the fiber bundle using Spivak's topological description of data types [49, 50]. In this work, we refer to the points of the dataset as *records*, as defined by Spivak. Each *component* of the record is a single object, such as a precipitation measurement, a station name, or an image. We generalize *component* to mean all objects in the dataset of a given type, such as all precipitation values or station names or images. The way in which these records are connected is the *continuity* or more generally topology.

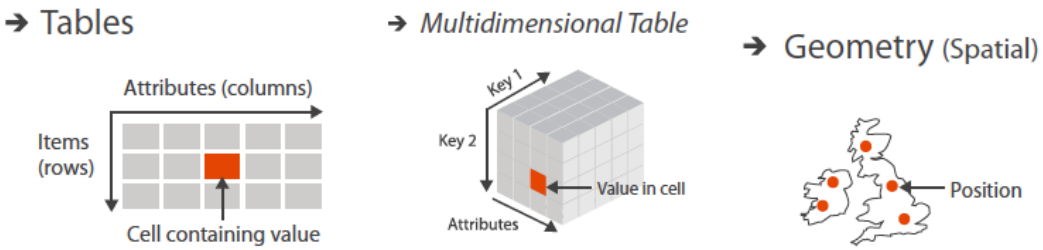


Figure 6: Values in a dataset have keys associated with them that describe where the value is in the dataset. These keys can be indexers or semantically meaningful;for example, in a table the keys are the variable name and the row ID. In the data cube, the keys is the row, column, and cell ID, and in the map the key is the position in the grid. Image is figure 2.8 in Munzner's Visualization Analysis and Design[51]

The *continuity* can be described in some datasets by components of the dataset. This is formalized by Munzner's notion of metadata as *keys* into the data structure that return associated *values*[52]. As shown in Figure 6, keys can be labeled indexes, such as the attribute name and row ID, or semantically significant physical entities such as locations on a map. In contrast to Munzner's model, in out model components may describe the keys but are never themselves the keys; instead we propose that keys are points on a topological

space encoding the continuity of the data. This allows the metadata to be altered without imposing new semantics on the underlying structure, for example by changing the coordinate systems or time resolution. This value agnostic model also supports encoding datasets where there may be multiple independent variables without having to assume any one variable is inducing the change , for example measures of plant growth given variations in water, sunlight, and time. For building block library developers, a non-semantic model of data continuity allows for the implementation of components that can traverse data structures without having to know the semantics of the data. Since these building block components are by design *equivariant* and *continuity preserving*, domain specific library developers in different domains that rely on the same continuity, for example 2D continuity, can use the same components to build tools that can make domain specific assumptions.

2.4 Contribution

In this work, we present a framework for understanding visualization as equivariant continuity preserving maps between topological spaces. Using this mathematical formalism, we develop an architecture specification developers can use to implement components in building block visualization libraries that domain specific library developers can carry through in the tools they build. Our work diverges from previous models of visualization and implementations of those models in that it contributes

1. formalization of the topology preserving relationship between data and graphic via continuous maps subsubsection 3.2.2
2. formalization of property preservation from data component to visual representation as equivariant maps subsubsection 3.3.2
3. functional oriented visualization architecture built on the mathematical model to demonstrate the utility of the model subsubsection 3.3.3
4. prototype of the architecture built on Matplotlib's infrastructure to demonstrate the feasibility of the model. subsection 4.1

We validate our model by using it to re-design artist and data access layer of Matplotlib, a general purpose visualization tool. We evaluate whether the redesign is successful by comparing it to existing implementation, recreating existing domain specific functionality with the new components, and by implementing components that provide new functionality in Matplotlib. While much of this functionality is currently possible in Matplotlib, a functional approach allows us to implement components in a more robust, modular, reliable way.

3 Topological Equivariant Artist Model

To guide the implementation of structure preserving visualization components, we develop a mathematical formalism of visualization that specifies how these components preserve *continuity* and *equivariance*. Inspired by the analogous classes in Matplotlib[7], we call the transformation from data space to graphic space that these building block components implement the *artist*.

$$\mathscr{A} : \mathscr{E} \to \mathscr{H} \tag{1}$$

The *artist* $\mathscr{A}$ is a map from data $\mathscr{E}$ to graphic $\mathscr{H}$ fiber bundles. To explain how the *artist* is a structure preserving map from data to graphic, we first model data (subsection 3.1) and graphics (subsection 3.2) as topological structures that encapsulate component types and continuity. We then discuss the funtional maps from graphic to data (subsubsection 3.2.2), data components to visual components (subsubsection 3.3.2), and visual components into graphic (subsubsection 3.3.3) that make up the artist.

3.1 Data Space E

We use fiber bundles as the data model because they are inclusive enough to express all the types of structures of data described in subsection 2.2. A fiber bundle is a tuple (E, K, π, F) defined by the projection map π

$$F \hookrightarrow E \xrightarrow{\pi} K \tag{2}$$

that binds the components of the data in F to the continuity of the data encoded in K. Our use of fiber bundles builds on Butler's work proposing that fiber bundles should be the common data abstraction for visualization data[47, 48]. The fiber bundle models the properties of data component types F (subsubsection 3.1.1), the continuity of records K (subsubsection 3.1.3), the collections of records (subsubsection 3.1.4), and the space E of all possible datasets with these components and continuity. By definition fiber bundles are locally trivial[53, 54], meaning that over a localized neighborhood U the total space is the cartesian product $K \times F$.

3.1.1 Variables in Fiber Space F

To formalize the structure of the data components, we use Spivak's description of the schema [50] as a fiber bundle to bind the components of the fiber to variable names and data types. Spivak constructs a set $\mathbb{U}$ that is the disjoint union of all possible objects of types $\{T_0, \ldots, T_m\} \in \mathbf{DT}$, where $\mathbf{DT}$ are the data types of the variables in the dataset. He then defines the single variable set $\mathbb{U}_\sigma$

$$\begin{array}{ccc} \mathbb{U}_\sigma & \longrightarrow & \mathbb{U} \\ {\scriptstyle \pi_\sigma}\downarrow & & \downarrow{\scriptstyle \pi} \\ C & \xrightarrow[\sigma]{} & \mathbf{DT} \end{array} \tag{3}$$

which is $\mathbb{U}$ restricted to objects of type T bound to variable name c. The $\mathbb{U}_\sigma$ lookup is by name to specify that every component is distinct, since multiple components can have the same type T. Given σ, the fiber for a one variable dataset is

$$F = \mathbb{U}_{\sigma(c)} = \mathbb{U}_T \tag{4}$$

where σ is the schema that binds a variable name c to its datatype T. A dataset with multiple components has a fiber that is the cartesian cross product of $\mathbb{U}_\sigma$ applied to all the columns:

$$F = \mathbb{U}_{\sigma(c_1)} \times \ldots \mathbb{U}_{\sigma(c_i)} \ldots \times \mathbb{U}_{\sigma(c_n)} \tag{5}$$

which can also be written as

$$F = F_0 \times \ldots \times F_i \times \ldots \times F_n \tag{6}$$

which allows us to decouple F into components $F_i = \mathbb{U}_{\sigma(c_i)}$.

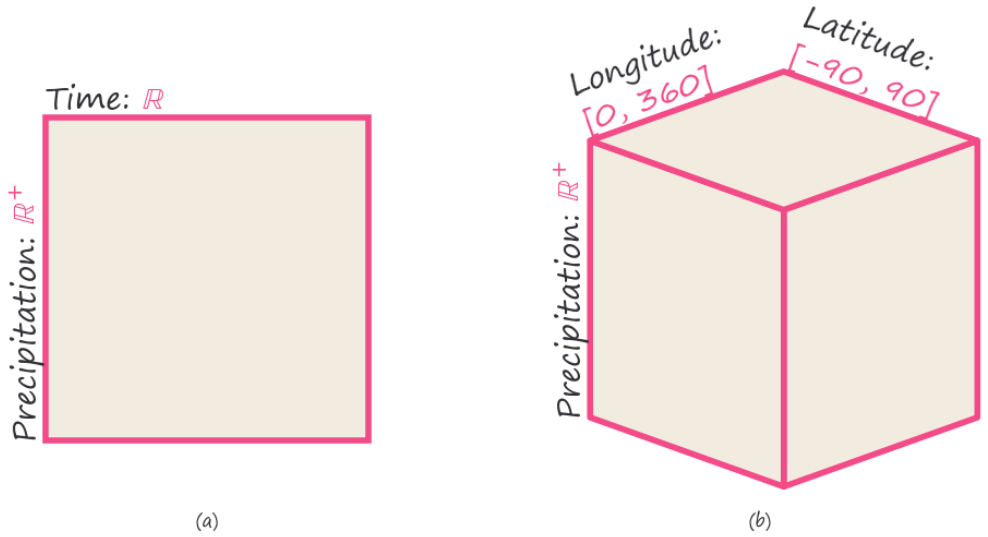


Figure 7: The fiber space is the cartesian product of the components. The 2D fiber $F = \mathbb{R} \times \mathbb{R}^+$ (a) encodes the properties of *time* and *precipitation* components. One dimension of the fiber encodes the range of possible values for the time component of the dataset, which is a subset of the $\mathbb{R}$, while the other dimension encodes the range of possible values $\mathbb{R}^+$ for the precipitation component. This means the fiber is the set of points *(precipitation, time)* that are all the combinations of *precipitation* $\times$ *time*. The 3D fiber (b) encodes points at all possible combinations of *precipitation*, *latitude*, and *longitude*.

For example, the records in the 2D fiber (a) in Figure 7 are a pair of *times* and *precipitation* measurements taken at those times. Time is a positive number of type `datetime` which can be resolved to $\mathbb{U}_{\texttt{datetime}} = \mathbb{R}$. Precipitation values are real positive numbers $\mathbb{U}_{\texttt{float}} = \mathbb{R}^+$. The fiber is

$$F = \mathbb{R} \times \mathbb{R}^+$$

where the first component F_0 is the set of values specified by ($c = time$, $T = \texttt{datetime}$, $\mathbb{U}_\sigma = \mathbb{R}$) and F_1 is specified by ($c = precipitation$, $T = \texttt{float}$, $\mathbb{U}_\sigma = \mathbb{R}$) and is the set of values

$\mathbb{U}_\sigma = \mathbb{R}$. In the 3D fiber (b) in Figure 7, time is replaced with location. This location variable is of type `point` and has two components *latitude* and *longitude* $\{(lat, lon) \in \mathbb{R}^2 \mid -90 \leq lat \leq 90,\ 0 \leq lon \leq 360\}$. The fiber for this dataset is

$$F = \mathbb{R} \times [0, 360] \times [-90, 90]$$

with components ($c = precipitation$, $T =$ `float`, $\mathbb{U}_\sigma = \mathbb{R}$), ($c = latitude$, $T =$ `float`, $\mathbb{U}_\sigma = [0, 360]$), and ($c = longitude$, $T =$ `float`, $\mathbb{U}_\sigma = [-90, 90]$. By adapting Spivak's framework, our model has a consistent way to describe the components of the data, no matter their complexity.

3.1.2 Measurement Scales: Monoid Actions

Implementing expressive visual encodings requires formally describing the structure on the components of the fiber. We conjecture that the structure can be formally defined as the actions of a monoid on the component. We can also discuss complex structures since a core property of monoids is composability [55]. In doing so, we can specify the properties of the component that must be preserved in a graphic representation.

A monoid [56] M is a set with a binary operation $* : M \times M \to M$ that satisfies the axioms:

$$\textbf{associativity} \text{ for all} a, b, c \in M\ (a \bullet b) \bullet c = a \bullet (b \bullet c)$$

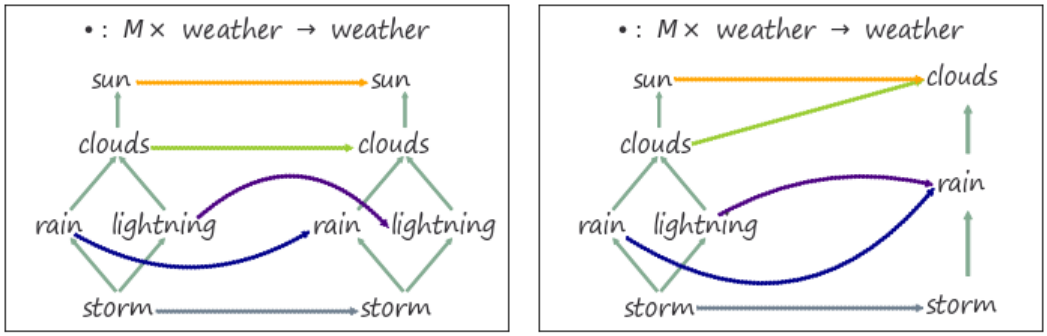
$$\textbf{identity} \text{ for all } a \in M,\ e \bullet a = a$$

As defined on a component of F, a left monoid action [**nlab:action**, 57] of M_i is a set F_i with an action $\bullet : M \times F_i \to F_i$ with the properties:

$$\textbf{associativity} \text{ for all } f, g \in M_i \text{ and } x \in F_i,\ f \bullet (g \bullet x) = (f * g) \bullet x$$

$$\textbf{identity} \text{ for all } x \in F_i, e \in M_i,\ e \bullet x = x$$

We conjecture that given these properties, if there is a partial ordering on both sides of the action then the action is equivariant and monotonic.



(a) (b)

Figure 8: The action • in Figure 8a is the arrows from the partial order diagram of weather states on the left to the diagram of weather states on the right. The action maps the weather states to themselves, thereby preserving the ordering defined by the monoid $*$ on both sides of the action. We conjecture that the action in Figure 8b is monotonic when the partial ordering of the weather states is the same as the partial order of the elements they are mapped to. Given $sun \geq clouds \geq rain$, $lightining$ on the right, we conjecture that the action sun, $clouds \rightarrow clouds$, and $rain$, $lightining \rightarrow rain$ is structure preserving such that the relative ordering of elements is the same as the elements they are mapped to.

One example of monoids are partial orderings on a set, such as seen in Figure 8. Each hasse diagram of the set of weather states describes an ordering on the set; the arrow goes from the lesser value to the greater one. For example, $storm \leq rain$. In Figure 8, the action • maps the elements of a set of weather states into itself by mapping them into other elements of the weather states. The action in Figure 8a, represented as the arrows between the hasse diagrams of the weather states, maps the weather states to themselves; therefore the ordering of the weather states is identical on both sides of the action. The action • in Figure 8b is a monotone map[58]

$$if\ a \leq b\ then\ \bullet(a) \leq \bullet(b) \mid a, b \in F_i$$

where the structure the action preserves is the relative, rather than exact, ordering. Since groups are monoids with invertible operations, this definition of structure is broad enough to include the Steven's measurement scales[59, 60].

As with the fiber F the total monoid space M is the cartesian product

$$M = M_0 \times \ldots \times M_i \times \ldots \times \ldots M_n \tag{7}$$

of each monoid M_i on F_i. The monoid is added to the specification of the fiber $(c_i, T_i, \mathbb{U}_\sigma M_i)$

3.1.3 Continuity of the Data K

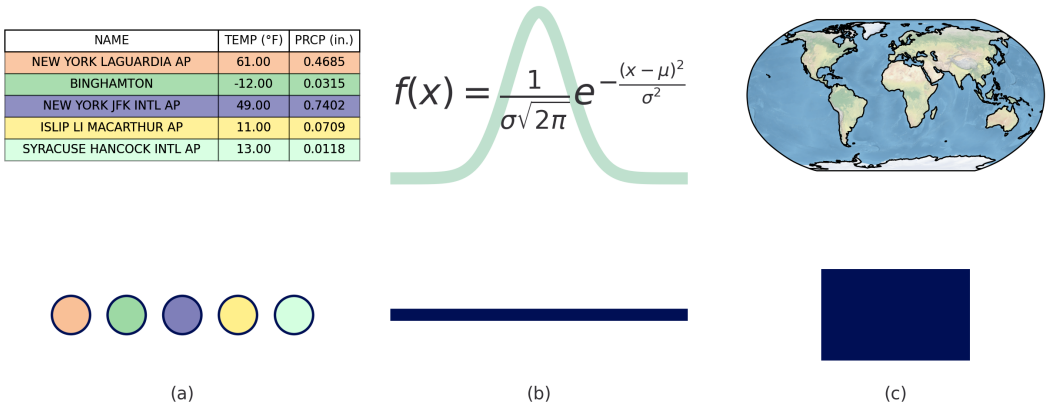


Figure 9: The topological base space K encodes the continuity of the data space. The table of discrete weather station records has discrete continuity such that each record maps to a single point (a). A gaussian has a value at all points along the interval x is sampled from and therefore has a 1D continuity (b). The globe has a value at all points (latitude, longitude) on the globe and therefore has 2D continuity (c).

The base space K provides a way to explicitly encode the continuity of the data, as described in subsection 2.3. This explicit topology is a concise way of distinguishing between visualizations that appear identical but assume different continuity, for example heatmaps and images. The base space K acts as an indexing space, as emphasized by Butler[47, 48], to express how the records in E are connected to each other. As shown in Figure 9, K can have any number of dimensions and can be continuous or discrete. Formally K is the

quotient space [61] of E meaning it is the finest space[62] such that every $k \in K$ has a corresponding fiber F_k[61].

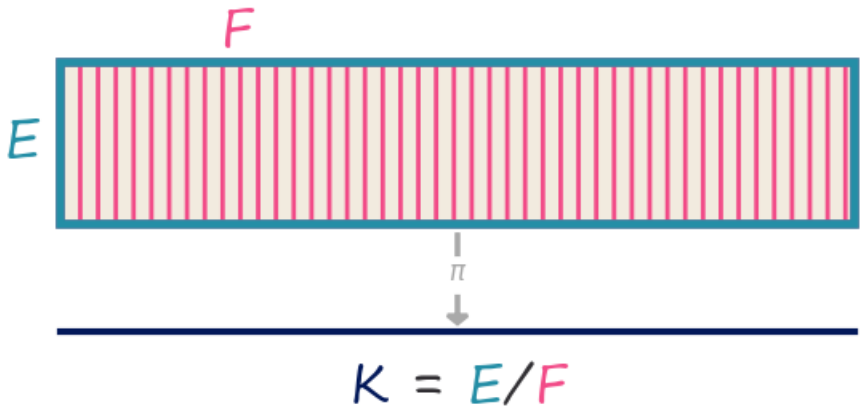


Figure 10: The total space E is divided into fiber segments F. The base space K acts as an index into the records in the fibers, such that every point k has a corresponding fiber F_k. The projection map π maps every fiber F_k to a point $k \in K$ in the base space.

In Figure 10, E is a rectangle divided by vertical fibers F, so the minimal K for which there is always a mapping $\pi : E \to K$ is the closed interval $[0, 1]$. While the total space E may have components in F that describe any given point $k \in K$, such as *time, latitutde, longitude*, these labels are indexed into from K the same as any other components. In contrast to the structural *keys* with associated *values* proposed by Munzner[51], our model treats keys k as a pure reference to topology. Decoupling the keys from their semantics allows the components identifying the keys to be altered, which provides for coordinate agnostic representation of the continuity and facilitates encoding of data where the independent variable may not be clear. For example total rainfall is dependent on time of day and how much rain has already fallen, and changing the coordinate system or time resolution should have no effect on how the records are connected to each other.

As with Equation 6 and Equation 7, we can decompose the total space into component bundles $\pi : E_i \to K$ where

$$\pi : E_1 \oplus \ldots \oplus E_i \oplus \ldots \oplus E_n \to K \tag{8}$$

such that the monoid M_i acts on component bundle E_i. The K remains the same because the continuity of the data does not change just because there are fewer components in each record.

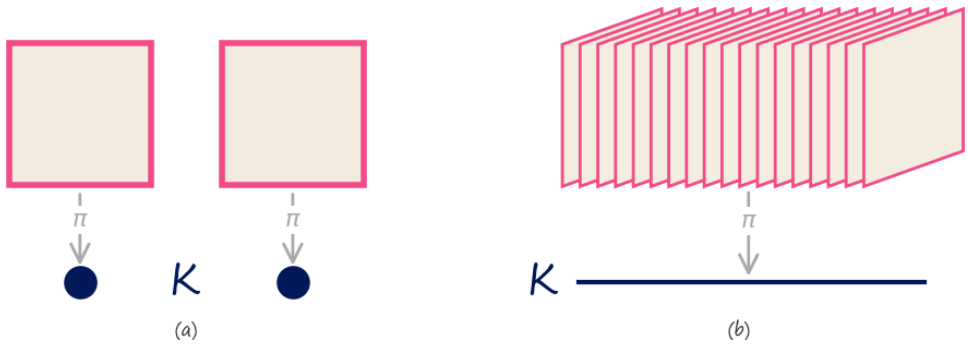


Figure 11: The fiber bundles in (a) and (b) encode the two component dataset from Figure 7, with *(time, precipitation)* components, as having different continuities. The fiber bundle with discrete continuity (a) encodes the dataset as being a set of discrete records. The fiber bundle over the continuous interval K (b) encodes the records as if they were sampled from a 1D continuous space.

The datasets in Figure 11 have the same fiber of (precipitation, time). The points (a) represent a discrete base space K, meaning that every dataset encoded in the fiber bundle has discrete continuity. The line (b) is a representation of a 1D continuity, meaning that every dataset in the fiber bundle is 1D continuous. Explicitly encoding data continuity, for example that (a) has discrete continuity and (b) is 1D continuous, provides a means to explicitly specify the continuities visualization components must preserve.

3.1.4 Data τ

While the projection function $\pi : E \to K$ ties together the base space K with the fiber F, a section $\tau : K \to E$ encodes a dataset. A section function takes as input location $k \in K$ and returns a record $r \in E$. For example, in the special case of a table [50], K is a set of row ids, F is the columns, and the section τ returns the record r at a given key in K. For any fiber bundle, there exists a map

$$\begin{array}{ccc} F & \hookrightarrow & E \\ & & \pi \downarrow \uparrow \tau \\ & & K \end{array} \tag{9}$$

such that $\pi(\tau(k)) = k$. The set of all global sections is denoted as $\Gamma(E)$. Assuming a trivial fiber bundle $E = K \times F$, the section can be decomposed as

$$\tau(k) = (k, (g_{F_0}(k), \ldots, g_{F_n}(k))) \tag{10}$$

where $g : K \to F$ is the index function into the fiber. This formulation of the section also holds on locally trivial sections of a non-trivial fiber bundle. Because we can decompose the bundle and the fiber (Equation 8, Equation 6), we can decompose τ as

$$\tau = (\tau_0, \ldots, \tau_i, \ldots, \tau_n) \tag{11}$$

where each section τ_i maps into a record on a component $F_i \in F$. This allows for accessing the data component wise in addition to accessing the data in terms of its location over K.

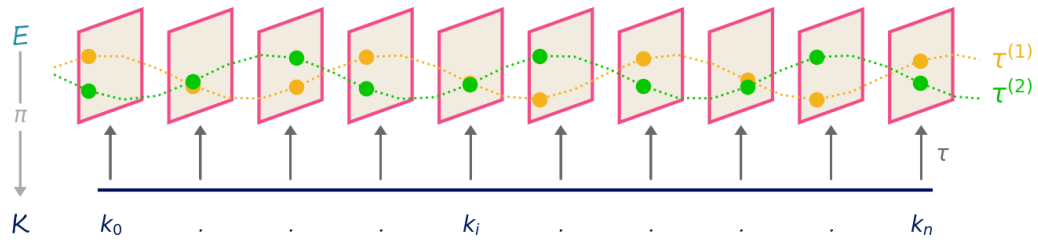


Figure 12: Fiber (time, precipitation) with a 1D continuous K defined on an interval $[0, n]$. The sections $\tau^{(1)}$ and $\tau^{(2)}$ are constrained such that the time variable must be monotonic, which means each section is a timeseries of precipitation values. They are included in the global set of sections $\tau^{(1)}, \tau^{(2)} \in \Gamma(E)$

In Figure 12, the fiber is the same encoding of (*time*, *precipitation*) illustrated in Figure 7, and the base space is the interval K shown in Figure 11. The section $\tau^{(1)}$ is a function that for a point k returns a record in the fiber F. The section applied to a set of points in K resolves to a series of monotonically increasing in time records of *(time, precipitation)* values. Section $\tau^{(2)}$ returns a different timeseries of *(time, precipitation)* values. Both sections are included in the global set of sections $\tau^{(1)}, \tau^{(2)} \in \Gamma(E)$.

3.1.5 Sheafs

Dynamic visualizations require evaluating sections on different subspaces of K; this can be achieved using a mathematical structure, called a sheaf $\mathcal{O}$, for defining collections of objects[63–65] on mathematical spaces. On the fiber bundle E, we can describe a sheaf as the collection of local sections $\iota^*\tau$

$$\begin{array}{ccc} \iota^* E & \xhookrightarrow{\iota^*} & E \\ {\scriptstyle\pi}\downarrow\uparrow{\scriptstyle\iota^*\tau} & & {\scriptstyle\pi}\downarrow\uparrow{\scriptstyle\tau} \\ U & \xhookrightarrow{\iota} & K \end{array} \tag{12}$$

which are sections of E pulled back over local neighborhood $U \subset E$ via the inclusion map $\iota : E \to U$. The collation of sections enabled by sheafs is necessary for navigation techniques such as pan and zoom[66] and dynamically updated visualizations such as sliding windows[67, 68].

3.1.6 Applications

Using fiber bundles as our data abstraction allows the model to be applicable to a wide variety of fields and use cases. For example, the section can be any instance of a numpy array[69] that stores an image, such as an image where the K is a 2D continuous plane and the F is $(\mathbb{R}^3, \mathbb{R}, \mathbb{R})$. In this fiber, the $\mathbb{R}^3$ components encode color, and the other two components are the x and y positions of the sampled data in the image. The continuity of the image is implicitly encoded in the array as the index, so the position components encode the resolution. Instead of an image, the numpy array could also store a 2D discrete table. The fiber may not change, but the K would now be 0D discrete points. These different choices in topology indicate, for example, what sorts of interpolation would be appropriate when visualizing the data. Labeled containers can also be described in this framework because of the schema like structure of the fiber. One such example is a pandas series which stores a labeled list, another is a dataframe[70] which has the structure of a relational table. A series could store the values of $\tau^{(1)}$ and a second series could be $\tau^{(2)}$, while a dataframe would have multiple components and each data frame would be a unique section τ. The ability to encode complexity in continuity and components is particulary beneficial when working with N dimensional labeled data containers. For example, an xarray[45] data cube that stores precipitation would be a section of a fiber bundle with a K that is a continuous volume and components *(time, latitude, longitude, precipitation)*. This section does not need to resolve to values immediately and instead can be an instance of a distributed data container, such as a dask array [71]. Our model provides a common formalism for describing widely used data containers without sacrificing the semantic structure embedded in each container.

3.2 Graphic Space H

To establish that the artist is a structure preserving map from data E to graphic H we construct a graphic bundle so that we can define *equivariance* in terms of maps on the fiber spaces and *continuity* in terms of maps on the base space. As with the data, we can

represent the target graphic as a section ρ of a bundle (H, S, π, D).

$$\begin{array}{ccc} D & \hookrightarrow & H \\ & & \pi \downarrow \uparrow \rho \\ & & S \end{array} \tag{13}$$

The graphic bundle H consists of a base S(subsubsection 3.2.1) that is a thickened form of K a fiber D(subsubsection 3.2.2) that is an idealized display space, and sections ρ(subsubsection 3.2.3) that encode a graphic where the visual characteristics are fully specified.

3.2.1 Idealized Display D

To fully specify the visual characteristics of the image, we construct a fiber D that is a non-pixelated version of the target space. Typically H is trivial and therefore sections can be thought of as mappings into D. In this work, we assume a 2D opaque image $D = \mathbb{R}^5$ with elements

$$(x,\ y,\ r,\ g,\ b) \in D$$

such that a rendered graphic only consists of 2D position and color. To support overplotting and transparency, the fiber could be $D = \mathbb{R}^7$ such that $(x, y, z, r, g, b, a) \in D$ specifies the target display. By abstracting the target display space as D, the model can support different targets, such as a 2D screen or 3D printer.

3.2.2 Continuity of the Graphic S

For a visualization component to preserve continuity, we propose that there must exist a structure preserving surjective map $\xi : S \to K$ from the data base space K to the graphic base space S. Formally, we require that K be a deformation retract[72] of S such that K and S have the same homotopy, meaning there is a continuous map from S to K[73]. The surjective map $\xi : S \to K$

$$\begin{array}{ccc} E & & H \\ \pi \downarrow & & \pi \downarrow \\ K & \xleftarrow{\xi} & S \end{array} \tag{14}$$

goes from region $s \in S_k$ to its associated point $k \in K$. This means that if $\xi(s) = k$, the record at k is copied over the region s such that $\tau(k) = \xi^*\tau(s)$ where $\xi^*\tau(s)$ is τ pulled back over S. The map ξ is part of the implementation of the artist $\mathcal{A}$ and therefore is not defined in terms of the data; instead it is how we specify the constraint that the type of the graphic *continuity* must be able to map to the type of the data *continuity*.

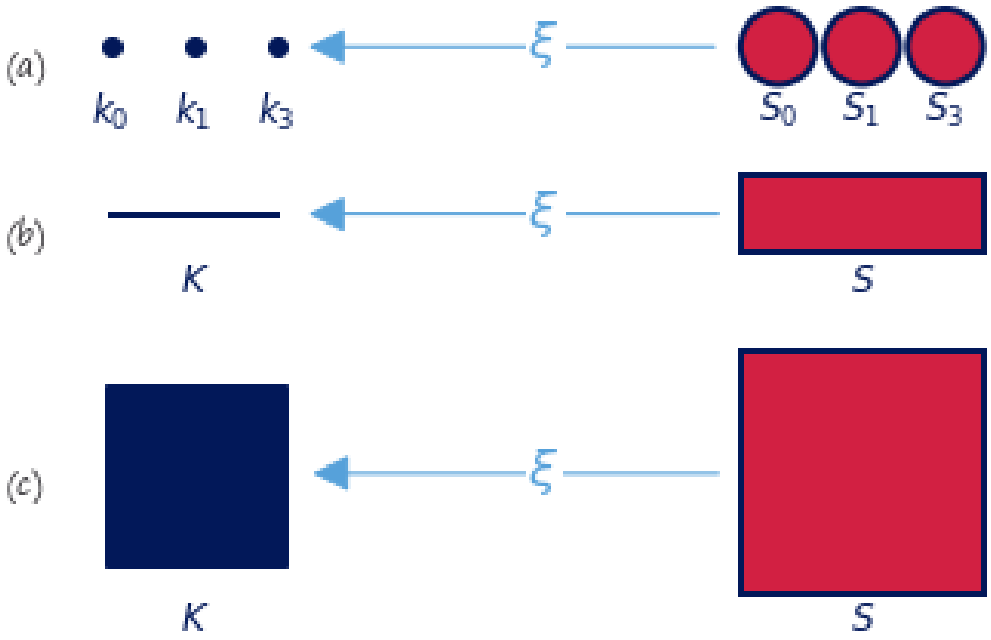


Figure 13: For a visualization component to preserve continuity, it must have a continuous surjective map $\xi : S \to K$ from graphic continuity to data continuity. The scatter (a) and line (b) graphic base spaces S have one more dimension of continuity than K so that S can encode physical aspects of the glyph, such as shape (a circle) or thickness. The image (c) has the same dimension in S as in K because K is already 2D and therefore can directly map into screen space.

To encode the continuity of the elements in the display fiber D, the graphic base space S has the same dimensionality as the target output space. For example, in Figure 13 the base space S is a representation of a region of a 2D display space. Since S must have the same dimensionality as the output graphic, it is allowed to add dimensions to K to make K renderable. A point that is 0D in K cannot be represented on screen unless it is thickened to

2D (a) to encode the connectivity of the pixels that visually represent the point. This is also the case with the line (b), which would be infinitely think on screen if S was not thickened to 2D. This thickening is often not necessary when the dimensionality of K matches the dimensionality of the target space, for example if K is 2D and the display is a 2D screen (c). Since the mapping function ξ binds the graphic base space to the data base space, it can be used by interactive visualization components to look up the data associated with a region on screen. One example is to fill in details in a hover tooltip, another is to convert region selection (such as zooming) on S to a query on the data to access the corresponding record components on K.

3.2.3 Graphic ρ

The section $\rho : S \to H$ is the graphic in an idealized prerender space and also acts as a specification for rendering the graphic to target display format. To demonstrate the role of ρ it is sufficient to sketch out how an arbitrary pixel would be rendered, where a pixel p in a real display corresponds to a region S_p in the idealized display. To determine the color of the pixel, we aggregate the color values over the region via integration:

$$r_p = \iint_{S_p} \rho_r(s) ds^2$$
$$g_p = \iint_{S_p} \rho_g(s) ds^2$$
$$b_p = \iint_{S_p} \rho_b(s) ds^2$$

For a 2D screen, the pixel is defined as a region $p = [y_{top}, y_{bottom}, x_{right}, x_{left}]$ of the rendered graphic. Since the x and y in p are in the same coordinate system as the x and y components of D the inverse map of the bounding box $S_p = {\rho_{x,y}}^{-1}(p)$ is a region $S_p \subset S$. The color is the result of the integration over S_p.

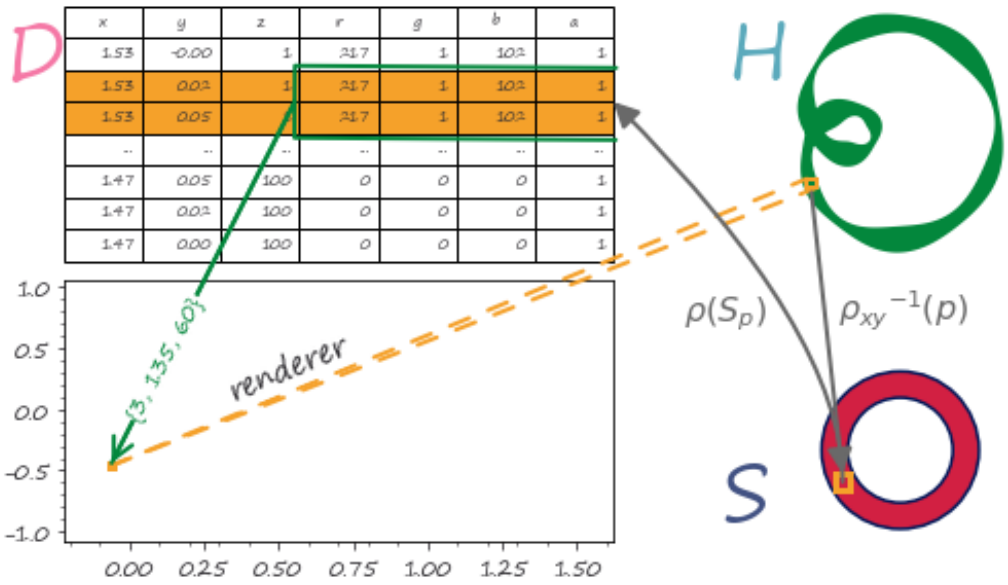


Figure 14: To render a graphic, a pixel p is selected in the display space, which is defined in the same coordinates as the x and y components in D via the renderer. In H the inverse mapping $\rho_{x,y}(p)$ returns a region $S_p \subset S$. $\rho(S_p)$ returns a set of points $(x, y, r, g, b) \in D$ that lie over S_p. The integral over the (r, g, b) pixels specifies that the pixel should be green

As shown in Figure 14, a pixel p in the output space, drawn in yellow, is selected and mapped, via the renderer, into a region on H. The region on H corresponds to a region $S_p \subset S$ via the inverse mapping $\rho_{xy}(p)$. The base space S is an annulus to match the topology of the graphic idealized in H. The section $\rho(S_p)$ then maps into the fiber D over S_p to obtain the set of points in D, here represented as a table, that correspond to that section. The integral over the pixel components of this set of points in the fiber yields the color of the pixel. In general, ρ is an abstraction of rendering. In very broad strokes ρ can be a specification such as PDF[74], SVG[75], or an openGL scene graph[76] or a rendering engine such as cairo[77] or AGG[78]. Implementation of ρ is out of scope for this proposal.

3.3 Artist

We propose that visualization is structure preserving maps from data E to graphic H; having described Ein subsection 3.1 and Hin subsection 3.2, we now define the visual transforma-

tions from E to H that formalize the components that visualization libraries implement. The topological artist A is a map from the sheaf on a data bundle E which is $\mathcal{O}(E)$ to the sheaf on the graphic bundle H, $\mathcal{O}(H)$.

$$A : \mathcal{O}(E) \to \mathcal{O}(H) \tag{15}$$

The artist preserves *continuity* through the ξ map discussed in subsubsection 3.2.2. We propose that the artist $\mathscr{A}$ is an *equivariant* map of monoid action $m \in M$

$$A(m \cdot r) = \varphi(m) \cdot A(r) \tag{16}$$

between data element $r \in \mathscr{E}$ and graphic element $A(r) \in \mathscr{H}$. To be equivariant with respect to monoids action, we conjecture that an artist carries a monoid homomorphism φ

$$\varphi : M \to M' \tag{17}$$

such that an action in data space $m \in M$ is equivalent to an action in graphic space $\varphi(M) \in M'$.

The artist A has two stages: the encoders $\nu : E' \to V$ convert the data components to visual components, and the assembly function $Q : \xi^* V \to H$ composites the fiber components of $\xi^* V$ into a graphic in H.

$$\begin{array}{ccccccc} E & \xrightarrow{\nu} & V & \xleftarrow{\xi^*} & \xi^* V & \xrightarrow{Q} & H \\ & \searrow^{\pi} & \downarrow{\scriptstyle \pi} & & \downarrow{\scriptstyle \xi^*\pi} & \swarrow_{\pi} & \\ & & K & \xleftarrow{\xi} & S & & \end{array} \tag{18}$$

$\xi^* V$ is the visual bundle V pulled back over S via the equivariant continuity map $\xi : S \to K$ introduced in subsubsection 3.2.2. The functional decomposition of the visualization artist in Equation 18 facilitates building reusable components at each stage of the transformation because the equivariance constraints are defined on ν, Q, and ξ. We name this map the artist as that is the analogous part of the Matplotlib[7] architecture that builds visual elements.

3.3.1 Visual Fiber Bundle V

We introduce a visual bundle V to store the mappings of the data components into components of the graphic. These graphic components are implicit visualization library APIS; by making them explicit as components of the fiber we can define expectations of how these parameters behave. As with the data and graphic bundles, the visual bundle (V, K, π, P) is defined by the projection map π

$$\begin{array}{ccc} P & \hookrightarrow & V \\ & & \pi \downarrow \uparrow \mu \\ & & K \end{array} \tag{19}$$

where μ is the visual variable encoding, as described by Bertin [9], of the data section τ. The visual bundle V is the full design space[79] of possible parameters of a visualization type, such as a scatter plot or line plot. For example, one section μ of V is a tuple of visual values that specifies the visual characteristics of a part of a graphic.

$\boldsymbol{\nu_i}$	$\boldsymbol{\mu_i}$	$\boldsymbol{codomain(\nu_i) \subset P_i}$
position	x, y, z, theta, r	$\mathbb{R}$
size	linewidth, markersize	$\mathbb{R}^+$
shape	markerstyle	$\{f_0, \ldots, f_n\}$
color	color, facecolor, markerfacecolor, edgecolor	$\mathbb{R}^4$
texture	hatch	$\mathbb{N}^{10}$
	linestyle	$(\mathbb{R}, \mathbb{R}^{+n, n\%2=0})$

Table 1: Some possible components of the fiber P for a visualization function implemented in Matplotlib

In Table 1, the fiber components are specified by the visual parameter they are encoding. Multiple parameters can be encoded with the same transformation from data space to graphic space, for example x and y are both positions on a screen. Given a fiber of $\{x, y, color\}$ one possible section could be $\{.5, .5, (255, 20, 147)\}$. The $codomain(\nu_i)$ in Table 1 specifies the libraries internal representation of visual variables and can be used to determine which monoids can act on P_i.

3.3.2 Visual Encoders ν

We propose that the map from data components to graphic components $\nu : \tau \mapsto \mu$ is a monoid *equivariant* map. By specifying this constraints, we can guarantee that the stage of the artist that transforms data components into graphic representations is equivariant. These constraints then guide the implementation of reusable component transformers ν that

are composed when generating the graphic. We define the visual transformers ν

$$\{\nu_0, \ldots, \nu_n\} : \{\tau_0, \ldots, \tau_n\} \mapsto \{\mu_0, \ldots, \mu_n\} \tag{20}$$

as the set of equivariant maps $\nu_i : \tau_i \mapsto \mu_i$. Given M_i is the monoid action on E_i and that there is a monoid $M_i{}'$ on V_i, then there is a monoid homomorphism from $\varphi : M_i \to M_i{}'$ that ν must preserve. As mentioned in subsubsection 3.1.2, monoid actions define the structure on the fiber components and are therefore the basis for equivariance. Therefore, a validly constructed ν is one where the diagram of the monoid transform m commutes

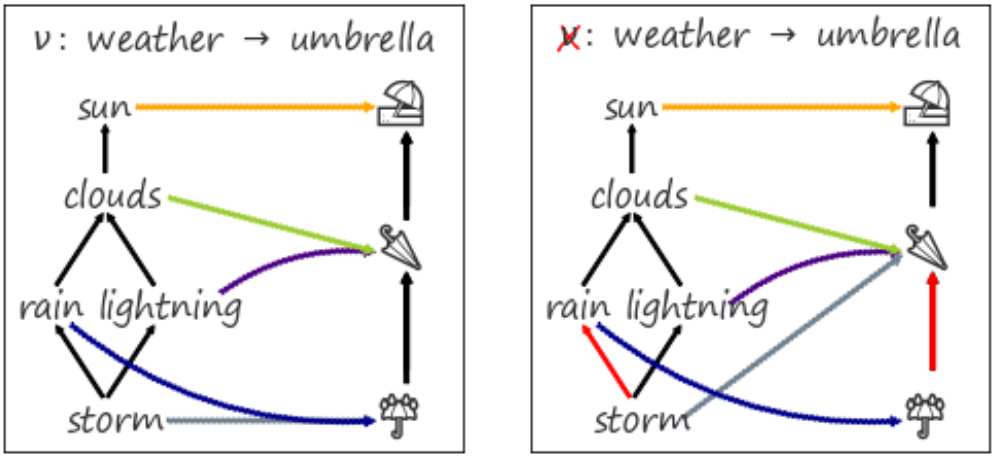
$$\begin{array}{ccc} E_i & \xrightarrow{\nu_i} & V_i \\ {\scriptstyle m_r}\downarrow & & \downarrow{\scriptstyle m_v} \\ E_i & \xrightarrow{\nu_i} & V_i \end{array} \tag{21}$$

such that applying equivariant monoid actions to E_i and V_i preserves the map $\nu_i : E_i \to V_i$. In general, the data fiber F_i cannot be assumed to be of the same type as the visual fiber P_i and the actions of M on F_i cannot be assumed to be the same as the actions of M' on P; therefore an equivariant ν_i must satisfy the constraint

$$\nu_i(m_r(E_i)) = \varphi(m_r)(\nu_i(E_i)) \tag{22}$$

such that φ maps a monoid action on data to a monoid action on visual elements. However, without a loss of generality we can assume that an action of M acts on F_i and on P_i compatibly such that φ is the identity function. We can make this assumption because we can construct a monoid action of M on P_i that is compatible with a monoid action of M on F_i. We can then compose the monoid actions on the visual fiber $M' \times P_i \to P_i$ with the homomorphism φ that takes M to M'. This allows us to define a monoid action on P of M that is $(m, v) \to \varphi(m) \bullet v$, which lets us incorporate φ into the action $\bullet$ such that φ does not need to be explicitly defined in the constraints.

The mapping from weather state to umbrella in Figure 15a is monotonic, and therefore we conjecture equivariant, because $\nu(rain) = \nu(storm)$ satisfies the monotonic con-



(a) (b)

Figure 15: The ν mapping in Figure 15a represented by the colored arrows is monotonic, and therefore monoid equivariant, since $\nu(storm) = \nu(rain)$ satisfies the condition $\nu(storm) \geq \nu(storm)$. In contrast, the map from data component to visual component in Figure 15b is not monotonic, and therefore not monoid equivariant, because $rain \geq storm$ is mapped to elements with the reverse ordering $\nu(storm) \geq \nu(storm)$.

dition of $rain \geq storm$. Figure 15 is an example of how the model supports partially ordered data components, which was a motivation for defining equivariance as monoid homomorphisms. In contrast, the translation from weather state data to visual representation as umbrella emoji in Figure 15b is an invalid visual encoding map ν because it is not monotonic and therefore not equivariant This is because the monotonic condition $rain \geq storm \implies \nu(rain) \geq \nu(storm)$ is not met since $\nu(rain) \leq \nu(storm)$. To satisfy the monotonic condition for $rain \geq storm$, either red arrow in Figure 15b would have to go in a different direction.

scale	group	constraint
nominal	permutation	if $r_1 \neq r_2$ then $\nu(r_1) \neq \nu(r_2)$
ordinal	monotonic	if $r_1 \leq r_2$ then $\nu(r_1) \leq \nu(r_2)$
interval	translation	$\nu(x + c) = \nu(x) + c$
ratio	scaling	$\nu(xc) = \nu(x) * c$

Table 2: Equivariance constraints for the Stevens' measurement scales[80]

The Stevens measurement types[59], listed in Table 2, are specified in terms of groups, which are monoids with invertible operations[81]. We generalize to monoids to account for limitations in the types of data that can be described with the Stevens' scales [82, 83]

3.3.3 Visualization Assembly

Having described the maps to components in subsubsection 3.3.2, we now specify the assembly function $\hat{Q}$ that composites components in V into a graphic in H. Since the component transformes ν are equivariant, the equivariance constraints carry through to $\hat{Q}$. We specify these constraints to guide the implementation of library components responsible for generating graphics.

The transformation from data into graphic is analogous to a map-reduce operation; as illustrated in Figure 16, data components E_i are mapped into visual components V_i that are reduced into a graphic in H. The space of all graphics that Q can generate is the subset of graphics reachable via applying the reduction function $Q(\Gamma(V)) \in \Gamma(H)$ to the visual section $\mu \in \Gamma(V)$. The full space of graphics is not necessarily equivariant; therefore we formalize the constraints on Q such that it produces structure preserving graphics.

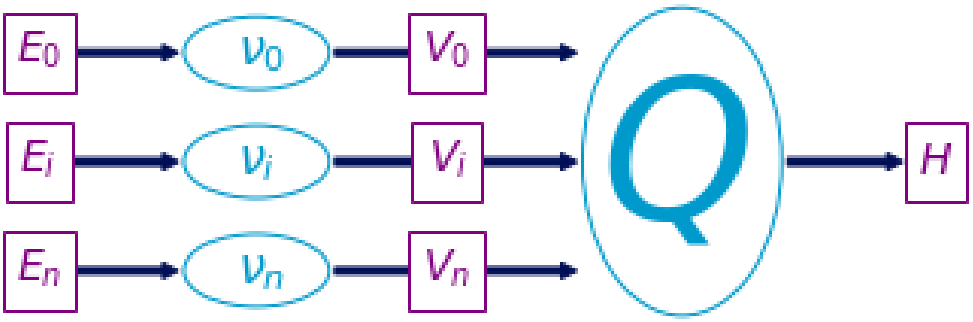


Figure 16: The transform functions ν_i convert data $\tau_i \in E$ to visual characteristics $\mu_i \in V$. These visual components μ_i are then assembled by Q into a graphic $\rho \in H$.

We define the visualization assembly function $Q : \mu \mapsto \rho$ as an equivariant map to formalize the expectation that two Q functions parameterized in the same way should generate the same graphic. We then define the constraint on Q such that if Q is applied to two visual sections μ and μ' that generate the same ρ then the output of μ and μ' acted on by the same monoid m must be the same. We do not define monoid actions on all of $\Gamma(H)$ because there may be graphics $\rho \in \Gamma(H)$ for which we cannot construct a valid mapping from V. Lets call

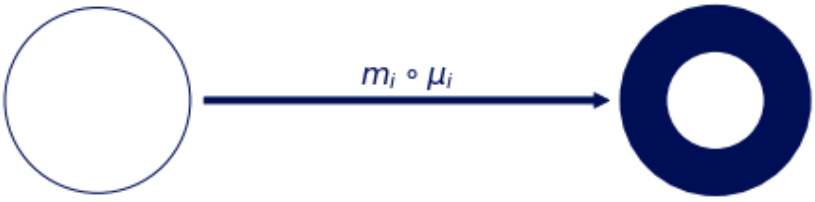


Figure 17: These two glyphs are generated by the same annulus Q function. The monoid action m_i on edge thickness μ_i of the first glyph yields the thicker edge $\mu_i{}'$ in the second glyph.

the visual representations of the components $\Gamma(V) = X$ and the graphic $Q(\Gamma(V)) = Y$

Proposition 1. *If for elements of the monoid $m \in M$ and for all $\mu, \mu' \in X$, we define the monoid action on X so that it is by definition equivariant*

$$Q(\mu) = Q(\mu') \implies Q(m \circ \mu) = Q(m \circ \mu') \tag{23}$$

then a monoid action on Y can be defined as $m \circ \rho = \rho'$. If and only if Q satistfies Equation 23, we can state that the transformed graphic $\rho' = Q(m \circ \mu)$ is equivariant to a monoid action applied on Q with input $\mu \in Q^{-1}(\rho)$ that must generate valid ρ.

For example, given fiber $P = (xpos, ypos, color, thickness)$, then sections $\mu = (0, 0, 0, 1)$ and $Q(\mu) = \rho$ generates a piece of the thin circle. The action $m = (e, e, e, x + 2)$, where e is identity, translates μ to $\mu' = (e, e, e, 3)$ and the corresponding action on ρ causes $Q(\mu')$ to be the thicker circle in Figure 17.

We formally describe a glyph as Q applied to the regions k that map back to a set of path connected components $J \subset K$ as input

$$J = \{j \in K \text{ exists } \gamma \text{ s.t. } \gamma(0) = k \text{ and } \gamma(1) = j\} \tag{24}$$

where the path[84] γ from k to j is a continuous function from the interval [0,1]. We define the glyph as the graphic generated by $Q(S_j)$

$$H \underset{\rho(S_j)}{\overset{}{\rightleftarrows}} S_j \overset{\xi(s)}{\underset{\xi^{-1}(J)}{\rightleftarrows}} J_k \tag{25}$$

such that for every glyph there is at least one corresponding region on K, in keeping with the definition of glyph as any visually distinguishable element put forth by Ziemkiewicz and Kosara[85]. The primitive point, line, and area marks[9, 86] are specially cased glyphs.

3.3.4 Assembly Q

Given the continuities described in 13, we illustrate a minimal Q that will generate the most minimal visualizations associated with those continuities: non-overlapping scatter points, a non-infinitely thin line, and an image.

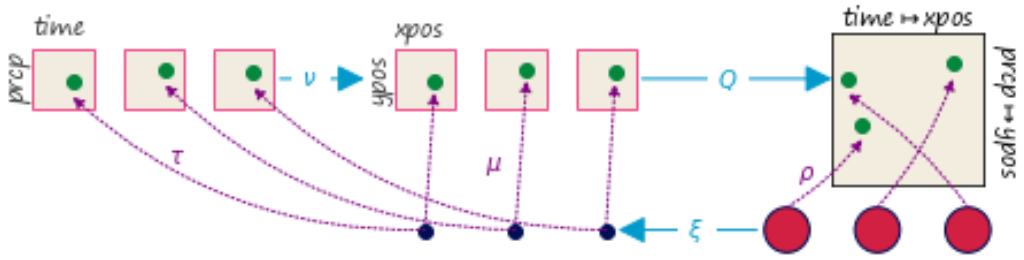


Figure 18: The data is discrete points (time, precipitation). Via ν these are converted to (xpos, ypos) and pulled over discrete S via ξ^*. The pulled back visual secion ν is composited with the assembly function $\hat{Q} \circ \nu = \rho$ to produce the instructions to make the graphic ρ. The graphic section fills in the pixels in the screen via lookup on S.

The scatter plot in Figure 18 has a constant size and color $\rho_{RGB} = (0, 0, 0)$ that are defined as part of the point assembly function.

$$Q(xpos, ypos)(\alpha, \beta) \tag{26}$$

$$x = \text{size} * \alpha \cos(\beta) + xpos$$

$$y = \text{size} * \alpha \sin(\beta) + ypos$$

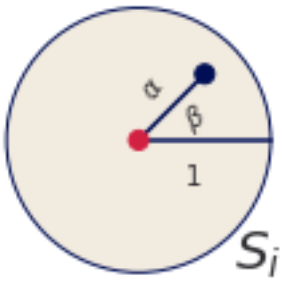


Figure 19: The simplest form of the scatter plot takes as input the expected position of the marker in visual space *(xpos, ypos)*. The marker shape is determined by the polar coordinates (α, β) on the disc; these coordinates dictate whether anything is drawn at that region of S. To obtain the color of the pixel at *(x,y)*, the region on S is scaled by a constant size and shifted by the *xpos* and *ypos*.

The position of this swatch of color is computed relative to the location on the disc $(\alpha, \beta) \in S_k$ as shown in Figure 19. The region $\alpha, \beta)$ is scaled by a constant size and shifted by *xpos* and *ypos*. This computation yields the values *(x,y)* that map into D and have a corresponding function $\rho(s) = (x, y, 0, 0, 0)$ which colors the point *(x,y)* black.

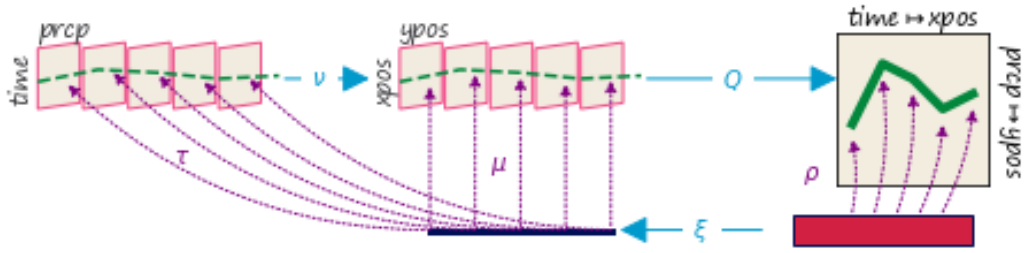
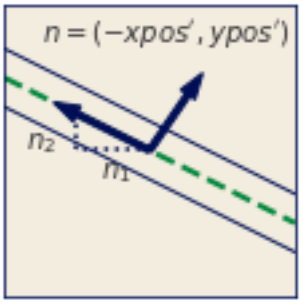


Figure 20: The line fiber (*time*, *precipitation*) is thickened with the derivative (*time′*, *precipitation′* because that information will be necessary to figure out the tangent to the point to draw a line. This is because the line needs to be pushed perpendicular to the tangent of (xpos, ypos). The data is converted to visual characteristics (xpos, ypos). The α coordinates on S specifies the position of the line, the β coordinate specifies thickness.

478 In contrast, the line plot in Figure 20 has a ξ function that is not only parameterized on
479 k but also on the α distance along the interval k and corresponding region in S.

(27) $Q(xpos, n_1, ypos, n_2)(\alpha, \beta)$



$$|n| = \sqrt{{n_1}^2(\xi(\alpha)) + {n_2}^2(\xi(\alpha))}$$

$$\hat{n}_1 = \frac{n_1(\xi(\alpha))}{|n|},\ \hat{n}_2 = \frac{n_2(\xi(\alpha))}{|n|}$$

$$x = xpos(\xi(\alpha)) + \text{width} * \beta \hat{n}_1(\xi(\alpha))$$

$$y = ypos(\xi(\alpha)) + \text{width} * \beta \hat{n}_2(\xi(\alpha))$$

Figure 21: The *xpos* and *ypos* variables give the position of the line in screen space, but render an infinitely thin line. To draw equidistant lines parallel to(*xpos*, *ypos*), defined by the distance (n_1, n_2), requires the derivatives $(n_1 = xpos', n_2 = ypos')$. The position (*xpos*, *ypos*) and width of the line is then used to determine whether a pixel is colored at the position $(x,\ y)$. The values in data space are only looked up via the α coordinate of S because it maps to a location on K. The β parameter is used to specify how thick the line is in conjunction with the constant width.

As shown in Figure 21, line needs to know the tangent of the data to draw an envelope above and below each (xpos,ypos) such that the line appears to have a thickness; therefore the artist takes as input the jet bundle [87, 88] $\mathcal{J}^2(E)$ which is the data E and the first and second derivatives of E. The indexing map $\xi(\alpha)$ finds the point in K corresponding to the region in S at coordinate α. The section τ on the k that corresponds to the region in S returns the position *xpos, ypos* and the derivatives $\hat{n}_1, \hat{n}_2$. The derivatives are then multiplied by a width parameter to specify the thickness of the line. This is then used to determine the color of the pixel at *(x,y)*.

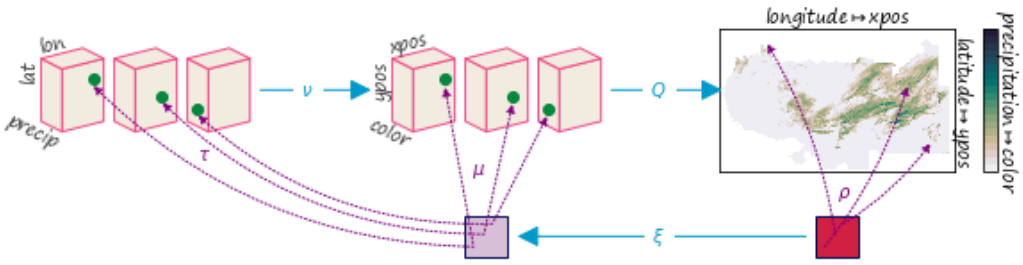


Figure 22: Via ξ the artist maps from a point *(x,y)* on the screen to a corresponding point on K. This maps into F via τ. These data points are converted to visual points via ν and then Q assembles the *(xpos, ypos, color)* parameters into attributes of each pixel.

In Figure 22, the image is a direct lookup into $\xi : S \to K$. The indexing variables (α, β) define the distance along the space, which is then used by ξ to map into K to lookup the color values.

$$Q(xpos, ypos, color)(\alpha, \beta) \tag{28}$$

$$x = xpos(\xi(\alpha))$$

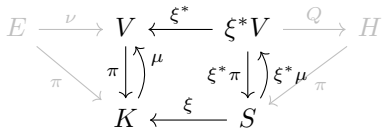
$$y = ypos(\xi(\beta))$$

$$R, G, B = color(\xi(\alpha, \beta))$$

In the case of an image, the indexing mapper ξ may do some translating to a convention expected by Q, for example reorientng the array such that the first row in the data is at the bottom of the graphic.

3.3.5 Assembly Template $\hat{Q}$

The graphic base space S is not accessible in many architectures, including Matplotlib; instead we can construct a factory function $\hat{Q}$ over K that can build a Q. As shown in Equation 18, Q is a bundle map $Q : \xi^* V \to H$ where $\xi^* V$ and H are both bundles over S.



(29)

The map from graphic base space $\xi : S \to K$ (subsubsection 3.2.2) to data space maps many points in S to a single point in K. This means that the preimage of the continuity map $\xi^{-1}(k) \subset S$ is such that many graphic continuity points $s \in S_K$ go to one data continuity point k; therefore, by definition the pull back of μ

$$\xi^* V \mid_{\xi^{-1}(k)} = \xi^{-1}(k) \times P \tag{30}$$

copies the visual fiber P over the the points s in graphic space S that correspond to one k in data space K. This set of points s are the preimage $\xi^{-1}(k)$ of k.

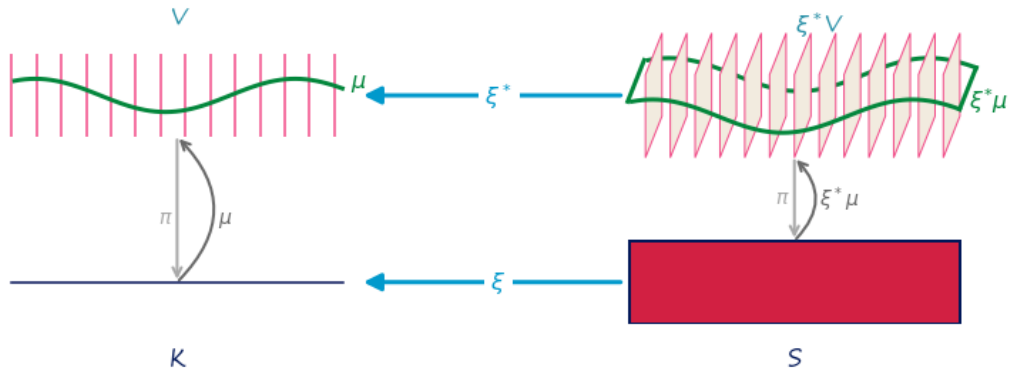


Figure 23: Because the pullback of the visual bundle $\xi^* V$ is the replication of a μ over all points s that map back to a single k, we can construct a $\hat{Q}$ on μ over k that will fabricate the Q for the equivalent region of s associated to that k

As shown in Figure 23, given the section $\xi^*\mu$ pulled back from μ and the point $s \in \xi^{-1}(k)$, there is mapping from section $\xi^*\mu$ over $s$to μ over k. This means that the pulled back section $\xi^*\mu(s) = \xi^*(\mu(k))$ is the section μ copied over all $s$such that $\xi^*\mu$ is identical for all s where $\xi(s) = k$. In Figure 23 each dot on P is equivalent to the line on $P^*\mu$.

Given the equivalence between μ and $\xi^*\mu$ defined above, the reliance on S can be factored out. When Q maps visual sections into graphics $Q : \Gamma(\xi^* V) \to \Gamma(H)$, if we restrict Q input to $\xi^*\mu$ then the graphic section ρ evaluated on a visual region s

$$\rho(s) \coloneqq Q(\xi^*\mu)(s) \tag{31}$$

is defined as the assembly function Q with input $\xi^*\mu$ evaluated on s. Since the pulled back section $\xi^*\mu$ is the section μ copied over every graphic region $s \in \xi^{-1}(k)$, we can define a Q factory function

$$\hat{Q}(\mu(k))(s) \coloneqq Q((\xi^*\mu)(s)) \tag{32}$$

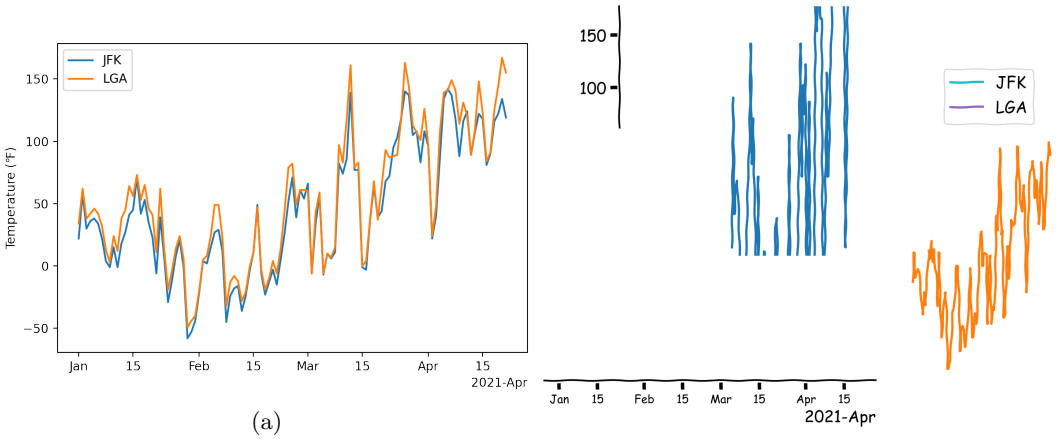
where $\hat{Q}$ with input μ is defined to Q that takes as input the copied section $\xi^*\mu$ such that both functions are evaluated over the same location $\xi^{-1}(k) = s$ in the base space S. We

can then factor s out of Equation 32, which yields

$$\hat{Q}(\mu(k)) = Q(\xi^* \mu) \tag{33}$$

where Q is no longer bound to input but $\hat{Q}$ is still defined in terms of K. In fact, $\hat{Q}$ is a map from visual space to graphic space $\hat{Q} : \Gamma(V) \to \Gamma(H)$ locally over k such that it can be evaluated on a single visual record $\hat{Q} : \Gamma(V_k) \to \Gamma(H \mid_{\xi^{-1}(k)})$. This allows us to construct a $\hat{Q}$ that only depends on K, such that for each $\mu(k)$ there is part of $\rho \mid_{\xi^{-1}(k)}$. The construction of $\hat{Q}$ allows us to retain the functional map reduce benefits of Q without having to restructure the existing pipeline for libraries that delegate the construction of ρ to a back end such as Matplotlib.

3.3.6 Composition of Artists: +



(a) (b)

Figure 24: In Figure 24a, these artists are composited before being added to the image. Disjoint union of E aligns the two timeseries with the x and y axis so all these elements use a shared coordinate system. A more complex composition dictates that the legend is connected to the E such that it must use the same color as the data it is identifying. None of this machinary exists in Figure 24b, therefore each artist is added to the page independented of the other elements.

Visualizations generally consist of more than one artist, commonly having visual elements such as the plot and axis labels and maybe legends. To generate these composite images,

we define addition operators and specify the constraints for compositing artists. Given the family of artists $(E_i : i \in I)$ that are rendered to the same image, the $+$ operator

$$+ \coloneqq \bigsqcup_{i \in I} E_i \tag{34}$$

defines a simple composition of artists. For example, in Figure 24a the data is joined via disjoint union; doing so aligns the components in F such the ν to the same component in P targets the same coordinate system. In Figure 24b, these artists are all added to the image independently of the other and therefore there are no constraints on where they are placed in the image. When artists share a base space $K_2 \hookrightarrow K_1$, a composition operator can be defined such that the artists are acting on different components of the same section. This type of composition is important for visualizations where elements update together in a consistent way, such as multiple views [89, 90] and brush-linked views[91, 92].

3.3.7 Equivalance class of artists A'

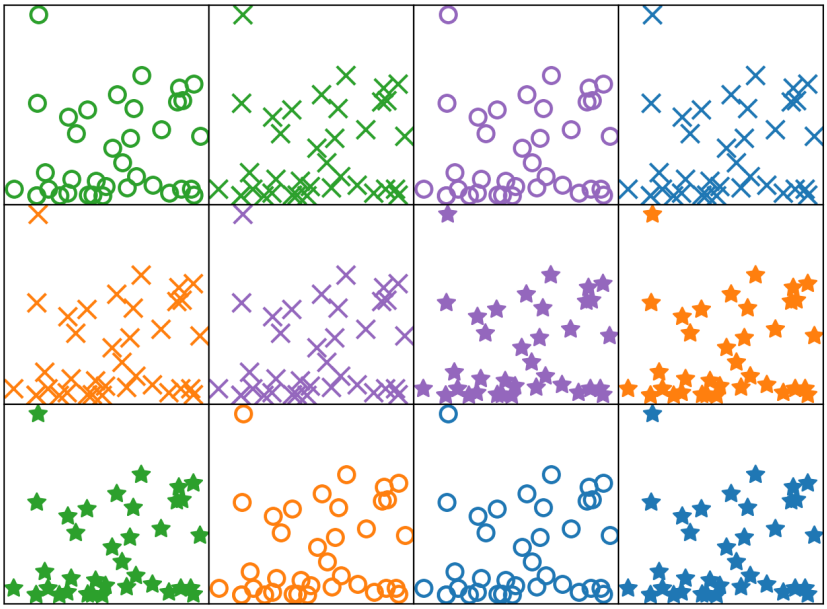


Figure 25: Each scatter plot is generated via a unique artist function A_i, but they only differ in aesthetic styling. Therefore, these artists are all members of an equivalence class $A_i \in A'$

Representational invariance, as defined by Kindlmann and Scheidegger, is the notion that visualizations are equivalent if changing the visual representation, such as colors or shapes, does not change the meaning of the visualization[13]. By defining a criteria for invariance, we can evaluate whether two artists generate the same type of graphic and compare artists across libraries. We propose that visualizations are invariant if they are generated by artists that are members of an equivalence class

$$\{A \in A' : A_1 \equiv A_2\}$$

For example, every scatter plot in Figure 25 is a scatter of the same datasets mapped to the *x position* and *y position* in the same way. The scatter plots only differ in the choice of constant visual literals, differing in color and marker shape. Each scatter is generated by an artist A_i, and every scatter is generated by a member of the equivalence class $A_i \in A'$. Since it is impractical to implement a new artist for every single graphic, the equivalance class provides a way to evaluate an implementation of a generalized artist.

4 Prototype: Matplottoy

To evaluate the feasibility of the model described in section 3, we built prototypes of a `point`, `line`, and `bar` artist. We make use of the Matplotlib Figure and Axes artists [6, 7] 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 Equation 18, we implement the prototype in a heavily object oriented manner. This is done to manage function inputs, especially parameters that are passed through to methods that are structurally functional.

```
fig, ax = plt.subplots()
artist = ArtistClass(E, V)
ax.add_artist(artist)
```

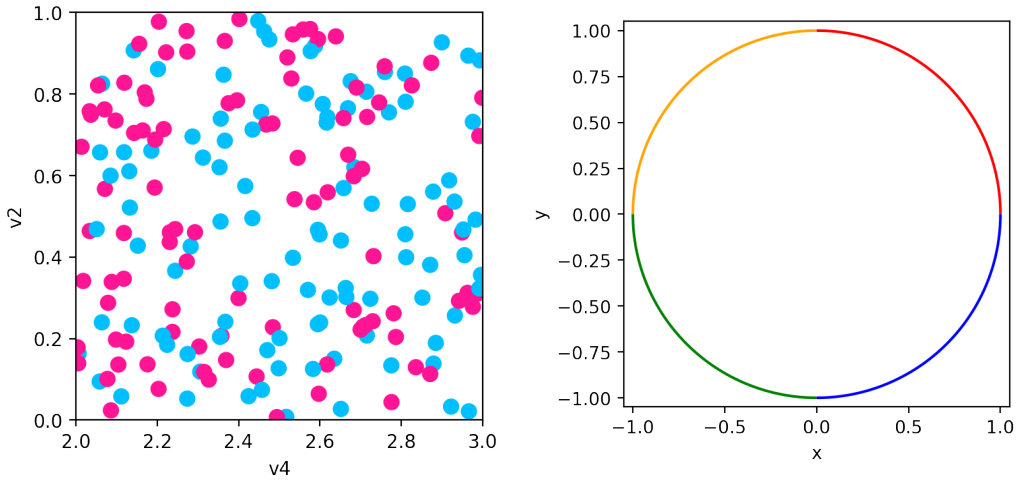
Building on the current Matplotlib Artists which construct an internal representation of the graphic, `ArtistClass` acts as an equivalence class artist A' as described in Figure 3.3.7. The visual bundle V is specified as the `V` dictionary of the form `{parameter:(variable name, encoder)}` where parameter 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 `E` object. By binding data and transforms to A' inside `__init__`, the `draw` method is a fully specified artist A as defined in Equation 15.

```python
class ArtistClass(matplotlib.artist.Artist): #A'
    def __init__(self, E, V, *args, **kwargs):
        # properties that are specific to the graphic
        self.E = E
        self.V = V
        super().__init__(*args, **kwargs)

    def q_hat(self, **args):
        # set the properties of the graphic

    def draw(self, renderer):
        # returns K, indexed on fiber then key
        tau = self.E.view(self.axes)
        # visual channel encoding applied fiberwise
        mu = {p: nu(tau(c))
                 for p, (c, nu) in self.V.items()}
        self.q_hat(**mu)
        # pass configurations off to the renderer
        super().draw(renderer)
```

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 `view` method takes the `axes` 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 subsubsection 3.1.5. We require that the `view` method be atomic so that we do not risk race conditions. Atomicity means that the values cannot change after the method is called in draw until a new call to draw[28], which ensures the integrity of the section.

The ν functions are then applied to the data, as describe in Equation 20, to generate the visual section μthat here is the object `V`. The conversion from data to visual space is simplified here to directly show that it is the encoding ν applied to the component. The `q_hat` function that is $\hat{Q}$, as defined in Equation 33, is responsible for generating a representation such that it could be serialized to recreate a static version of the graphic. The last step in the artist function is handing itself off to the renderer. The extra `*arg, **kwargs` arguments in `__init__`,`draw` are artifacts of how these objects are currently implemented.

4.1 Scatter, Line, and Bar Artists



(a) (b)

Figure 26: Scatter plot and line plot implemented using `Point` and `Line` artists and fiber bundle inspired data models. Matplotlib is used for the rendering.

The figure in Figure 26a is described by **??**. This is implemented via a `Line` 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 `q_hat` function here because the `__init__, draw` are inherited from the prototype artist `ArtistClass`.

The `view` method repackages the data as a fiber component indexed table of vertices. Even though the `view` 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.

```
class Point(ArtistClass, mcollections.Collection):
  def q_hat(self, x, y, s, facecolors): #\hat{Q}
    # construct geometries of circle glyphs
    self._paths = [mpath.Path.circle((xi,yi), radius=si)
                for (xi, yi, si) in zip(x, y, s)]
    # set attributes of glyphs, these are vectorized
    # circles and facecolors are lists of the same size
    self.set_facecolors(facecolors)
```

In `q_hat`, 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 `Path.circle` object.

```
class Line(ArtistClass, mcollections.LineCollection):
  def q_hat(self, x, y, color): #\hat{Q}
    #assemble line marks as set of segments
    segments = [np.vstack((vx, vy)).T for vx, vy
                in zip(x, y)]
    self.set_segments(segments)
    self.set_color(color)
```

To generate Figure 26b, the `Line` artist `view` 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 `Point`, the data is then converted into visual variables. In `q_hat`, described by **??**, this visual representation is composed into a set of line segments, where each segment is the array generated by `np.vstack((vx, vy))`. 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 implementation of line in Matplotlib

does not have this functionality because it has no notion of rows and columns of a table, and therefore no notion of a τ. Instead, line is assumed to be points along one edge and therefore has only one color, and the user is responsible for aligning the x and y components and colors along the implicit K over which they are plotted.

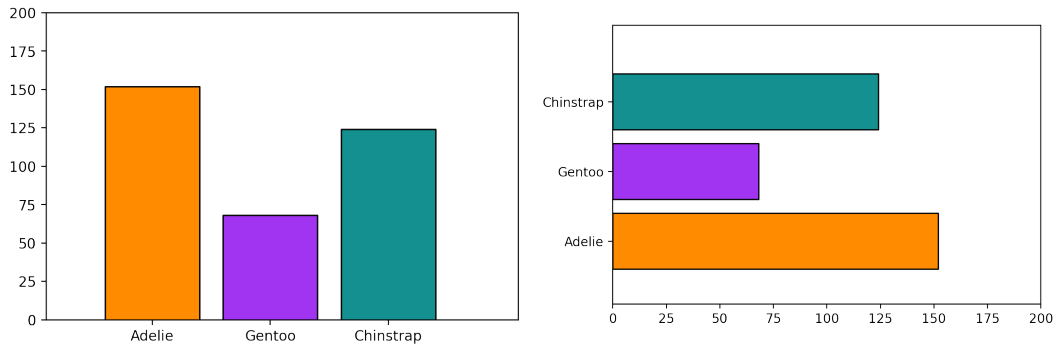


Figure 27: Frequency of Penguin types visualized as discrete bars.

The bar charts in figure 27 are generated with a `Bar` 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 $\hat{Q}$.

```
class Bar(ArtistClass, mcollections.Collection):
    def __init__(self, E, V, orientation, *args, **kwargs):
        """
        orientation: str
            v: bars aligned along x axis, heights on y
            h: bars aligned along y axis, heights on x
        """
        self.orientation = orientation
        super().__init__(*args, **kwargs) # set E & V
```

```
10
11      def q_hat(self, position, length, floor, width,
12                      facecolors, edgecolors, offset):
13          # offset is passed through via assemblers such as multigroup,
14          # not supposed to be directly tagged to position
15          position = position + offset
16
17          def make_bars(xval, xoff, yval, yoff):
18               return [[(x, y), (x, y+yo), (x+xo, y+yo), (x+xo, y), (x, y)]
19                  for (x, xo, y, yo) in zip(xval, xoff, yval, yoff)]
20          #build bar glyphs based on graphic parameter
21          if self.orientation in {'vertical', 'v'}:
22              verts = make_bars(position, width, floor, length)
23          elif self.orientation in {'horizontal', 'h'}:
24              verts = make_bars(floor, length, position, width)
25
26          self._paths = [mpath.Path(xy, closed=True) for xy in verts]
27          self.set_edgecolors(edgecolors)
28          self.set_facecolors(facecolors)
```

As with `Point` and `scatter`, `q_hat` function constructs bars and sets their properties, face and edge colors. The `make_bars` function converts the input position and length to the coordinates of a rectangle of the given width. Typically defaults are used for the type of chart shown in figure 27, but these visual variables are often set when building composite versions of this chart type as discussed in section 4.4.

4.2 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

```
cmap =  color.Categorical({'true':'deeppink',
                           'false':'deepskyblue'})
# {P_i name:{'name':c_i, 'encoder:\nu_i}}
V = {'x': {'name': 'v4', 'encoder': lambda x: x},
     'y': {'name': 'v2', 'encoder': lambda x: x},
     'facecolors': {'name':'v3', 'encoder': cmap},
     's':{'name': None ,
          'encoder': lambda _: itertools.repeat(.02)}}
```

where `lambda x: x` is an identity ν, `{'name':None}` maps into P without corresponding τ to set a constant visual value, and `color.Categorical` is a custom ν implemented as a class.

```
#\nu_i(m_r(E_i)) = \varphi(m_r)(\nu_i((E_i))
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)
```

As described in Equation 22, a test for equivariance can be implemented trivially. It is currently factored out of the artist for clarity.

4.3 Data Model

The data input into the `Artist` 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 `view` method that returns an atomic object that encapsulates τ. To support specifying the fiber bundle, we define a `FiberBundle` data class[93]

```
@dataclass
class FiberBundle:
    K: dict #{'tables': []}
    F: dict #  {variable name: type}
```

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 `tau` method that is the section.

```
class PointData:
    def __init__(self):
        self.FB = FiberBundle({'tables': ['vertex']},
                        {'v1': float, 'v2': str, 'v3': float})
    def tau(self, k):
        return # tau evaluated at one point k

class LineData:
    def __init__(self):
        self.FB = FiberBundle({'tables': ['edge']},
                      {'x': float, 'y': float, 'color': str})
    def tau(self, k):
        return # tau evaluated on interval k
```

The discrete `tau` method returns a record of discrete points, akin to a row in a table, while the line`tau` returns a sampling of points along an edge k. These continuities are also specified in the `k={'tables':[]}` dictionary.

```
def view(self, axes):
    table = defaultdict(list)
    for k in self.keys:
        table['index'].append(k)
        for (name, value) in zip(self.FB.fiber.keys(),
                                 self.tau(k)[1]):
            table[name].append(value)
    return table
```

In both cases, the `view` 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.

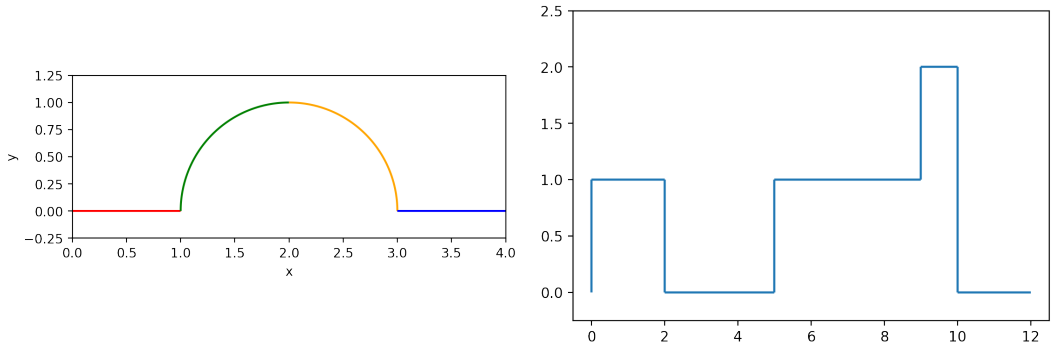


Figure 28: Continuous and discontinuous lines as defined via the same data model, and generated with the same A' `Line`

The graphics in figure Figure 28 are made using the `Line` artist and the `GraphData` 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,

```
GraphData(FB, edges, verticies, num_samples=100, connect=True)
```

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 only needs to be evaluated at the edges of the interval

```
GraphData(FB, edges, verticies, 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.

4.4 Case Study: Penguins

Building on the `Bar` artist in subsection 4.1, 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 achieved 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[94, 95], packaged as a pandas dataframe[96] since that is a very commonly used Python labeled data structure.

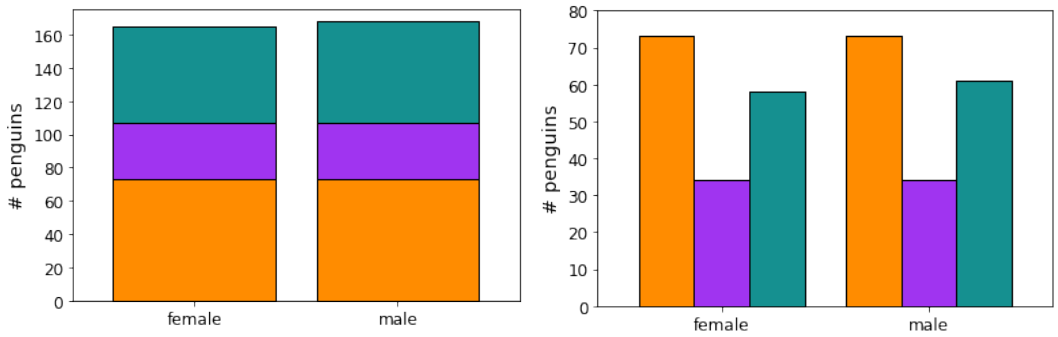
sex	Adelie	Chinstrap	Gentoo	Adelie_c	Chinstrap_c	Gentoo_c
female	73	34	58	Adelie	Chinstrap	Gentoo
male	73	34	61	Adelie	Chinstrap	Gentoo

Table 3: Palmer Penguins dataset that is processed to become input into the grouped bar chart. This data is a count of penguin species. The columns with suffix c are used to specify the color of the corresponding visual element.

The wrapper is very thin because there is explicitly only one section.

```
class DataFrame:
    def __init__(self, dataframe):
        self.FB = FiberBundle(K = {'tables':['vertex']},
                              F = dict(dataframe.dtypes))
        self._tau = dataframe.iloc
        self._view = dataframe

    def view(self, axes=None):
        return self._view
```

Since the aim for this wrapper is to be very generic, here the fiber is set by querying the dataframe for its metadata. The `dtypes` 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 29: Penguin count disaggregated by island and species

The stacked and grouped bar charts in figure 29 are both composites of `Bar` artists such that the difference between `StackedBar` and `GroupedBar` is specific to the ways in which the `Bar` are stitched together. These two artists have identical constructors and `draw` methods. As with `Bar`, the orientation is set in the constructor. In both these artists, we separate the

transforms `V` that are applied to only one component (column) from transforms `MV` applied to multiple components (columns). This convention allows us to, for example, map one column to position, but multiple to length. In effect, we are decomposing E into $E_1, \cdots, E_i, \cdots, E_n$ via specifications in `V` rather than by directly taking subsections of the table. This allows us to ensure shared K and coherant τ.

```
class StackedBar(martist.Artist):
    def __init__(self, E, V, MV, orientation='v', *args, **kwargs):
        """
        Parameters
        ----------

        orientation: str, optional
            vertical: bars aligned along x axis, heights on y
            horizontal: bars aligned along y axis, heights on x
        """
        super().__init__(*args, **kwargs)
        self.E = E
        self.orientation = orientation
        self.V = V
        self.MV = MV

    def q_hat(self):
        tau = self.data.view(self.axes)
        self.children = [] # list of bars to be rendered
        floor = 0
        for group in self.MV:
            # pull out the specific group transforms
            bar = Bar(self.E, {**group, **self.V, 'floor':floor},
```

```
                        self.orientation, transform=self.axes.transData)
            self.children.append(bar)
            floor += view[group['length']['name']]


    def draw(self, renderer, *args, **kwargs):
        # all the visual conversion gets pushed to child artists
        self.assemble()
        #self._transform = self.children[0].get_transform()
        for artist in self.children:
            artist.draw(renderer, *args, **kwargs)
```

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

```
class GroupedBar(martist.Artist):
    def q_hat(self):
        self.children = [] # list of bars to be rendered
        ngroups = len(self.mtransforms)

        for gid, group in enumerate(self.mtransforms):
            group.update(self.transforms)
            width = group.get('width', .8)
            gwidth = width/ngroups
            offset = gid/ngroups*width
```

```
            bar = Bar(self.E, {**group, **self.V, 'width':gwidth, 'offset':offset},
                      self.orientation, transform=self.axes.transData)
            self.children.append(bar)
```

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

```
transforms = {'position': {'name':'sex',
              'encoder': position.Nominal({'female':0, 'male':1})}}
group_transforms =  [{'length': {'name':'Adelie'},
                      'facecolors': {'name':"Adelie_s", 'encoder':cmap}},
                     {'length': {'name':'Chinstrap'},
                      'facecolors': {'name':"Chinstrap_s", 'encoder':cmap}},
                     {'length': {'name':'Gentoo'},
                      'facecolors': {'name':"Gentoo_s", 'encoder':cmap}}]
```

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

```
artistSB = bar.StackedBar(bt, ts, group_transforms)
artistGB = bar.GroupedBar(bt, ts, group_transforms)
```

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 `draw` 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.

5 Discussion

The Topological Equivariant Artist Model (TEAM) is a functional model of the structure preserving maps from data to visual representation. TEAM expresses the specifications that graphic and data must have equivalent *continuity* equivalent to the data, and that the visual characteristics of the graphics are *equivariant* to their corresponding components. TEAM expresses these constraints in the encoding ν, assembly Q, and reindexing ξ functions that make up the artist A. This decomposition of the artist into smaller components functional pieces allows TEAM to provide well specified guidance on implementing visualization components based on this architecture. The proof of concept prototype built using this model validates that it is usable for a general purpose visualization tool. Additionally, the decomposition facilitates iteratively integrating TEAM into existing architecture rather than starting from scratch, since ν, Q and ξ functions can be implemented independently. This prototype demonstrates that this framework can generate the fundamental point (scatter plot) and line (line chart) marks. Furthermore, combining Butler's proposal of a fiber bundle model of visualization data with Spivak's formalism of schema lets TEAM support a variety of data continuities, including discrete relational tables, multivariate high resolution spatio temporal datasets, and complex networks. Although the prototype currently only implements 0D and 1D continuity, we expected it to generalize to the other continuities.

5.1 Limitations

The TEAM model is a specification visualization library developers can use to implement structure preserving library components. Implementing a TEAM based architecture involves developers explicitly describing the structure and continuity of the data and the structure and continuity the artist expect. TEAM does not provide a framework for recommending visualizations to the user, therefore effectiveness [12, 97] is out of scope. But, automatic recommendation tools could be built using TEAM components.

While TEAM specifies the components, the developers building libraries using TEAM components decide which compositions of components are semantically correct for the domain. For this reason, TEAM does not include data space transforms, as are incorporated into libraries like Tableau or ggplot, instead leaving choice of computations to implementors of the data object. TEAM's intentional ignorance of semantics also means it cannot evaluate whether a figurative glyph [2] is a semantically correct choice, but it can enforce equivariance constraints of glyphs generated from data components enforcing equivariance of figurative glyphs [2] generated from data components[98, 99]. TEAM also allows graphics to have a lower dimensional continuity than the source data when a retraction map from one continuity to the other exists. For example, TEAM components could transform 1D continuous segments into 0D discrete elements, e.g. bar charts or scatter plots. As with computations, it is the role of the domain specific library to determine which figurative glyphs and continuity downgrades are appropriate.

The prototype is deeply tied to Matplotlib's existing architecture, so it has not yet been worked through how the model generalizes to libraries such as R graphics[100], VTK, and D3. TEAM has only been tested using PNGs rendered with AGG[78], but is expected to work with all the file types Matplotlib currently supports, including svg, pdf, and eps. We have not yet addressed how this framework interfaces with high performance rendering libraries such as openGL[76] that implement different models of ρ.

5.2 Future Work

More work is needed to formalize the composition operators, equivalence class A', and the mathematical model of interactivity. We also need to implement artists that demonstrate that the model can underpin a minimally viable library, foremost an image[101, 102], a heatmap[103, 104], and an inherently computational artist such as a boxplot[36]. In summary, the proposed scope of work is

work period	**milestones & tasks**
April - July 2021	prepare and submit **conference presentation** on new functionality enabled by model for *SciPy*: artists that do not inherit from existing Matplotlib artists, computational artists such as histograms, non tabular data, composite interactive artist
June - Sept 2021	prepare and submit **theory paper** on interactivity to *TCVG or Eurovis 2022*: fully work out and describe math of addition operators and lookups from graphic to data space, implement brush linked artist (shared base space) and artist that exploits sheafs
May - Nov 2021	prepare and submit **applications paper** on high dimensional to *TCVG*: math and implementation of computational artists, concurrent artists and data sources, non-trivial data bundles
Aug 2021 - Feb 2022	prepare and submit **systems paper** on building domain specific libraries based on this model to *Infoviz 2022*: domain specific structured data, composite artists, inference of meta data components, mathematical notion of a visualization (labeled, multiple artists, etc)
March 2022	**dissertation** writing: synthesize previous work on climate data, compile topological equivariant artist model work
April 2022	**defense**

Table 4

In acknowledgement that the schedule is optimistic, this work has various scales of data applications. We plan to apply this model to datasets with complex continuities, such as the trajectories of rats running around a maze and the positions of their limbs. We also potentially can look at large scale biology or climate datasets. The data applications could be further integrated with topological[105] and functional [106] data analysis methods. Since this model formalizes notions of structure preservation, it can serve as a good base for tools that assess quality metrics[107] or invariance [13] of visualizations with respect to graphical encoding choices. This specification of structure could also be used to develop a serialization structure tht could then be use to allow Matplotlib to interface with other visualization libraries such as open GL via shared serialization protocol. While this paper formulates visualization in terms of monoidal action homomorphisms between fiberbundles, the model lends itself to a categorical formulation[58, 108] that could be further explored.

6 Conclusion

A TEAM driven refactor of visualizations libraries could produce more maintainable, reusable, and extensible code, leading to better building blocks for the ecosystem of tools built on top of TEAM architectured libraries. Building block libraries could better support downstream, including domain specific, libraries without having to explicitly incorporate the specific data structure and visualization needs of those domains back into the base library. Adopting this model would induce a separation of data representation and visual representation that, for example, in Matplotlib is so entangled that it has lead to a brittle and sometimes incoherent API and internal code base. A refactor that incorporated the generalized data model and functional transforms presented in TEAM would lead to building block libraries that provide a more consistent, reusable, flexible, collection of blocks.

References

[1] Michael Friendly. "A Brief History of Data Visualization". en. In: *Handbook of Data Visualization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 15–56. ISBN: 978-3-540-33036-3 978-3-540-33037-0. DOI: 10.1007/978-3-540-33037-0_2.

[2] L. Byrne, D. Angus, and J. Wiles. "Acquired Codes of Meaning in Data Visualization and Infographics: Beyond Perceptual Primitives". In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (Jan. 2016), pp. 509–518. ISSN: 1077-2626. DOI: 10.1109/TVCG.2015.2467321.

[3] Krist Wongsuphasawat. *Navigating the Wide World of Data Visualization Libraries (on the Web)*. 2021.

[4] J. Hughes. "Why Functional Programming Matters". In: *The Computer Journal* 32.2 (Jan. 1989), pp. 98–107. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.98.

[5] Zhenjiang Hu, John Hughes, and Meng Wang. "How Functional Programming Mattered". In: *National Science Review* 2.3 (Sept. 2015), pp. 349–370. ISSN: 2095-5138. DOI: 10.1093/nsr/nwv042.

[6] J. D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science Engineering* 9.3 (May 2007), pp. 90–95. ISSN: 1558-366X. DOI: 10.1109/MCSE.2007.55.

[7] John Hunter and Michael Droettboom. *The Architecture of Open Source Applications (Volume 2): Matplotlib*. https://www.aosabook.org/en/matplotlib.html.

[8] A. Sarikaya et al. "What Do We Talk About When We Talk About Dashboards?" In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (Jan. 2019), pp. 682–692. ISSN: 1941-0506. DOI: 10.1109/TVCG.2018.2864903.

[9] Jacques Bertin. *Semiology of Graphics : Diagrams, Networks, Maps*. English. Redlands, Calif.: ESRI Press, 2011. ISBN: 978-1-58948-261-6 1-58948-261-1.

[10] Wilson Stothers. *Similarity Group*. https://www.maths.gla.ac.uk/wws/cabripages/klein/similarity.html.

[11] Jock Mackinlay. “Automating the Design of Graphical Presentations of Relational Information”. In: *ACM Transactions on Graphics* 5.2 (Apr. 1986), pp. 110–141. ISSN: 0730-0301. DOI: 10.1145/22949.22950.

[12] Jock Mackinlay. “Automatic Design of Graphical Presentations”. English. PhD Thesis. Stanford, 1987.

[13] G. Kindlmann and C. Scheidegger. “An Algebraic Process for Visualization Design”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (Dec. 2014), pp. 2181–2190. ISSN: 1941-0506. DOI: 10.1109/TVCG.2014.2346325.

[14] Ricky Shadrach. *Introduction to Groups*. https://www.mathsisfun.com/sets/groups-introduction.html. 2017.

[15] *Naturalness Principle - InfoVis:Wiki*. https://infovis-wiki.net/wiki/Naturalness_Principle.

[16] Edward R. Tufte. *The Visual Display of Quantitative Information*. English. Cheshire, Conn.: Graphics Press, 2001. ISBN: 0-9613921-4-2 978-0-9613921-4-7 978-1-930824-13-3 1-930824-13-0.

[17] J. Heer and M. Agrawala. “Software Design Patterns for Information Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (2006), pp. 853–860. DOI: 10.1109/TVCG.2006.178.

[18] E. H. Chi. “A Taxonomy of Visualization Techniques Using the Data State Reference Model”. In: *IEEE Symposium on Information Visualization 2000. INFOVIS 2000. Proceedings*. Oct. 2000, pp. 69–75. DOI: 10.1109/INFVIS.2000.885092.

[19] Jeffrey Heer and Michael Bostock. “Declarative Language Design for Interactive Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.6 (Nov. 2010), pp. 1149–1156. ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.144.

[20] C. Stolte, D. Tang, and P. Hanrahan. “Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases”. In: *IEEE Transactions on Visualization and Computer Graphics* 8.1 (Jan. 2002), pp. 52–65. ISSN: 1941-0506. DOI: 10.1109/2945.981851.

[21] Pat Hanrahan. "VizQL: A Language for Query, Analysis and Visualization". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 721. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142560.

[22] J. Mackinlay, P. Hanrahan, and C. Stolte. "Show Me: Automatic Presentation for Visual Analysis". In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (Nov. 2007), pp. 1137–1144. ISSN: 1941-0506. DOI: 10.1109/TVCG.2007.70594.

[23] Leland Wilkinson. *The Grammar of Graphics*. en. 2nd ed. Statistics and Computing. New York: Springer-Verlag New York, Inc., 2005. ISBN: 978-0-387-24544-7.

[24] Hadley Wickham. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN: 978-3-319-24277-4.

[25] M. Bostock and J. Heer. "Protovis: A Graphical Toolkit for Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1121–1128. ISSN: 1941-0506. DOI: 10.1109/TVCG.2009.174.

[26] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. "Declarative Interaction Design for Data Visualization". en. In: *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*. Honolulu Hawaii USA: ACM, Oct. 2014, pp. 669–678. ISBN: 978-1-4503-3069-5. DOI: 10.1145/2642918.2647360.

[27] Jacob VanderPlas et al. "Altair: Interactive Statistical Visualizations for Python". en. In: *Journal of Open Source Software* 3.32 (Dec. 2018), p. 1057. ISSN: 2475-9066. DOI: 10.21105/joss.01057.

[28] Jeffrey D. Ullman and Jennifer. Widom. *A First Course in Database Systems*. English. Upper Saddle River, NJ: Pearson Prentice Hall, 2008. ISBN: 0-13-600637-X 978-0-13-600637-4.

[29] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. "NIH Image to ImageJ: 25 Years of Image Analysis". In: *Nature Methods* 9.7 (July 2012), pp. 671–675. ISSN: 1548-7105. DOI: 10.1038/nmeth.2089.

[30] Nicholas Sofroniew et al. *Napari/Napari: 0.4.5rc1*. Zenodo. Feb. 2021. DOI: 10.5281/zenodo.4533308.

[31] Software Studies. *Culturevis/Imageplot*. Jan. 2021.

[32] *Writing Plugins*. en. https://imagej.net/Writing_plugins.

[33] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. "Gephi: An Open Source Software for Exploring and Manipulating Networks". en. In: *Proceedings of the International AAAI Conference on Web and Social Media* 3.1 (Mar. 2009). ISSN: 2334-0770.

[34] John Ellson et al. "Graphviz— Open Source Graph Drawing Tools". In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 483–484. ISBN: 978-3-540-45848-7.

[35] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function Using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[36] Hadley Wickham and Lisa Stryjewski. "40 Years of Boxplots". In: *The American Statistician* (2011).

[37] *Data Representation in Mayavi — Mayavi 4.7.2 Documentation*. https://docs.enthought.com/mayavi/mayavi/d

[38] M. Tory and T. Moller. "Rethinking Visualization: A High-Level Taxonomy". In: *IEEE Symposium on Information Visualization*. 2004, pp. 151–158. DOI: 10.1109/INFVIS.2004.59.

[39] M. Bostock, V. Ogievetsky, and J. Heer. "D^3 Data-Driven Documents". In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pp. 2301–2309. ISSN: 1941-0506. DOI: 10.1109/TVCG.2011.185.

[40] Marcus D. Hanwell et al. "The Visualization Toolkit (VTK): Rewriting the Rendering Code for Modern Graphics Cards". en. In: *SoftwareX* 1-2 (Sept. 2015), pp. 9–12. ISSN: 23527110. DOI: 10.1016/j.softx.2015.04.001.

[41] Berk Geveci et al. “VTK”. In: *The Architecture of Open Source Applications* 1 (2012), pp. 387–402.

[42] P. Ramachandran and G. Varoquaux. “Mayavi: 3D Visualization of Scientific Data”. In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 40–51. ISSN: 1558-366X. DOI: 10.1109/MCSE.2011.35.

[43] Michael Waskom and the seaborn development team. *Mwaskom/Seaborn*. Zenodo. Sept. 2020. DOI: 10.5281/zenodo.592845.

[44] Brian Wylie and Jeffrey Baumes. “A Unified Toolkit for Information and Scientific Visualization”. In: *Proc.SPIE*. Vol. 7243. Jan. 2009. DOI: 10.1117/12.805589.

[45] Stephan Hoyer and Joe Hamman. “Xarray: ND Labeled Arrays and Datasets in Python”. In: *Journal of Open Research Software* 5.1 (2017).

[46] James Ahrens, Berk Geveci, and Charles Law. “Paraview: An End-User Tool for Large Data Visualization”. In: *The visualization handbook* 717.8 (2005).

[47] D. M. Butler and M. H. Pendley. “A Visualization Model Based on the Mathematics of Fiber Bundles”. en. In: *Computers in Physics* 3.5 (1989), p. 45. ISSN: 08941866. DOI: 10.1063/1.168345.

[48] David M. Butler and Steve Bryson. “Vector-Bundle Classes Form Powerful Tool for Scientific Visualization”. en. In: *Computers in Physics* 6.6 (1992), p. 576. ISSN: 08941866. DOI: 10.1063/1.4823118.

[49] David I Spivak. *Databases Are Categories*. en. Slides. June 2010.

[50] David I Spivak. “SIMPLICIAL DATABASES”. en. In: (), p. 35.

[51] Tamara Munzner. *Visualization Analysis and Design*. AK Peters Visualization Series. CRC press, Oct. 2014. ISBN: 978-1-4665-0891-0.

[52] Tamara Munzner. “Ch 2: Data Abstraction”. In: *CPSC547: Information Visualization, Fall 2015-2016* ().

[53] E.H. Spanier. *Algebraic Topology*. McGraw-Hill Series in Higher Mathematics. Springer, 1989. ISBN: 978-0-387-94426-5.

[54] *Locally Trivial Fibre Bundle - Encyclopedia of Mathematics*. https://encyclopediaofmath.org/wiki/Locally_trivia

[55] Brent A Yorgey. "Monoids: Theme and Variations (Functional Pearl)". en. In: (), p. 12.

[56] "Monoid". en. In: *Wikipedia* (Jan. 2021).

[57] "Semigroup Action". en. In: *Wikipedia* (Jan. 2021).

[58] Brendan Fong and David I. Spivak. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. en. First. Cambridge University Press, July 2019. ISBN: 978-1-108-66880-4 978-1-108-48229-5 978-1-108-71182-1. DOI: 10.1017/9781108668804.

[59] S. S. Stevens. "On the Theory of Scales of Measurement". In: *Science* 103.2684 (1946), pp. 677–680. ISSN: 00368075, 10959203.

[60] W A Lea. "A Formalization of Measurement Scale Forms". en. In: (), p. 44.

[61] "Quotient Space (Topology)". en. In: *Wikipedia* (Nov. 2020).

[62] Professor Denis Auroux. "Math 131: Introduction to Topology". en. In: (), p. 113.

[63] Robert W. Ghrist. *Elementary Applied Topology*. Vol. 1. Createspace Seattle, 2014.

[64] Robert Ghrist. "Homological Algebra and Data". In: *Math. Data* 25 (2018), p. 273.

[65] David Urbanik. "A Brief Introduction to Schemes and Sheaves". en. In: (), p. 16.

[66] Dmitry Nekrasovski et al. "An Evaluation of Pan & Zoom and Rubber Sheet Navigation with and without an Overview". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 11–20. ISBN: 1-59593-372-7. DOI: 10.1145/1124772.1124775.

[67] Michael S. Crouch, Andrew McGregor, and Daniel Stubbs. "Dynamic Graphs in the Sliding-Window Model". In: *European Symposium on Algorithms*. Springer, 2013, pp. 337–348.

[68] Chia-Shang James Chu. “Time Series Segmentation: A Sliding Window Approach”. In: *Information Sciences* 85.1 (July 1995), pp. 147–173. ISSN: 0020-0255. DOI: 10.1016/0020-0255(95)00021-G.

[69] Charles R Harris et al. “Array Programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362.

[70] Jeff Reback et al. *Pandas-Dev/Pandas: Pandas 1.0.3*. Zenodo. Mar. 2020. DOI: 10.5281/zenodo.3715232.

[71] Matthew Rocklin. “Dask: Parallel Computation with Blocked Algorithms and Task Scheduling”. In: *Proceedings of the 14th Python in Science Conference*. Vol. 126. Citeseer, 2015.

[72] “Retraction (Topology)”. en. In: *Wikipedia* (July 2020).

[73] Eric W. Weisstein. *Homotopy*. en. https://mathworld.wolfram.com/Homotopy.html. Text.

[74] Tim Bienz, Richard Cohn, and Calif.) Adobe Systems (Mountain View. *Portable Document Format Reference Manual*. Citeseer, 1993.

[75] A. Quint. “Scalable Vector Graphics”. In: *IEEE MultiMedia* 10.3 (July 2003), pp. 99–102. ISSN: 1941-0166. DOI: 10.1109/MMUL.2003.1218261.

[76] George S. Carson. “Standards Pipeline: The OpenGL Specification”. In: *SIGGRAPH Comput. Graph.* 31.2 (May 1997), pp. 17–18. ISSN: 0097-8930. DOI: 10.1145/271283.271292.

[77] *Cairographics.Org*. https://www.cairographics.org/.

[78] Maxim Shemanarev. *Anti-Grain Geometry*. https://antigrain.com/.

[79] S. K. Card and J. Mackinlay. “The Structure of the Information Visualization Design Space”. In: *Proceedings of VIZ ’97: Visualization Conference, Information Visualization Symposium and Parallel Rendering Symposium*. Oct. 1997, pp. 92–99. DOI: 10.1109/INFVIS.1997.636792.

[80] *Measurement Scales and Statistics: Resurgence of an Old Misconception. - PsycNET*. https://psycnet.apa.org/doiLanding?doi=10.1037%2F0033-2909.87.3.564.

[81] Christian Remling. *Algebra (Math 5353/5363) Lecture Notes*. Lecture Notes. University of Oklahoma.

[82] H. M. Johnson. "Pseudo-Mathematics in the Mental and Social Sciences". In: *The American Journal of Psychology* 48.2 (1936), pp. 342–351. ISSN: 00029556. DOI: 10.2307/1415754.

[83] M. A. Thomas. *Mathematization, Not Measurement: A Critique of Stevens' Scales of Measurement*. en. SSRN Scholarly Paper ID 2412765. Rochester, NY: Social Science Research Network, Oct. 2014. DOI: 10.2139/ssrn.2412765.

[84] "Connected Space". en. In: *Wikipedia* (Dec. 2020).

[85] Caroline Ziemkiewicz and Robert Kosara. "Embedding Information Visualization within Visual Representation". In: *Advances in Information and Intelligent Systems*. Ed. by Zbigniew W. Ras and William Ribarsky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 307–326. ISBN: 978-3-642-04141-9. DOI: 10.1007/978-3-642-04141-9_15.

[86] Sheelagh Carpendale. *Visual Representation from Semiology of Graphics by J. Bertin*. en.

[87] "Jet Bundle". en. In: *Wikipedia* (Dec. 2020).

[88] Jana Musilová and Stanislav Hronek. "The Calculus of Variations on Jet Bundles as a Universal Approach for a Variational Formulation of Fundamental Physical Theories". In: *Communications in Mathematics* 24.2 (Dec. 2016), pp. 173–193. ISSN: 2336-1298. DOI: 10.1515/cm-2016-0012.

[89] Yael Albo et al. "Off the Radar: Comparative Evaluation of Radial Visualization Solutions for Composite Indicators". In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (Jan. 2016), pp. 569–578. ISSN: 1077-2626. DOI: 10.1109/TVCG.2015.2467322.

[90] Z. Qu and J. Hullman. "Keeping Multiple Views Consistent: Constraints, Validations, and Exceptions in Visualization Authoring". In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (Jan. 2018), pp. 468–477. ISSN: 1941-0506. DOI: 10.1109/TVCG.2017.2744198.

[91] Richard A. Becker and William S. Cleveland. "Brushing Scatterplots". In: *Technometrics* 29.2 (May 1987), pp. 127–142. ISSN: 0040-1706. DOI: 10.1080/00401706.1987.10488204.

[92] Andreas Buja et al. "Interactive Data Visualization Using Focusing and Linking". In: *Proceedings of the 2nd Conference on Visualization '91*. VIS '91. Washington, DC, USA: IEEE Computer Society Press, 1991, pp. 156–163. ISBN: 0-8186-2245-8.

[93] *Dataclasses — Data Classes — Python 3.9.2rc1 Documentation*. https://docs.python.org/3/library/dataclasses.

[94] Kristen B. Gorman, Tony D. Williams, and William R. Fraser. "Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus Pygoscelis)". In: *PLOS ONE* 9.3 (Mar. 2014), e90081. DOI: 10.1371/journal.pone.0090081.

[95] Allison Marie Horst, Alison Presmanes Hill, and Kristen B Gorman. *Palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. Manual. 2020. DOI: 10.5281/zenodo.3960218.

[96] Muhammad Chenariyan Nakhaee. *Mcnakhaee/Palmerpenguins*. Jan. 2021.

[97] John M Chambers et al. *Graphical Methods for Data Analysis*. Vol. 5. Wadsworth Belmont, CA, 1983.

[98] F. Beck. "Software Feathers Figurative Visualization of Software Metrics". In: *2014 International Conference on Information Visualization Theory and Applications (IVAPP)*. Jan. 2014, pp. 5–16.

[99] Lydia Byrne, Daniel Angus, and Janet Wiles. "Figurative Frames: A Critical Vocabulary for Images in Information Visualization". In: *Information Visualization* 18.1 (Aug. 2017), pp. 45–67. ISSN: 1473-8716. DOI: 10.1177/1473871617724212.

[100] Paul Murrell. *R Graphics, Third Edition*. 3rd. Chapman & Hall/CRC, 2018. ISBN: 1-4987-8905-6.

[101] Robert B Haber and David A McNabb. "Visualization Idioms: A Conceptual Model for Scientific Visualization Systems". In: *Visualization in scientific computing* 74 (1990), p. 93.

[102] Charles D Hansen and Chris R Johnson. *Visualization Handbook*. Elsevier, 2011.

[103] Leland Wilkinson and Michael Friendly. "The History of the Cluster Heat Map". In: *The American Statistician* 63.2 (May 2009), pp. 179–184. ISSN: 0003-1305. DOI: 10.1198/tas.2009.0033.

[104] Toussaint Loua. *Atlas Statistique de La Population de Paris*. J. Dejey & cie, 1873.

[105] C. Heine et al. "A Survey of Topology-Based Methods in Visualization". In: *Computer Graphics Forum* 35.3 (June 2016), pp. 643–667. ISSN: 0167-7055. DOI: 10.1111/cgf.12933.

[106] James O Ramsay. *Functional Data Analysis*. Wiley Online Library, 2006.

[107] Enrico Bertini, Andrada Tatu, and Daniel Keim. "Quality Metrics in High-Dimensional Data Visualization: An Overview and Systematization". In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (2011), pp. 2203–2212.

[108] Bartosz Milewski. "Category Theory for Programmers". en. In: (), p. 498.