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

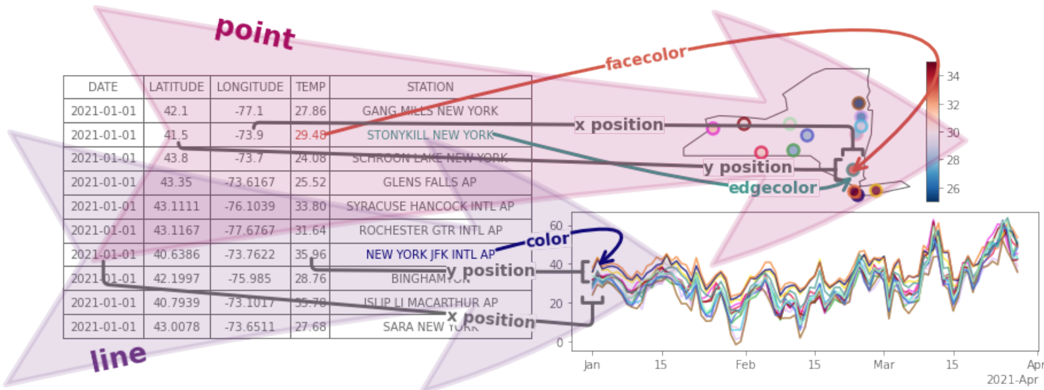


Figure 1: Visualizations are made up of transformations from data into visual representation. These functions are are both transformations of individual data values to visual representation, such as date to x position or latitude to y position, and the assembly of all these transformations into a visual mark, such as a line or point. The same variable can be mapped in different ways, for example line is mapped to a color in the scatter plot and to y position in the line plot.

Visualization is the transformation of data into visual representation. As illustrated by Figure 1, these translations are both at the level of the individual variable and the entire record. In the case of the scatter plot, the latitude and longitude are encoded as the x and y position, respectively, while the temperature and station are represented by the face and edge colors. A row in the table is collectively encoded as a point mark. None of these encodings are fixed, as evidenced by temperature being translated into the y value in the case of the line plot. The station is now the source of the color of the entire line, and the date is the x position. As with scatter, the encodings of the individual transformations, which again are on values from the same record in the table, are composited into a line mark. It is these raw transformations from data space to visualization space that are implemented by building block level visualization libraries, named as such because the functions provided by the library can be composited in any number of ways to yield visualizations [1]. We propose that like physical building blocks, building block libraries must provide a collection of well defined pieces that can be composed in whichever ways the blocks fit together. We specify that a valid visualization block is a structure preserving transformation from data to visual space, and we define structure in terms of *continuity* and *equivariance*. We then use this model to develop a design specification for the components of a building block visualization library. The notion of self contained, inherently modular, building blocks lends itself naturally to a functional paradigm of visualization [2]. We adopt a functional model for a redesign because the lack of side effects means functional architecture can be evaluated for correctness, functional programs tend to be shorter and clearer, and are well suited to distributed, concurrent, and on demand tasks[3].

This work is strongly motivated by the needs of the Matplotlib[4, 5] visualization library. One of the most widely used visualization libraries in Python, since 2002 new components and features have been added in a some what adhoc, sometimes hard to maintain, manner. Particularly, each new 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 then expressed in the API for that component. In turn, this yields an inconsistent API for interfacing with the data, for example when updating streaming visualizations or constructing dashboards[6]. 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 that these issues can be ameliorated via 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 formalisms of the notion that visualization is structure preserving maps from data to visual representation, and many visualization libraries that attempt to preserve 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 to propose a new model of the constraints visual transformations must satisfy such that they can be composed to produce visualize representations that can be considered equivalent to the data being represented.

2.1 Structure:

Visual representations of data, by definition, reflect something of the underlying structure and semantics[7], 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 [8]. 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*.

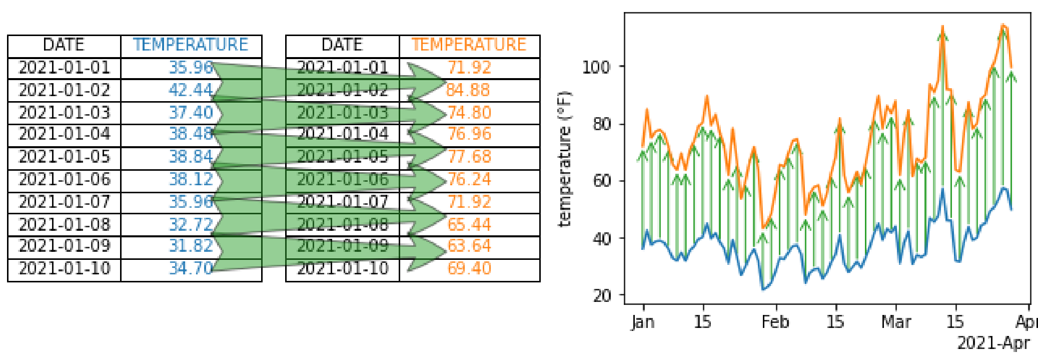
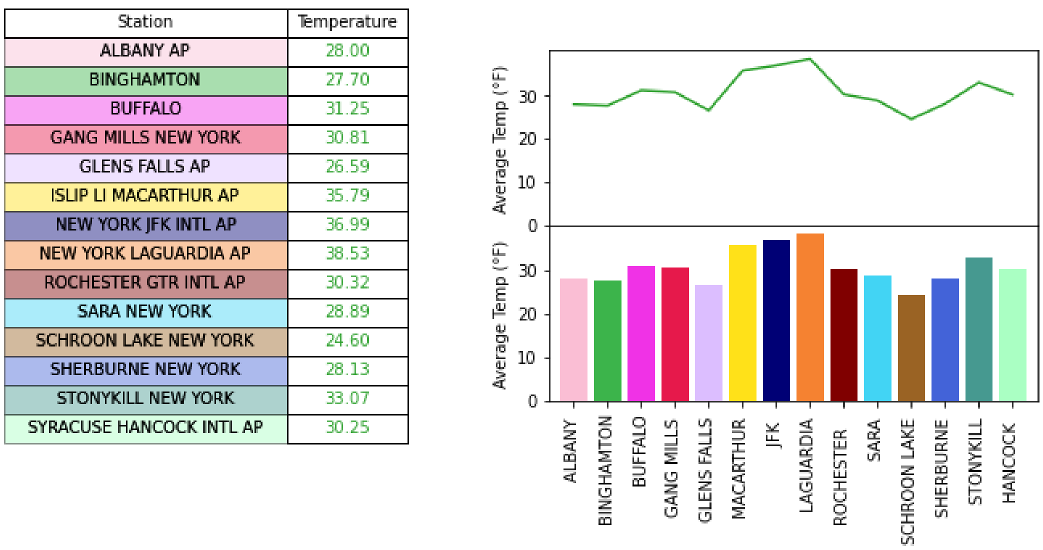


Figure 2: The data in blue is scaled by a factor of two, yielding the data in orange. To preserve *equivariance*, the blue line plot representation of the unscaled data is also scaled by a factor of two, yielding the orange line plot that is equivalent to the scaled data.

Bertin proposes that there are classes of visual encodings-such as position, shape, color, and texture-that when mapped to from specific types of measurement, quantitative or qualitative, will preserve the properties of that measurement type. For example, in Figure 2, the data and visual representation are scaled by equivalent factors of two, resulting in the change illustrated in the shift from blue to orange data and lines. 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[10, 11]. The algebraic model of visualization proposed by Kindlmann and Scheidegger uses equivariance to refer generally to invertible binary transformations[12], which are mathematical groups [13]. 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 explicitely include continuity.



Station	Temperature
ALBANY AP	28.00
BINGHAMTON	27.70
BUFFALO	31.25
GANG MILLS NEW YORK	30.81
GLENS FALLS AP	26.59
ISLIP LI MACARTHUR AP	35.79
NEW YORK JFK INTL AP	36.99
NEW YORK LAGUARDIA AP	38.53
ROCHESTER GTR INTL AP	30.32
SARA NEW YORK	28.89
SCHROON LAKE NEW YORK	24.60
SHERBURNE NEW YORK	28.13
STONYKILL NEW YORK	33.07
SYRACUSE HANCOCK INTL AP	30.25

Figure 3: The line plot does not preserve *continuity* because it implies that the average temperature at each station lie along a 1D continuous line, while the bar plot preserves *continuity* by representing the average temperatures at each station as the discrete values they are.

Bertin proposes that the visual encodings be composited into 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. In Figure 3, the line plot does not preserve continuity because the line connecting the discrete categories implies that the frequency of weather events is sampled from a continuous interval and the categories are points on that interval. But, when the continuity is preserved, as in the bar chart, then the graphic has not introduced new structure into the data.

Structure

continuity How records in the dataset are connected to each other, e.g. discrete rows, neworked 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.

The notion that a graphic should be equivalent to the data has been expressed in a variety of ways. Informally, Norman's Naturalness Principal[14] 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[15]. expressing, as defined by Mackinlay, is a measure how much of the mathematical structure in the data that can be expressed in the visualizations; for example that ordered variables can be mapped into ordered visual elements. We propose that a graphic is an equivalent representation of the data when *continuity* and *equivariance* are preserved.

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

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

One of the reasons we developed a new formalism rather than adopting the architecture of an existing library is that most information visualization software design patterns, as categorized by Heer and Agrawala[16], are tuned to very specific data structures. These libraries can often assume the expected data structure because they are domain specific, and that is the common data structure in that domain. For users who generally work in one domain, such as the data, networks, or graphs shown in Figure 4, this well defined data space (and corresponding visual space[17]) often yields a very coherant user experience[18]. But, for developers who want to build new visualizations on top of these libraries, they must work around the existing assumptions, sometimes in ways that break the model the libraries are developed around.

For example, many domain specific libraries integrate computation into the visualization, for example libraries based that assume all data is a relational database. This assumption is core to tools influenced by APT, such as Tableau[19–21] and the Grammar of Graphics[22], such as ggplot[23], protovis[24], vega[25] and altair[26]. Since these libraries represent all data as a table, and computations on tables are fairly well defined[27], they can include computations on the table with a fair bit of confidence that the computation is accurate. Since most computations are specific to domains, general purpose block libraries can not make this assumption; instead a goal of this model is to identify which computations are specifically part of the visual encoding - for example mapping data to a color-and which are manipulations on the data. Disentangling the computation from the visual transforms allows us to determine whether the visualization library needs to handle them or if they can be more efficiently computed by the data container.

A different class of user facing tools are those that support images, such as ImageJ[28] or Napari[29]. These tools often have some support for visualizing non image components of a complex data set, but mostly in service to the image being visualized. These tools are ill suited for general purpose libraries that need to support data other than images because the architecture is oriented towards building plugins into the existing system [30] where the image is the core data structure. Even the digital humanities oriented ImageJ macro ImagePlot[31], which supports some non-image aggregate reporting charts, is still built around image data as the primary input. The need to visualize and manipulate graphs has spawned tools like Gephi[32], Graphviz[33], and Networkx[34]. As with tables and images, extending network libraries to work with other types of data either require breaking their internal model of how data is structured and what transformations of the data are allowable or growing a model for other types of data structures alongside the network model.

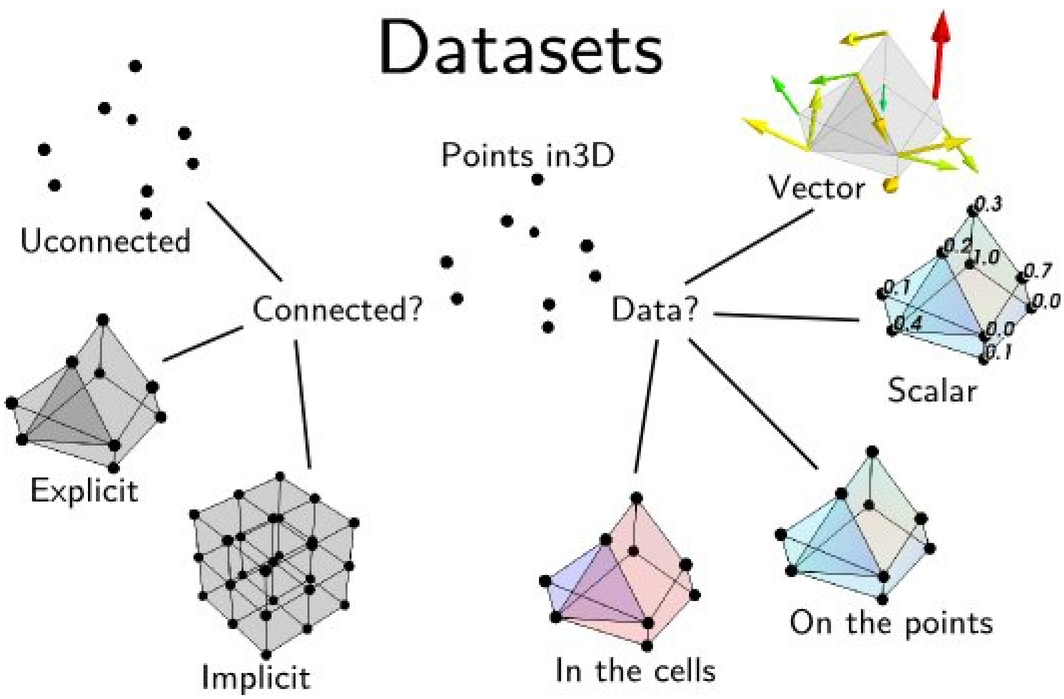


Figure 5: One way to describe data is by the connectivity of the points in the dataset. A database for example is often discrete unconnected points, while an image is an implicitely connected 2D grid. This image is from the Data Representation chapter of the MayaVi 4.7.2 documentation.[35]

Many building block libraries carry multiple models of data internally because they cannot assume a data structure. Algorithms are designed such that the structure of data is assumed, as described in Tory and Möller's taxonomy [**ToryRethinkingVisualization2004**], and by definition building block libraries try to provide the components to build any sort of visualization. Matplotlib, D3[36], and VTK [**geveci2012vtk**, 37] and its derivatives such as MayaVi[38] and extensions such as ParaView[39] and the infoviz themed Titan[40]. Where GoG and ImageJ type libraries have coherant APIs for their visualization tools because the data structure is the same, the APIs for visualizations in Matplotlib, D3, and VTK are significantly dependent on the structure of the data it expects. VTK has codified this in terms of *continuity* based data representations, as illustrated in figure 5. This API choice can lead to visualizations that break *continuity* when fed into visualizations with different assumptions about structure. The lack of consistent data model can also mean no consistent way of updating the data and therefore no way of guaranteeing that the views are in sync, in visualizations that consistent of multiple views of the same datasource, such as dashboards[6, 41]. To resolve this issue, our functional model takes as input a structure aware data abstraction general enough to provide a common interface for many different types of visualization.

2.3 Data

One such general abstraction are fiber bundles, which Butler proposed as a core data structure for visualization because they encode data continuity separately from the variable properties and are flexible enough to support discrete and ND continuous datasets [42, 43]. Since Butler's model lacks a robust way of describing variables, we can encode a schema like description of the data in the fiber bundle by folding in Spivak's topological description of data types [44, 45]. In this work we will refer to the points of the dataset as *records* to indicate that a point can be a vector of heterogenous elements. Each *component* of the record is a single object, such as a temperature measurement, a color value, or an image. We also generalize *component* to mean all objects in the dataset of a given type, such as all temperatures or colors or images. The way in which these records are connected is the *connectivity*, *continuity*, or more generally *topology*.

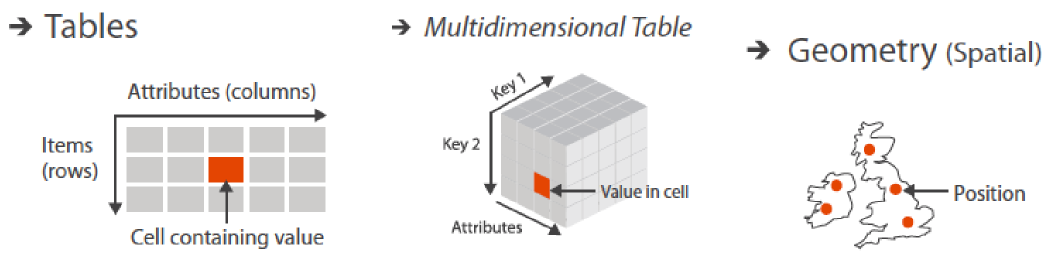


Figure 6: Values in a dataset have keys associated with them that describe where the value is in the dataset. These keys can be indexers or semantically meaningful;for example, in a table the keys are the variable name and the row ID. In the data cube, the keys is the row, column, and cell ID, and in the map the key is the position in the grid. Image is figure 2.8 in Munzner's Visualization Analysis and Design[46]

The *continuity* can often be described by some variables in the dataset; this is formalized Munzner's notion of metadata as *keys* into the data structure that return associated *values*[47]. As shown in Figure 6, keys can be labeled indexes, such as the attribute name and row ID, or physical entities such as locations on a map. We propose that information rich metadata are part of the components and instead the values are keyed on coordinate free structural ids. In contrast to Munzner's model where the semantic meaning of the key is tightly coupled to the position of the value in the dataset, our model considers keys to be a pure reference to topology. 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 - such as a measure of plant growth given variations in water, sunlight, and time - without having to assume any one variable is inducing the change in growth. For building block library developers, this means the components are able to fully traverse the data structures without having to know anything about the values or the semantic meaning of the structure. Since these components are by design *equivariant* and *continuity* preserving, domain specific library developers in different domains that both rely on the same continuity, for example 2D continuity, can then safely reuse the components to build tools that can safely make domain specific assumptions.

2.4 Contribution

The contribution of this work is

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 monoid action 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

3 Topological Artist Model

As discussed in the introduction, visualization is generally defined as structure preserving maps from a data object to a graphic object. In order to formalize this statement, we describe the connectivity of the records using topology and define the structure on the components in terms of the monoid actions on the component types. By formalizing structure in this way, we can evaluate the extent to which a visualization preserves the structure of the data it is representing and build structure preserving visualization tools. We introduce the notion of an artist $\mathscr{A}$ as an equivariant map from data to graphic

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

that carries a homomorphism of monoid actions $\varphi : M \to M'$ [48], which are discussed in detail in section 3.1.2. Given M on data $\mathscr{E}$ and M' on graphic $\mathscr{H}$, we propose that artists $\mathscr{A}$ are equivariant maps

$$\mathscr{A}(m \cdot r) = \varphi(m) \cdot \mathscr{A}(r) \tag{2}$$

such that applying a monoid action $m \in M$ to the data $r \in \mathscr{E}$ input to $\mathscr{A}$ is equivalent to applying a monoid action $\varphi(M) \in M'$ to the graphic $A(r) \in \mathscr{H}$ output of the artist.

We model the data $\mathscr{E}$, graphic $\mathscr{H}$, and intermediate visual encoding $\mathscr{V}$ stages of visualization as topological structures that encapsulate types of variables and continuity. To explain which structure the artist is preserving, we first describe how we model data (3.1), graphics (3.2), and intermediate visual characteristics (3.3) as fiber bundles. We then discuss the equivariant maps between data and visual characteristics (3.3.2) and visual characteristics and graphics (3.3.3) that make up the artist.

3.1 Data Space E

Building on Butler's proposal of using fiber bundles as a common data representation structure for visualization data[42, 43], a fiber bundle is a tuple (E, K, π, F) defined by the projection map π

$$F \hookrightarrow E \xrightarrow{\pi} K \tag{3}$$

that binds the components of the data in F to the continuity represented in K. The fiber bundle models the properties of data component types F (3.1.1), the continuity of records K (3.1.3), the collections of records τ (3.1.4), and the space E of all possible datasets with these components and continuity.

By definition fiber bundles are locally trivial[49, 50], meaning that over a localized neighborhood we can dispense with extra structure on E and focus on the components and continuity. We use fiber bundles as the data model because they are inclusive enough to express all the types of data described in section 2.3.

3.1.1 Variables in Fiber Space F

To formalize the structure of the data components, we use notation introduced by Spivak [45] that binds the components of the fiber to variable names. This allows us to describe the components in a schema like way. 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 $\mathbb{U}_\sigma$

$$
\begin{tikzcd}
\mathbb{U}_\sigma \arrow[r] \arrow[d, "\pi_\sigma"'] & \mathbb{U} \arrow[d, "\pi"] \\
C \arrow[r, "\sigma"'] & \mathbf{DT}
\end{tikzcd}
\tag{4}
$$

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

$$F = \mathbb{U}_{\sigma(c)} = \mathbb{U}_T \tag{5}$$

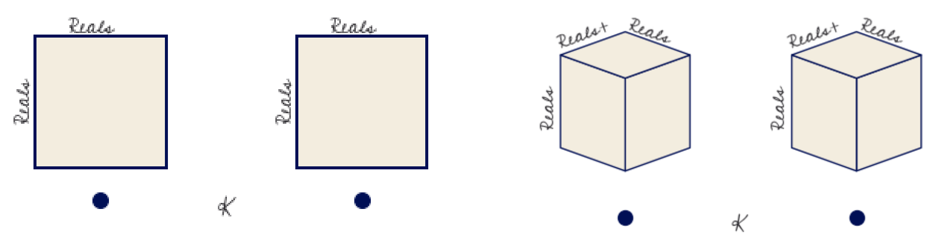
where σ is the schema binding variable name c to its datatype T. A dataset with multiple variables has a fiber that is the cartesian cross product of $\mathbb{U}_\sigma$ applied to all the columns:

$$F = \mathbb{U}_{\sigma(c_1)} \times \ldots \mathbb{U}_{\sigma(c_i)} \ldots \times \mathbb{U}_{\sigma(c_n)} \tag{6}$$

which is equivalent to

$$F = F_0 \times \ldots \times F_i \times \ldots \times F_n \tag{7}$$

which allows us to decouple F into components F_i.



(a)

(b)

Figure 7: These two datasets have the same base space K of discrete points, but figure 7a has fiber $F = \mathbb{R} \times \mathbb{R}$ which is (time, temperature) while figure 7b has fiber $\mathbb{R} \times \mathbb{R}^{+} \times \mathbb{R}$ which is (time, wind=(speed, direction))

For example, the data in figure 7a is a pair of times and °K temperature measurements taken at those times. Time is a positive number of type `datetime` which can be resolved to floats $\mathbb{U}_{\texttt{datetime}} = \mathbb{R}$. Temperature values are real positive numbers $\mathbb{U}_{\texttt{float}} = \mathbb{R}^+$. The fiber is

$$\mathbb{U} = \mathbb{R} \times \mathbb{R}^+ \tag{8}$$

where the first component F_0 is the set of values specified by ($c = \textit{time}$, $T = \texttt{datetime}$, $\mathbb{U}_\sigma = \mathbb{R}$) and F_1 is specified by ($c = \textit{temperature}$, $T = \texttt{float}$, $\mathbb{U}_\sigma = \mathbb{R}^+$) and is the set of values $\mathbb{U}_\sigma = \mathbb{R}^+$. In figure 7b, temperature is replaced with wind. This wind variable is of type `wind` and has two components speed and direction $\{(s, d) \in \mathbb{R}^2 \mid 0 \leq s, 0 \leq d \leq 360\}$. Therefore, the fiber is

$$F = \mathbb{R}^+ \times \mathbb{R}^2 \tag{9}$$

such that F_1 is specified by ($c = \textit{wind}$, $T = \texttt{wind}$, $\mathbb{U}_\sigma = \mathbb{R}^2$). As illustrated in figure 7, Spivak's framework provides a consistent way to describe potentially complex components of the input data.

3.1.2 Measurement Scales: Monoid Actions

Implementing expressive visual encodings requires formally describing the structure on the components of the fiber, which we define by the actions of a monoid on the component. In doing so, we specify the properties of the component that must be preserved in a graphic representation. While structure on a set of values is often described algebraically as operations or through the actions of a group, for example Steven's scales [51], we generalize to monoids to support more component types. Monoids are also commonly found in functional programming because they specify compositions of transformations [52, 53].

A monoid [54] M is a set with an associative binary operator $* : M \times M \to M$. A monoid has an identity element $e \in M$ such that $e * a = a * e = a$ for all $a \in M$. As defined on a component of F, a left monoid action [55, 56] of M_i is a set F_i with an action

$\bullet : M \times F_i \to F_i$ with the properties:

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

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

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

$$M = M_0 \times \ldots \times M_i \times \ldots \times \ldots M_n \tag{10}$$

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

Steven's described the measurement scales[51, 57] in terms of the monoid actions on the measurements: nominal data is permutable, ordinal data is monotonic, interval data is translatable, and ratio data is scalable [58]. For example, given an arbitrary interval scale fiber component ($c = \textit{temperature}$, $T =$ `float`, $\mathbb{U}_\sigma = \mathbb{R}$) with with arbitrary monoid translation actions chosen for this example:

- monoid operator addition $* = +$
- monoid operations: $f : x \mapsto x + 1^{\circ}C$, $g : x \mapsto x + 2^{\circ}C$
- monoid action operator composition $\bullet = \circ$

By structure preservation, we mean that monoid actions are composable. For the translation actions described above on the temperature fiber, this means that they satisfy the condition

$$\begin{tikzcd} \mathbb{R} \arrow[d, "x+1^{\circ}"'] \arrow[dr, "(x+1^{\circ}C)\circ(x+2^{\circ}C)"] & \\ \mathbb{R} \arrow[r, "x+2^{\circ}C"'] & \mathbb{R} \end{tikzcd} \tag{11}$$

where $1^{\circ}C$ and $2^{\circ}C$ are valid distances between two temperatures x. What this diagram means is that either the fiber could be shifted by $1^{\circ}C$ (vertical line) then by $2^{\circ}C$ (horizontal), or the two shifts could be combined such that in this case the fiber is shifted by $3^{\circ}C$ (diagonal) and these two paths yield the same temperature.

While many component types will be one of the measurement scale types, we generalize to monoids specifically for the case of partially ordered set. Given a set $W = \{mist, drizzle, rain\}$, then the map $f : W \to W$ defined by

1. $f(rain) = drizzle$,
2. $f(drizzle) = mist$
3. $f(mist) = mist$

is order preserving such that $mist \leq drizzle \leq rain$ but has no inverse since $drizzle$ and $mist$ go to the same value $mist$. Therefore order preserving maps do not form a group, and instead we generalize to monoids to support partial order component types. Defining the monoid actions on the components serves as the basis for identifying the invariance[**kindlmann2014algebraic**] that must be preserved in the visual representation of the component. We propose equivariance of monoid actions individually on the fiber to visual component maps and on the graphic as a whole.

3.1.3 Continuity of the Data K

The base space K is way to express how the records in E are connected to each other, for example if they are discrete points or if they lie in a 2D continous surface. Connectivity type is assumed in the choice of visualization, for example a line plot implies 1D continuous data, but an explicit representation allows for verifying that the topology of the graphic representation is equivalent to the topology of the data.

Figure 8: The topological base space K encodes the connectivity of the data space, for example if the data is independent points or on a plane or a sphere

As illustrated in figure 8, K is akin to an indexing space into E that describes the structure of E. K can have any number of dimensions and can be continuous or discrete.

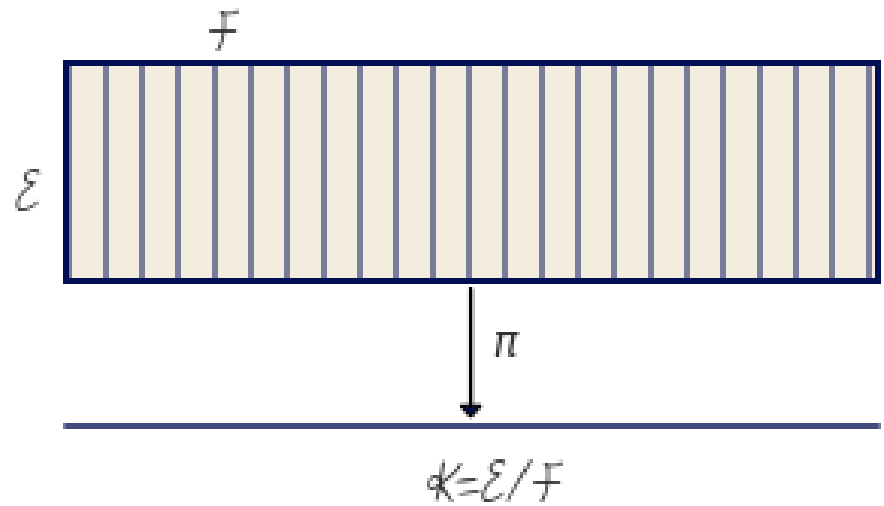
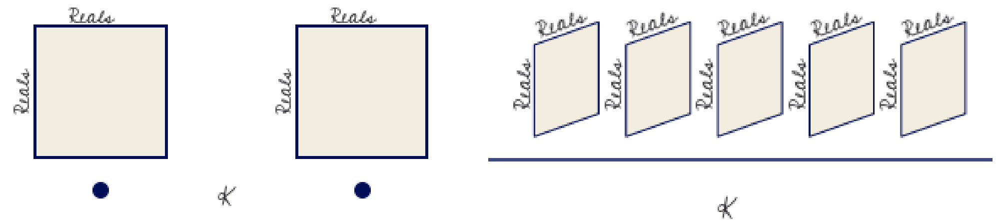


Figure 9: The base space E is divided into fiber segments F. The base space K acts as an index into the records in the fibers.

Formally K is the quotient space [59] of E meaning it is the finest space[60] such that every $k \in K$ has a corresponding fiber F_k[59]. In figure 9, 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 $[0, 1]$. As with fibers and monoids, we can decompose the total space into components $\pi : E_i \to K$ where

$$\pi : E_1 \oplus \ldots \oplus E_i \oplus \ldots \oplus E_n \to K \tag{12}$$

which is a decomposition of F. The K remains the same because the connectivity of records does not change just because there are fewer elements in each record.



(a)

(b)

Figure 10: These two datasets have the same (time, temperature) fiber. In figure 10a the total space E is discrete over points $k \in K$, meaning the records in the fiber are also discrete. In figure 10b E lies over the continuous interval K, meaning the records in the fiber are sampled from a continuous space.

The datasets in figure 10 have the same fiber of (temperature, time). In figure 10a the fibers lie over discrete K such that the records in the datasets in the fiber bundles are discrete. The same fiber in figure 10b lies over a continuous interval K such that the records are samples from a continuous function defined on K. By encoding this continuity in the model as K the data model now explicitly carries information about its structure such that the implicit assumptions of the visualization algorithms are now explicit. The explicit topology is a concise way of distinguishing visualizations that appear identical, for example heatmaps and images.

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 [45], 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

$$\begin{tikzcd} F \arrow[r, hook] & E \arrow[d, "\pi"', shift right] \\ & K \arrow[u, "\tau"', shift right, bend right] \end{tikzcd} \tag{13}$$

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 is

$$\tau(k) = (k, (g_{F_0}(k), \ldots, g_{F_n}(k))) \tag{14}$$

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, we can decompose τ as

$$\tau = (\tau_0, \ldots, \tau_i, \ldots, \tau_n) \tag{15}$$

where each section τ_i is a variable or set of variables. This allows for accessing the data component wise in addition to accessing the data in terms of its location over K.

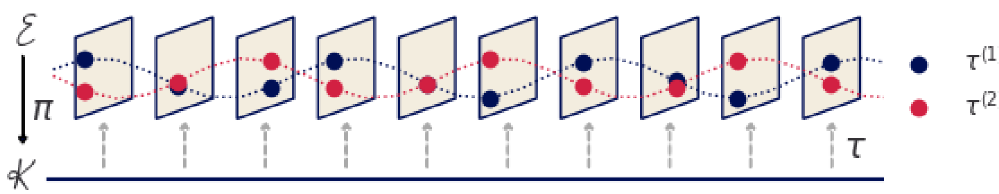


Figure 11: Fiber (time, temperature) with an interval K basespace. The sections $\tau^{(1)}$ and $\tau^{(2)}$ are constrained such that the time variable must be monotonic, which means each section is a timeseries of temperature values. They are included in the global set of sections $\tau^{(1)}, \tau^{(2)} \in \Gamma(E)$

In the example in figure 11, the fiber is (*time*, *temperature*) as described in figure 7 and the base space is the interval K. The section $\tau^{(1)}$ resolves to a series of monotonically increasing in time records of (time, temperature) values. Section $\tau^{(2)}$ returns a different timeseries of (time, temperature) values. Both sections are included in the global set of sections $\tau^{(1)}, \tau^{(2)} \in \Gamma(E)$.

3.1.5 Applications to Data Containers

This model provides a common formalism for widely used data containers without sacrificing the semantic structure embedded in each container. For example, the section can be any instance of a univariate numpy array[61] that stores an image. This could be a section of a fiber bundle where K is a 2D continuous plane and the F is $(\mathbb{R}^3, \mathbb{R}, \mathbb{R})$ where $\mathbb{R}^3$ is color, and the other two components are the x and y positions of the sampled data in the image. This position information is already implicitely encoded in the array as the index and the resolution of the image being stored.Instead of an image, the numpy array could also store a 2D discrete table. The fiber would 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.

There are also many types of labeled containers that can richly be described in this framework because of the schema like structure of the fiber. For example, a pandas series which stores a labeled list, or a dataframe[62] which stores a relational table. A series could store the values of $\tau^{(1)}$ and a second series could be $\tau^{(2)}$. We could also fatten the fiber to hold two temperature series, such that a section would be an instance of a dataframe with a time column and two temperature columns. While the series and dataframe explicitly have a time index column, they are components in our model and the index is assumed to be data independent references such as hashvalues, virtual memory locations, or random number keys.

Where this model particularly shines are N dimensional labeled data structures. For example, an xarray[63] data that stores temperature field could have a K that is a continuous volume and the components would be the temperature and the time, latitude, and longitude the measurements were sampled at. A section can also be an instance of a distributed data container, such as a dask array [64]. As with the other containers, K and F are defined in terms of the index and dtypes of the components of the array. Because our framework is defined in terms of the fiber, continuity, and sections, rather than the exact values of the data, our model does not need to know what the exact values are until the renderer needs to fill in the image.

3.2 Graphic Space H

We introduce a graphic bundle to hold the essential information necessary to render a graphical design constructed by the artist. As with the data, we can represent the target graphic as a section ρ of a bundle (H, S, π, D). The graphic bundle H consists of a base S(3.2.1) that is a thickened form of K a fiber D(3.2.2) that is an idealized display space, and sections ρ(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 an infinite resolution 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

$$(x,\, y,\, r,\, g,\, b) \in D \tag{16}$$

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

Just as the K encodes the connectivity of the records in the data, we propose an equivalent S that encodes the connectivity of the rendered elements of the graphic. For example, consider a S that is mapped to the region of a 2D display space that represents K. For some visualizations, K may be lower dimension than S. For example, a point that is 0D in K cannot be represented on screen unless it is thickened to 2D to encode the connectivity of the pixels that visually represent the point. 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. We introduce S to thicken K in a way which preserves the structure of K.

Formally, we require that K be a deformation retract[65] of S so that K and S have the same homotopy. The surjective map $\xi : S \to K$

$$\begin{tikzcd} E \arrow[d, "\pi"'] & H \arrow[d, "\pi"'] \\ K & S \arrow[l, "\xi"'] \end{tikzcd} \tag{17}$$

goes from region $s \in S_k$ to its associated point s. 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.

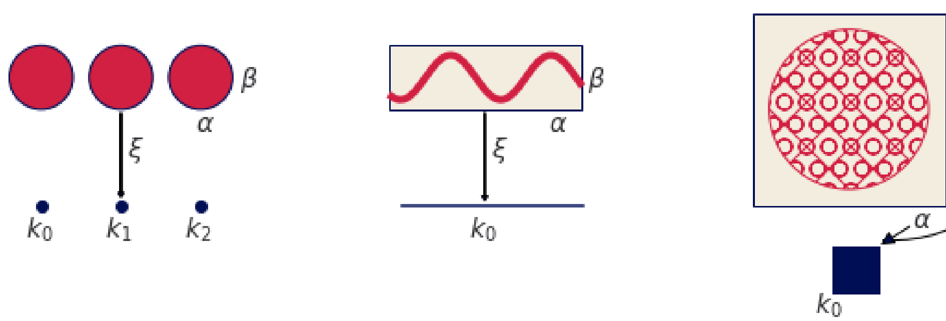


Figure 12: The scatter and line graphic base spaces have one more dimension of continuity than K so that S can encode physical aspects of the glyph, such as shape (a circle) or thickness. The image has the same dimension in S as in K.

When K is discrete points and the graphic is a scatter plot, each point $k \in K$ corresponds to a 2D disk S_k as shown in figure 12. In the case of 1D continuous data and a line plot, the region β over a point α_i specifies the thickness of the line in S for the corresponding τ on k. The image has the same dimensions in data space and graphic space such that no extra dimensions are needed in S.

The mapping function ξ provides a way to identify the part of the visual transformation that is specific to the the connectivity of the data rather than the values; for example it is common to flip a matrix when displaying an image. The ξ mapping is also 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

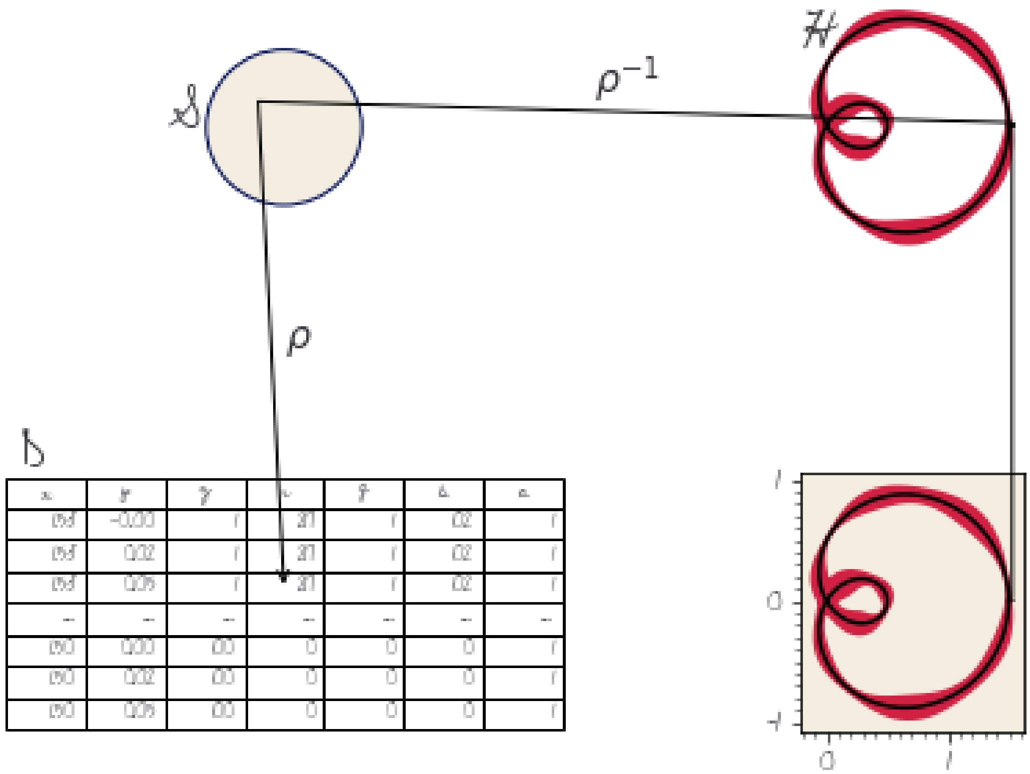


Figure 13: To render a graphic, a pixel p is selected in the display space, which is defined in the same coordinates as the x and y components in D. The inverse mapping $\rho_{xy}{}^{(}p)$ returns a region $S_p \subset S$. $\rho(S_p)$ returns the list of elements $(x,\, y,\, r,\, g,\, b) \in D$ that lie over S_p. The integral over the $(r,\, g,\, b)$ elements is the color of the pixel.

This section describes how we go from a graphic in an idealized prerender space to a rendered image, where the graphic is the section $\rho : S \to H$. 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.

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 $S_p = {\rho_{xy}}^{-1}(p)$ is a region $S_p \subset S$. To compute the color, we integrate on S_p

$$r_p = \iint\limits_{S_p} \rho_r(s) ds^2 \tag{18}$$

$$g_p = \iint\limits_{S_p} \rho_g(s) ds^2 \tag{19}$$

$$b_p = \iint\limits_{S_p} \rho_b(s) ds^2 \tag{20}$$

As shown in figure 13, a pixel p in the output space is selected and inverse mapped into the corresponding region $S_p \subset S$. This triggers a lookup of the ρ over the region S_p, which yields the set of elements in D that specify the (r, g, b) values corresponding to the region p. The color of the pixel is then obtained by taking the integral of $\rho_{rgb}(S_p)$.

In general, ρ is an abstraction of rendering. In very broad strokes ρ can be a specification such as PDF[66], SVG[67], or an openGL scene graph[68]. Alternatively, ρcan be a rendering engine such as cairo[69] or AGG[**AntiGrainGeometry**]. Implementation of ρ is out of scope for this work,

3.3 Artist

We propose that the transformation from data to visual representation can be described as a structure preserving map from one topological space to another. We name this map the artist as that is the analogous part of the Matplotlib[5] architecture that builds visual elements. The topological artist A is a monoid equivariant sheaf map from the sheaf on a data bundle E which is $\mathcal{O}(E)$ to the sheaf on the graphic bundle H, $\mathcal{O}(H)$.

$$A : \mathcal{O}(E) \to \mathcal{O}(H) \tag{21}$$

Sheafs are a mathematical object with restriction maps that define how to glue τover local neighborhoods $U \subseteq K$, discussed in section **??**, such that the A maps are consistent over continuous regions of K. While Acan usually construct graphical elements solely with the data in τ, some visualizations, such as line, may also need some finite number n of derivatives, which is captured by the jet bundle $\mathcal{J}^n$ [70, 71] with $\mathcal{J}^0(E) = E$. In this work, we at most need $\mathcal{J}^2(E)$ which is the value at τ and its first and second derivatives; therefore the artist takes as input the jet bundle $E' = \mathcal{J}^2(E)$.

Specifically, $A$is the equivariant map from E' to a specific graphic $\rho \in \Gamma(H)$

$$\begin{tikzcd}
E' \arrow[r, "\nu"] \arrow[dr, "\pi"'] & V \arrow[d, "\pi"] & \xi^*V \arrow[l, "\xi^*"'] \arrow[r, "Q"] \arrow[d, "\xi^*\pi"'] & H \arrow[dl, "\pi"] \\
& K & S \arrow[l, "\xi"] &
\end{tikzcd} \tag{22}$$

where the input can be point wise $\tau(k) \mid k \in K$. The encoders $\nu : E' \to V$ convert the data components to visual components(3.2.2). The continuity map $\xi : S \to K$ then pulls back the visual bundle V over S(3.3.2). Then the assembly function $Q : \xi^*V \to H$ composites the fiber components of ξ^*V into a graphic in H(3.3.3). This functional decomposition of the visualization artist facilitates building reusable components at each stage of the transformation because the equivariance constraints are defined on ν, Q, and ξ.

3.3.1 Visual Fiber Bundle V

We introduce a visual bundle V to store the visual representations the artist needs to assemble into a graphic. The visual bundle (V, K, π, P) has section $\mu : V \to K$ that resolves to a visual variable in the fiber P. The visual bundle V is the latent space of possible parameters of a visualization type, such as a scatter or line plot. We define P in terms of the parameters of a visualization libraries compositing functions; for example table 1 is a sample of the fiber space for Matplotlib [4].

$\boldsymbol{\nu_i}$	$\boldsymbol{\mu_i}$	$\boldsymbol{codomain(\nu_i) \subset P_i}$
position	x, y, z, theta, r	$\mathbb{R}$
size	linewidth, markersize	$\mathbb{R}^+$
shape	markerstyle	$\{f_0, \dots, f_n\}$
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

A section μ is a tuple of visual values that specifies the visual characteristics of a part of the graphic. For example, given a fiber of $\{xpos, ypos, color\}$ one possible section could be $\{.5, .5, (255, 20, 147)\}$. The $codomain(\nu_i)$ determines the monoid actions on P_i. These fiber components are implicit in the library, by making them explicit as components of the fiber we can build consistent definitions and expectations of how these parameters behave.

3.3.2 Visual Encoders ν

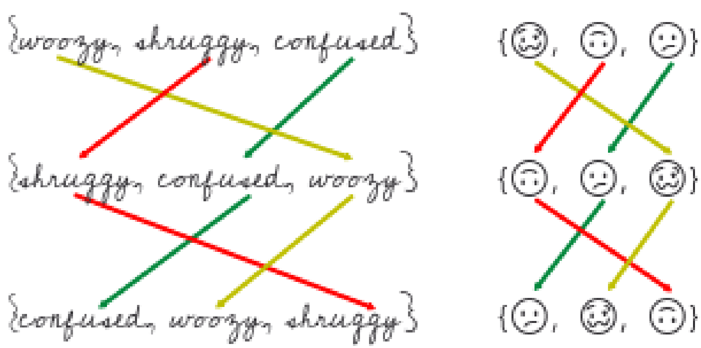


Figure 14: In this artist, ν maps the strings to the emojis. This ν is equivariant because the monoid actions (which are represented by the colored arrows) are the same on both the τ input and μ output sets.

As introduced in section 2.1, there are many ways to visually represent data components. We define the visual transformers ν

$$\{\nu_0, \dots, \nu_n\} : \{\tau_0, \dots, \tau_n\} \mapsto \{\mu_0, \dots, \mu_n\} \tag{23}$$

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 section 3.1.2, we choose monoid actions as the basis for equivariance because they define the structure on the fiber components.

A validly constructed ν is one where the diagram of the monoid transform m commutes such that

$$\begin{tikzcd} E_i \arrow[r, "\nu_i"] \arrow[d, "m_r"'] & V_i \arrow[d, "m_v"] \\ E_i \arrow[r, "\nu_i"] & V_i \end{tikzcd} \tag{24}$$

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

$$\nu_i(m_r(E_i)) = \varphi(m_r)(\nu_i(E_i)) \tag{25}$$

such that φ maps a monoid action on data to a monoid action on visual elements. However, 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 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 $(m, v) \to \varphi(m) \bullet v$. Therefore, 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.

On example of an equivariant ν is illustrated in figure 14, which is a mapping from **Strings** to symbols. The data is an example of a Steven's nominal measurement set, which is defined as having on it permutation group actions

$$\text{if } r_1 \neq r_2 \text{ then } \nu(r_1) \neq \nu(r_2) \tag{26}$$

such that shuffling the words must have an equivalent shuffle of the symbols they are mapped to. This is illustrated in the identical actions, represented by the colored arrows, on the words and emojis. To preserve ordinal and partial order monoid actions, ν must be a monotonic function such that given $r_1, r_2 \in E_i$,

$$\text{if } r_1 \leq r_2 \text{ then } \nu(r_1) \leq \nu(r_2) \tag{27}$$

the visual encodings must also have some sort of ordering. For interval scale data, ν is equivariant under translation monoid actions if

$$\nu(x + c) = \nu(x) + c \tag{28}$$

while for ratio data, there must be equivalent scaling[58]

$$\nu(xc) = \nu(x) * c \tag{29}$$

We therefore can test if a ν is equivariant by testing the actions under which is must commute. For example, we define a transform $\nu_i(x) = .5$ on interval data. This means it must commute under translation, for example $t(x) = x + 2$. Testing this constraint

$$\nu(t(r+2)) \stackrel{?}{=} \nu(r) + 2 \tag{30}$$

$$.5 \neq .5 + 2 \tag{31}$$

we find that the ν defined here does not commute and is therefore invalid. The constraints on ν can be embedded into our artist such that the ν functions can test for equivariance and also provide guidance on constructing new ν functions.

3.3.3 Graphic Assembler Q

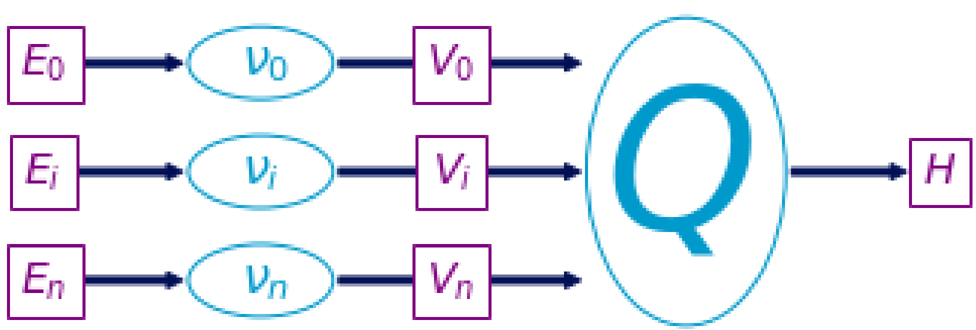


Figure 15: ν_i functions convert data τ_i to visual characteristics μ_i, then Q assembles μ_i into a graphic ρ such that there is a map ξ preserving the continuity of the data. ρ applied to a region of connected components S_j generates a part of a graphic, for example the point graphical mark.

As shown in figure 15, the assembly function Q combines the fiber F_i wise ν transforms into a graphic in H. Together, ν and Q are a map-reduce operation: map the data into their visual encodings, reduce the encodings into a graphic. As with ν the constraint on Q is that for every monoid action on the input μ there is corresponding monoid action on the output ρ.

While ρ generates the entire graphic, we will restrict the discussion of Q to generation of sections of a glyph. 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:

$$J = \{j \in K \text{ s. t. } \exists\gamma \text{ s.t. } \gamma(0) = k \text{ and } \gamma(1) = j\} \tag{32}$$

where the path[72] γ from k to j is a continuous function from the interval [0,1]. We define the glyph as the graphic generated by $Q(S_j)$

$$H \underset{\rho(S_j)}{\longleftrightarrow} S_j \underset{\xi^{-1}(J)}{\overset{\xi(s)}{\longleftrightarrow}} J_k \tag{33}$$

such that for every glyph there is at least one corresponding region on K. This is in keeping with the definition of glyph as any differentiable element put forth by Ziemkiewicz and Kosara[73]. The primitive point, line, and area marks[9, 74] are specially cased glyphs.

It is on sections of these glyphs that we define the equivariant map as $Q : \mu \mapsto \rho$ and an action on the subset of graphics $Q(\Gamma(V)) \in \Gamma(H)$ that Q can generate. We then define the constraint on Qsuch that if Q is applied to μ, μ' that generate the same ρ then the output of both sections acted on by the same monoid m must be the same. While it may seem intuitive that visualizations that generate the same glyph should consistently generate the same glyph given the same input, we formalize this constraint such that it can be specified as part of the implementation of Q.

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

$$Q(\mu) = Q(\mu') \implies Q(m \circ \mu) = Q(m \circ \mu') \tag{34}$$

then a monoid action on Y can be defined as $m \circ \rho = \rho'$. The transformed graphic ρ' is equivariant to a transform on the visual bundle $\rho' = Q(m \circ \mu)$ on a section that $\mu \in Q^{-1}(\rho)$ that must be part of generating ρ.

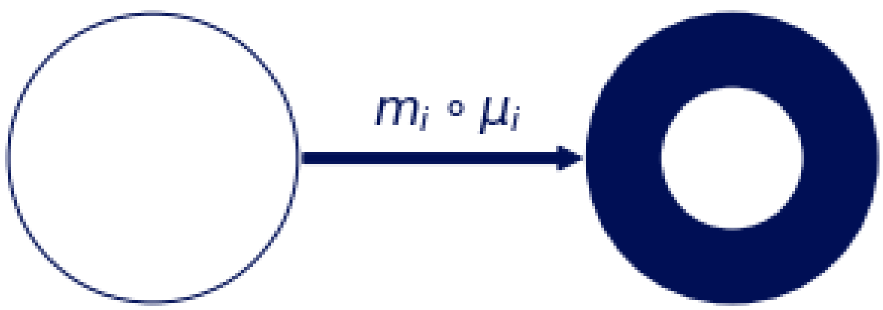


Figure 16: These two glyphs are generated by the same Q function. The monoid action m_i on edge thickness μ_i of the first glyph yields the thicker edge $\mu_i{}'$ in the second glyph.

The glyph in figure 16 has the following characteristics P specified by ($xpos$, $ypos$, $color$, $thickness$) such that one section is $\mu = (0, 0, 0, 1)$ and $Q(\mu) = \rho$ generates a piece of the thin hollow circle. The equivariance constraint on Q is that the action $m = (e, e, e, x + 2)$, where e is identity, translates μ to $\mu' = (e, e, e, 3)$. The corresponding action on ρ causes $Q(\mu')$ to be the thicker circle in figure 16.

3.3.4 Assembly Q

In this section we formulate the minimal Q that will generate distinguishable graphical marks: non-overlapping scatter points, a non-infinitely thin line, and an image.

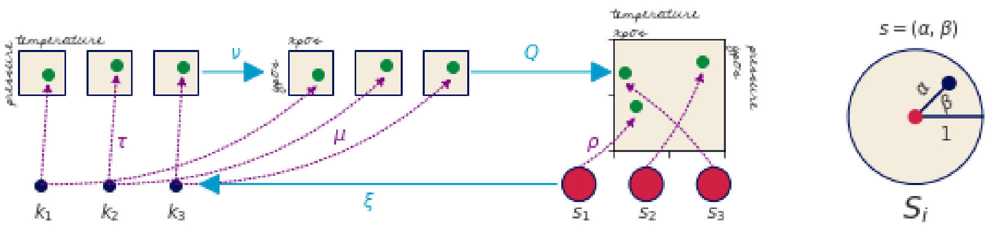


Figure 17: The data is discrete points (temperature, time). Via ν these are converted to (xpos, ypos) and pulled over discrete S. These values are then used to parameterize ρ which returns a color based on the parameters (xpos,ypos) and position α, β on S_k that ρ is evaluated on.

The scatter plot in figure 17 can be defined as $Q(xpos, ypos)(\alpha, \beta)$ where color $\rho_{RGB} = (0, 0, 0)$ is defined as part of Q and $s = (\alpha, \beta)$ defines the region on S. The position of this swatch of color can be computed relative to the location on the disc S_k as shown in figure 17:

$$x = size * \alpha \cos(\beta) + xpos \tag{35}$$

$$y = size * \alpha \sin(\beta) + ypos \tag{36}$$

such that $\rho(s) = (x, y, 0, 0, 0)$ colors the point (x,y) black. Here $size$ can either be defined inside Q or it could also be a parameter in V that is passed along with (xpos, vpos). As seen in figure 17, a scatter has a direct mapping from a region on S_k to its corresponding k.

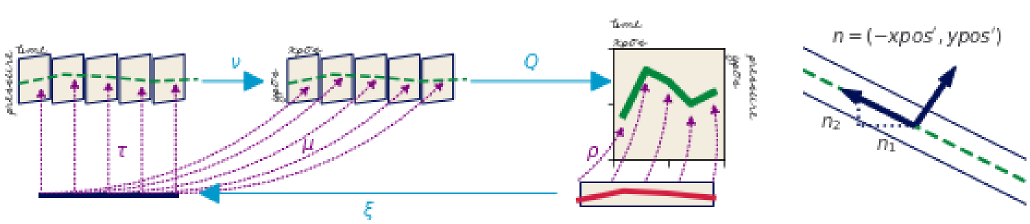


Figure 18: The line fiber ($time$, $temp$) is thickened with the derivative ($time'$, $temperature'$ because that information will be necessary to figure out the tangent to the point to draw a line. This is because the line needs to be pushed perpendicular to the tangent of (xpos, ypos). The data is converted to visual characteristics (xpos, ypos). The α coordinates on S specifies the position of the line, the β coordinate specifies thickness.

In contrast to the scatter, the line plot $Q(xpos, \hat{n_1}, ypos, \hat{n_2})(\alpha, \beta)$ shown in fig 18 has a ξ function that is not only parameterized on k but also on the α distance along k and corresponding region in $S\dot{T}$he line also exemplifies the need for the jet since the 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. The magnitude of the slope is

$$|n| = \sqrt{{n_1}^2 + {n_2}^2} \tag{37}$$

such that the normal is

$$\hat{n_1} = \frac{n_1}{|n|},\ \hat{n_2} = \frac{n_2}{|n|} \tag{38}$$

which yields components of ρ

$$x = xpos(\xi(\alpha)) + width * \beta\hat{n_1}(\xi(\alpha)) \tag{39}$$

$$y = ypos(\xi(\alpha)) + width * \beta\hat{n_2}(\xi(\alpha)) \tag{40}$$

where (x,y) look up the position $\xi(\alpha)$ on the data. At that point, we also look up the the derivatives $\hat{n_1}, \hat{n_2}$ which are then multiplied by a *width* parameter to specify the thickness. As with the *size* parameter in scatter, *width* can be defined in Q or as a component of V.

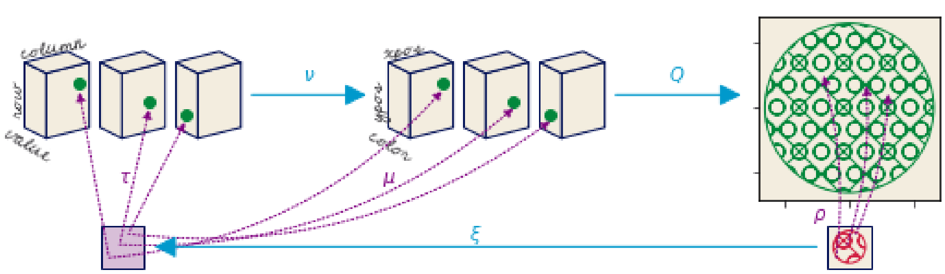


Figure 19: The only visual parameter an image requires is color since ξ encodes the mapping between position in data and position in graphic.

The image $Q(xpos, ypos, color)$ in figure 19 is a direct lookup into $\xi : S \to K$. Since K is 2D continuous space, the indexing variables (α, β) define the distance along the space.

This is then used by ξ to map into K to lookup the color values

$$R = R(\xi(\alpha, \beta)) \tag{41}$$

$$G = G(\xi(\alpha, \beta)) \tag{42}$$

$$B = B(\xi(\alpha, \beta)) \tag{43}$$

that the data values have been mapped into. 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 factory $\hat{Q}$

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 eq 22, Q is a bundle map $Q : \xi^*V \to H$ where ξ^*V and H are both bundles over S.

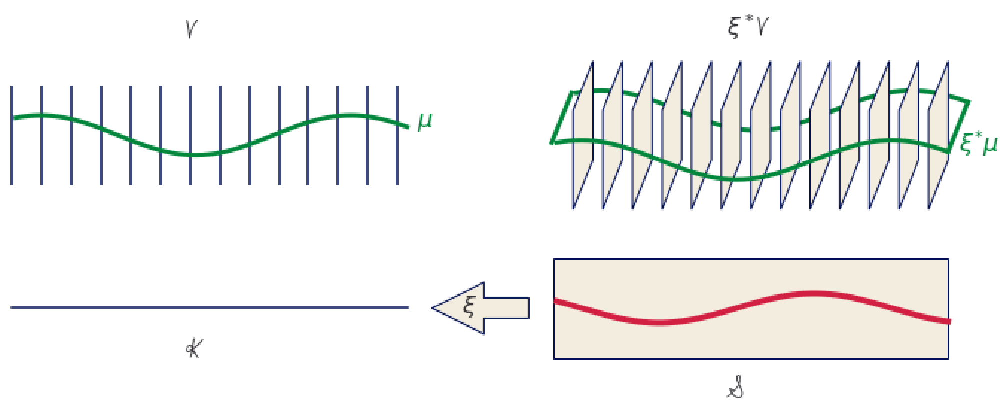


Figure 20: The pullback of the visual bundle ξ^*V is the replication of a μ over all points s that map back to a single k. Because the μ is the same, we can constract a $\hat{Q}$ on μ over k that will fabricate the Q for the equivalent region of s associated to that k

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

$$\xi^* V \mid_{\xi^{-1}(k)} = \xi^{-1}(k) \times P \tag{44}$$

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

This copying is illustrated in figure 20, where the 1D fiber $P \hookrightarrow V$ over K is copied repeatedly to become the 2D fiber $P^*\mu \hookrightarrow \xi^* V$ with identical components over S. Given the section $\xi^*\mu$ pulled back from μ and the point $s \in \xi^{-1}(k)$, there is a direct map from μon a point k, there is a direct map from the visual section over data base space $(k, \mu(k)) \mapsto (s, \xi^*\mu(s))$ to the visual section $\xi^*\mu$ over graphic base space. This map means that the pulled back section $\xi^*\mu(s) = \xi^*(\mu(k))$ is the section μ copied over all s. This means that $\xi^*\mu$ is identical for all s where $\xi(s) = k$, which is illustrated in figure 20 as each dot on $P$is equivalent to the line intersection $P^*\mu$.

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 the pulled back visual section $\xi^*\mu$ then

$$\rho(s) \coloneqq Q(\xi^*\mu)(s) \tag{45}$$

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

$$\hat{Q}(\mu(k))(s) \coloneqq Q((\xi^*\mu)(s)) \tag{46}$$

where the assembly function $\hat{Q}$ that takes as input the visual section on data μ is defined to be the assembly function 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.

Factoring out s from equation 46 yields $\hat{Q}(\mu(k)) = Q(\xi^*\mu)$ 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 majorly restructure the existing pipeline for libraries that delgate the construction of ρ to a back end such as Matplotlib.

3.3.6 Sheafs

The restriction maps of a sheaf describe how local τcan be glued into larger sections [75, 76]. As part of the definition of local triviality, there is an open neighborhood $U \subset K$ for every $k \in K$. We can define the inclusion map $\iota : U \to K$ which pulls Eover U

$$\begin{tikzcd}
\iota^* E \arrow[r, hook, "\iota^*"] \arrow[d, shift right, "\pi"'] & E \arrow[d, shift right, "\pi"'] \\
U \arrow[r, hook, "\iota"] \arrow[u, bend right, "\iota^*\tau"'] & K \arrow[u, bend right, "\tau"']
\end{tikzcd} \tag{47}$$

such that the pulled back $\iota^*\tau$ only contains records over $U \subset K$. By gluing $\iota^*\tau$ together, the sheaf is putting a continuous structure on local sections which allows for defining a section over a subset in K. That section over subset K maps to the graphic generated by A for visualizations such as sliding windows[77, 78] streaming data, or navigation techniques such as pan and zoom[79].

3.3.7 Composition of Artists: $+$

To build graphics that are the composites of multiple artists, we define a simple addition operator that is the disjoint union of fiber bundles E. For example, a scatter plot E_1 and a line plot E_2 have different Kthat are mapped to separate S. To fully display both graphics, the composite graphic $A_1 + A_2$ needs to include all records on both K_1 and K_2, which are the sections on the disjoint union $K_1 \sqcup K_2$. This in turn yields disjoint graphics $S_1 \sqcup S_2$ rendered to the same image. Constraints can be placed on the disjoint union such as that the fiber components need to have the same νposition encodings or that the position μneed to be in a specified range. There is a second type of composition where E_1 and E_2 share a base space $K_2 \hookrightarrow K_1$ such that the the artists can be considered to be acting on different components of the same section. This type of composition is important for creating visualizations where elements need to update together in a consistent way, such as multiple views [80, 81] and brush-linked views[82, 83].

3.3.8 Equivalance class of artists A'

It is impractical to implement an artist for every single graphic; instead we implement an approximation of an the equivalence class of artists $\{A \in A' : A_1 \equiv A_2\}$. Roughly, equivalent artists have the same fiber bundle V and same assembly function Q but act on different sections μ, but we will formalize the definition of the equivalance class in future work. As a first pass for implementation purposes, we identify a minimal P associated with each A' that defines what visual characteristics of the graphic must originate in the data such that the graphic is identifiable as a given chart type.

For example, a scatter plot of red circles is the output of one artist, a scatter plot of green squares the output of another. These two artists are equivalent since their only difference is in the literal visual encodings (color, shape). Shape and color could also be defined in Q but the position must come from the fiber $P = (xpos, ypos)$ since fundementally a scatter plot is the plotting of one position against another[7]. We also use this criteria to identify derivative types, for example the bubble chart[15] is a type of scatter where by definition the glyph size is mapped from the data. The criteria for equivalence class membership serves as the basis for evaluating invariance[**kindlmann2014algebraic**].

4 Prototype Implementation: Matplottoy

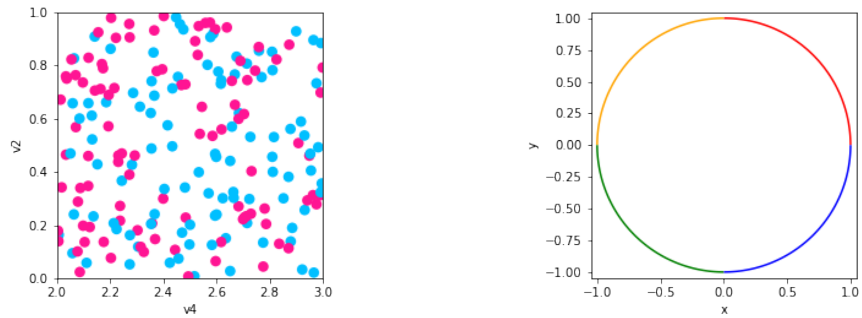


Figure 21: Scatter plot and line plot implemented using prototype artists and data models, building on Matplotlib rendering.

To prototype our model, we implemented the artist classes for the scatter and line plots shown in figure 21 because they differ in every attribute: different visual channels ν that composite to different marks Q with different continuities $\xi\dot{\mathrm{W}}$e make use of the Matplotlib figure and axes artists [4, 5] so that we can initially focus on the data to graphic transformations. We also exploit the Matplotlib transform stack to transform data coordinates into screen coordinates. To generate the images in figure 21, we instantiate `fig, ax` artists that will contain the new `Point, Line` primitive objects we implemented based on our topology model.

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

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

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

4.1 Artist Class A'

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

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

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

```

```python
    def draw(self, renderer):
        # returns K, indexed on fiber then key
        # is passed the
        view = self.data.view(self.axes)
        # visual channel encoding applied fiberwise
        visual = {p: t['encoder'](view[t['name']])
                   for p, t in self.transforms.items()}
        self.assemble(**visual)
        # 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 we can use to query back into data to select a subset as discussed in section 3.3.6. The ν functions are then applied to the data to generate the visual section μ that here is the object `visual`. The conversion from data to visual space is simplified here to directly show that it is the encoding ν applied to the component. In the full implementation, we allow for fixed visual parameter, such as setting a constant color for all sections, by verifying that the named component is in F before accessing the data. If the data component name is not in F this is interpreted to mean this component is a thickening of V that could be pulled back to E via an inverse identity ν.

The components of the visual object, denoted by the Python unpacking convention `**visual` are then passed into the `assemble` function that is $\hat{Q}$. This assembly function is responsible for generating a representation such that it could be serialized to recreate a static version of the graphic. Although `assemble` could be implemented outside the class such that it returns an object the artist could then parse to set attributes, the attributes are directly set here to reduce indirection. This artist is not optimized because we prioritized demonstrating the separability of ν and $\hat{Q}$. The last step in the artist function is handing itself off to the renderer. The extra *arg, **kwargs arguments in `__init__`,`draw` are artifacts of how these objects are currently implemnted in Matplotlib.

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

```python
class Point(mcollections.Collection):
    def assemble(self, x, y, s, facecolors='C0' ):
        # construct geometries of the circle glyphs in visual coordinates
        self._paths = [mpath.Path.circle(center=(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)
```

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. To ensure the integrity of the section, `view` must be atomic. This means that the values cannot change after the method is called in draw until a new call in draw. We put this constraint on the return of the `view` method so that we do not risk race conditions.

This table is converted to a table of visual variables and is then passed into `assemble`. In `assemble`, 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. As mentioned in sections **??** and 3.3.3, this assembly function could as easily be implemented such that it was fed one $\tau(k)$ at a time.

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

```
class Line(mcollections.LineCollection):
    def assemble(self, x, y, color='C0'):
            #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)
```

In the `Line` artist, `view` returns a table of edges. Each edge consists of (x,y) points sampled along the line defined by the edge and information such as the color of the edge. As with `Point`, the data is then converted into visual variables. In `assemble`, this visual representation is composed into a set of line segments, where each segement 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 view.

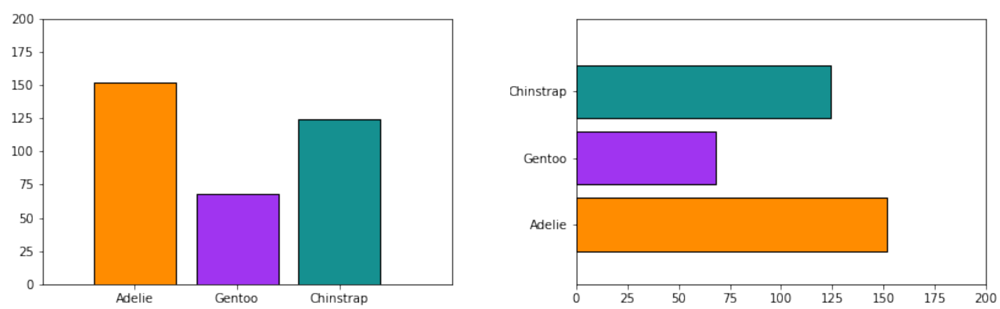


Figure 22: Frequency of Penguin types visualized as discrete bars.

The bar charts in figure 22 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 $\hat{Q}$.

```python
class Bar(mcollections.Collection):
    def __init__(self, data, transforms, orientation='v', *args, **kwargs):
        """
        orientation: str, optional
            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)
        self.data = data
        self.transforms = copy.deepcopy(transforms)

    def assemble(self, position, length, floor=0, width=0.8,
                     facecolors='C0', edgecolors='k', offset=0):
        #set some defaults
        width = itertools.repeat(width) if np.isscalar(width) else width
        floor = itertools.repeat(floor) if np.isscalar(floor) else (floor)

        # offset is passed through via assemblers such as multigroup,
        # not supposed to be directly tagged to position
        position = position + offset

```

```
        def make_bars(xval, xoff, yval, yoff):
             return [[(x, y), (x, y+yo), (x+xo, y+yo), (x+xo, y), (x, y)]
                for (x, xo, y, yo) in zip(xval, xoff, yval, yoff)]
        #build bar glyphs based on graphic parameter
        if self.orientation in {'vertical', 'v'}:
            verts = make_bars(position, width, floor, length)
        elif self.orientation in {'horizontal', 'h'}:
            verts = make_bars(floor, length, position, width)

        self._paths = [mpath.Path(xy, closed=True) for xy in verts]
        self.set_edgecolors(edgecolors)
        self.set_facecolors(facecolors)

    def draw(self, renderer,  *args, **kwargs):
        view = self.data.view(self.axes)
        visual = {}
        for (p, t) in self.transforms.items():
            if isinstance(t, dict):
                if t['name'] in self.data.FB.F and 'encoder' in t:
                    visual[p] = t['encoder'](view[t['name']])
                elif 'encoder' in t: # constant value
                    visual[p] = t['encoder'](t['name'])
                elif t['name'] in self.data.FB.F: # identity
                    visual[p] = view[t['name']]
            else: # no transform
                 visual[p] = t
        self.assemble(**visual)
        super().draw(renderer,  *args, **kwargs)
```

The `draw` method here has a more complex unpacking of visual encodings to support passing in visual component data directly. This is vastly simplifies building composite objects as the alternative would be higher order functions that take as input the transforms passed in by the user. This construction supports a constant visual parameter, an identity transform where the value is the same in E and V, and setting the visual component directly. The `assemble` function constructs bars and sets their face and edge colors. The `make_bars` function converts the input position and length to the coordinates of a rectangle of the given width. Defaults are provided for 'width' and 'floor' to make this function more reusable. Typically the defaults are used for the type of chart shown in figure 22, but these visual variables are often set when building composite versions of this chart type as discussed in section 4.4.

4.2 Encoders ν

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

```
cmap =  color.Categorical({'true':'deeppink', 'false':'deepskyblue'})
transforms = {'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 the position *(x,y)* ν transformers are identity functions. The size s transformer is not acting on a component of F, instead it is a ν that returns a constant value. While size could be embedded inside the `assemble` function, it is added to the transformers to illustrate user configured visual parameters that could either be constant or mapped to a component in F.

The identity and constant ν are explicitly implemented here to demonstrate their implicit role in the visual pipeline, but they are somewhat optimized away in `Bar`. More complex encoders can be implemented as callable classes, such as

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

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

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

```python
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)
```

but is currently factored out of the artist for clarity. In this example, `is_nominal` checks for equivariance of permutation group actions by applying the encoder to a set of values, shuffling values, and checking that (value, encoding) pairs remain the same.

4.3 Data E

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. The object returned by the view must be key valued pairs of `{component name : component section}` where each section is a component as defined in equation 15. To support specifying the fiber bundle, we define a `FiberBundle` data class[84]

```python
@dataclass
class FiberBundle:
"""
Attributes
----------
K: {'tables': []}
F: {variable name: type}
"""
    K: dict
    F: dict
```

that asks the user to specify how $K$is triangulated and the attributes of F. Python dataclasses are a good abstraction for the fiber bundle class because the `FiberBundle` class only stores data. The K is specified as tables because the `assemble` functions expect tables that match the continuity of the graphic; scatter expects a vertex table because it is discontinuous, line expects an edge table because it is 1D continuous. The fiber informs appropriate choice of ν therefore it is a dictionary of attributes of the fiber components.

To generate the scatter plot in figure 21, we fully specify a dataset with random keys and values in a section chosen at random form the corresponding fiber component. The fiberbundle `FB` is a class level attribute since all instances of `VertexSimplex` come from the same fiberbundle.

```python
class VertexSimplex: #maybe change name to something else
    """Fiberbundle is consistent across all sections
    """
    FB = FiberBundle({'tables': ['vertex']},
            {'v1': float, 'v2': str, 'v3': float})

    def __init__(self, sid = 45, size=1000, max_key=10**10):
        # create random list of keys
    def tau(self, k):
        # e1 is sampled from F1, e2 from F2, etc...
        return (k, (e1, e2, e3, e4))

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

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

```
class EdgeSimplex:

        FB = FiberBundle({'tables': ['vertex','edge']},
                         {'x' : float, 'y':  float,
                         'color':mtypes.Color()}})
    def __init__(self, num_edges=4, num_samples=1000):
        self.keys = range(num_edge) #edge id
        # distance along edge
        self.distances = np.linspace(0,1, num_samples)
        # half generlized representation of arcs on a circle
        self.angle_samples = np.linspace(0, 2*np.pi, len(self.keys)+1)

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

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

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

```python

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

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

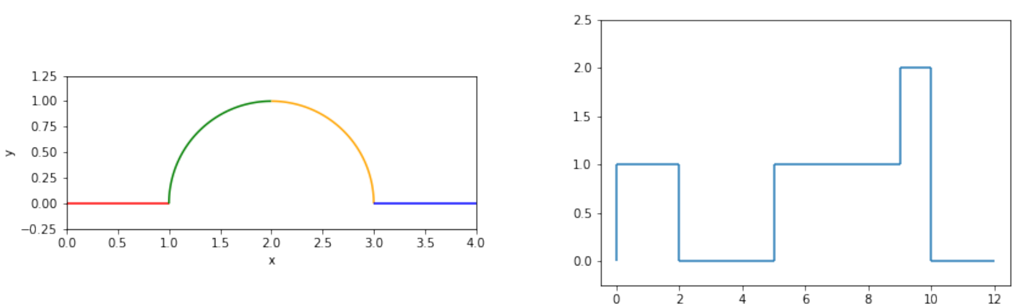


Figure 23: Continuous and discontinuous lines as defined via the same data model, and generated with the same A'`Line`

The graphics in figure 23 are made using the `Line` artist and the `Graphline` data source

```python
class GraphLine:
    def __init__(self, FB, edge_table, vertex_table, num_samples=1000, connect=False):
```

```
        #s set args as attributes and generate distance
        if connect: # test connectivity if edges are continuous
            assert edge_table.keys() == self.FB.F.keys()
            assert is_continuous(vertex_table)

    def tau(self, k):
        # evaluates functions defined in edge table
        return(k, (self.edges[c][k](self.distances) for c in self.FB.F.keys()))

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

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

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

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

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

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

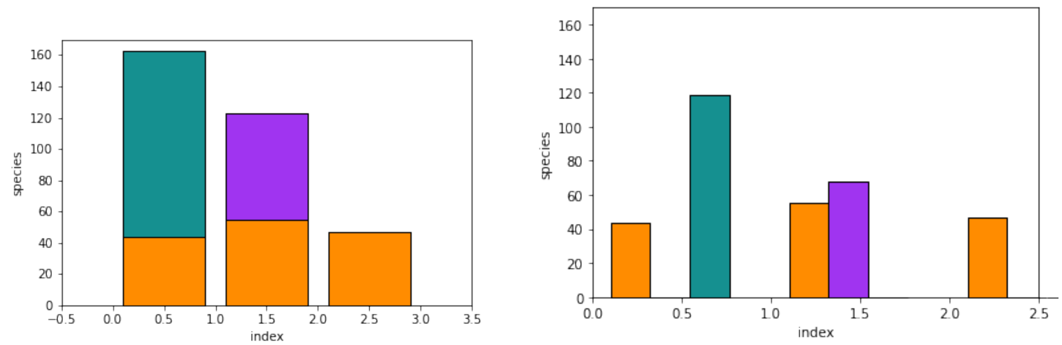
4.4 Case Study: Penguins

For this case study, we use the Palmer Penguins dataset[85, 86] since it is multivariate and has a varying number of penguins. We use a version of the data packaged as a pandas dataframe[87] since that is a very commonly used Python labeled data structure. The wrapper is very thin because there is explicitly only one section.

```python
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 24: Penguin count disaggregated by island and species

The stacked and grouped bar charts in figure 24 are both out 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 applied to only one component and the case `mtransforms` where the same transform is applied to multiple components such that V has multiple components that map to the same retinal variable.

```
class StackedBar(martist.Artist):
    def __init__(self, data, transforms, mtransforms, 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)
```

```python
        self.data = data
        self.orientation = orientation
        self.transforms = copy.deepcopy(transforms)
        self.mtransforms = copy.deepcopy(mtransforms)

    def assemble(self):
        view = self.data.view(self.axes)
        self.children = [] # list of bars to be rendered
        floor = 0
        for group in self.mtransforms:
            # pull out the specific group transforms
            group['floor'] = floor
            group.update(self.transforms)
            bar = Bar(self.data, group, 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 assemble(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)
            group['width'] = width/ngroups
            group['offset'] = gid/ngroups*width
            bar = Bar(self.data, group, 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':'island',
              'encoder': position.Nominal({'Biscoe':0.1, 'Dream':1.1, 'Torgersen':2.1})}}
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

This work contributes a mathematical description of the mapping A from data to visual representation. Combining Butler's proposal of a fiber bundle model of visualization data with Spivak's formalism of schema lets this model support a variety of datasets, including discrete relational tables,, multivariate high resolution spatio temporal datasets, and complex networks. Decomposing the artist into encoding ν, assembly Q, and reindexing ξ provides the specifications for producing visualization where the structure and properties match those of the input data. These specifications are that the graphic must have continuity equivalent to the data, and that the visual characteristics of the graphics are equivariant to their corresponding components under monoid actions. This model defines these constraints on the transformation function such that they are not specific to any one type of encoding or visual characteristic. Encoding the graphic space as a fiber bundle provides a structure rich abstraction of the target graphical design in the target display space.

The toy prototype built using this model validates that is usable for a general purpose visualization tool since it can be iteratively integrated into the existing architecture rather than starting from scratch. Factoring out glyph formation into assembly functions allows for much more clarity in how the glyphs differ. This prototype demonstrates that this framework can generate the fundemental marks: point (scatter plot), line (line chart), and area (bar chart). Furthermore, the grouped and stacked bar examples demonstrate that this model supports composition of glyphs into more complex graphics. These composite examples also rely on the fiber bundles section base book keeping to keep track of which components contribute to the attributes of the glyph. Implementing this example using a Pandas dataframe demonstrates the ease of incorporating existing widely used data containers rather than requiring users to conform to one standard.

5.1 Limitations

So far this model has only been worked out for a single data set tied to a primitive mark, but it should be extensible to compositing datasets and complex glyphs. The examples and prototype have so far only been implemented for the static 2D case, but nothing in the math limits to 2D and expansion to the animated case should be possible because the model is formalized in terms of the sheaf. While this model supports equivariance of figurative glyphs generated from parameters of the data[88, 89], it currently does not have a way to evaluate the semantic accuracy of the figurative representation. Effectiveness is out of scope for this model because it is not part of the structure being preserved, but potentially a developer building a domain specific library with this model could implement effectiveness criteria in the artists. Also, even though the model is designed to be backend and format independent, it has only really been tested against PNGs rendered with the AGG backend. It is especially unknown how this framework interfaces with high performance rendering libraries such as openGL[68]. Because this model has been limited to the graphic design space, it does not address the critical task of laying out the graphics in the image

This model and the associated prototype is deeply tied to Matplotlib's existing architecture. While the model is expected to generalize to other libraries, such as those built on Mackinlay's APT framework, this has not been worked through. In particular, Mackinlay's formulation of graphics as a language with semantic and syntax lends itself a declarative interface[90], which Heer and Bostock use to develop a domain specific visualization language that they argue makes it simpler for designers to construct graphics without sacrificing expressivity [18]. Similarly, the model presented in this work formulates visualization as equivariant maps from data space to visual space, and is designed such that developers can build software libraries with data and graphic topologies tuned to specific domains.

5.2 Future Work

While the model and prototype demonstrate that generation of simple marks from the data, there is a lot of work left to develop a model that underpins a minimally viable library. Foremost is implementing a data object that encodes data with a 2D continous topology and an artist that can consume data with a 2D topology to visualize the image[91–93] and also encoding a separate heatmap[94, 95] artist that consumes 1D discrete data. A second important proof of concept artist is a boxplot[96] because it is a graphic that assumes computation on the data side and the glyph is built from semantically defined components and a list of outliers. The model supports simple composition of glyphs by overlaying glyphs at the same position, but more work is needed to define an operator where the fiber bundles have shared $S_2 \hookrightarrow S_1$ such that fibers could be pulled back over the subset. While the model's simple addition supports axes as standalone artists with overlapping visual position encoding, the complex operator would allow for binding together data that needs to be updated together. Additionally, implementing the complex addition operator and explicit graphic to data maps would allow for developing a mathematical formalism and prototype of how interactivity would work in this model. In summary, the proposed scope of work for the dissertation is

- expansion of the mathematical framework to include complex addition
- formalization of definition of equivalance class A'
- implementation of artist with explicit ξ
- specification of interactive visualization
- mathematical formulation of a graphic with axes labeling
- implementation of new prototype artists that do not inherit from Matplotlib artists
- provisional mathematics and implementation of user level composite artists
- proof of concept domain specific user facing library

Other potential tasks for future work is implementing a data object for a non-trivial fiber bundle and exploiting the models section level formalism to build distributed data source models and concurrent artists. This could be pushed further to integrate with topological[97] and functional [98] data analysis methods. Since this model formalizes notions of structure preservation, it can serve as a good base for tools that assess quality metrics[99] or invariance [12] of visualizations with respect to graphical encoding choices. While this paper formulates visualization in terms of monoidal action homomorphisms between fiberbundles, the model lends itself to a categorical formulation[100, 101] that could be further explored.

6 Conclusion

An unoffical philosophy of Matplotlib is to support making whatever kinds of plots a user may want, even if they seem nonsensical to the development team. The topological framework described in this work provides a way to facilitate this graph creation in a rigorous manner; any artist that meets the equivariance criteria described in this work by definition generates a graphic representation that matches the structure of the data being represented.

We leave it to domain specialists to define the structure they need to preserve and the maps they want to make, and hopefully make the process easier by untangling these components into seperate constrained maps and providing a fairly general data and display model.

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[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] 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.

[11] Jock Mackinlay. "Automatic Design of Graphical Presentations". English. PhD Thesis. Stanford, 1987.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

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

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] Stephen Few and Perceptual Edge. “Dashboard Confusion Revisited”. In: *Perceptual Edge* (2007), pp. 1–6.

[42] 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.

[43] 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.

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

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

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

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

[48] A. M. Cegarra. “Cohomology of Monoids with Operators”. In: *Semigroup Forum*. Vol. 99. Springer, 2019, pp. 67–105. ISBN: 1432-2137.

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

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

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

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

[53] Michele Stieven. *A Monad Is Just a Monoid...* en. https://medium.com/@michelestieven/a-monad-is-just-a-monoid-a02bd2524f66. Apr. 2020.

[54] “Monoid”. en. In: *Wikipedia* (Jan. 2021).

[55] “Semigroup Action”. en. In: *Wikipedia* (Jan. 2021).

[56] nLab authors. “Action”. In: (Mar. 2021).

[57] W A Lea. “A Formalization of Measurement Scale Forms”. en. In: (), p. 44.

[58] Eric W. Weisstein. *Similarity Transformation.* en. https://mathworld.wolfram.com/SimilarityTransformation.ht Text.

[59] “Quotient Space (Topology)”. en. In: *Wikipedia* (Nov. 2020).

[60] Professor Denis Auroux. “Math 131: Introduction to Topology”. en. In: (), p. 113.

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

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

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

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

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

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

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

[68] 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.

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

[70] “Jet Bundle”. en. In: *Wikipedia* (Dec. 2020).

[71] 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.

[72] “Connected Space”. en. In: *Wikipedia* (Dec. 2020).

[73] 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.

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

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

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

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

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

[85] 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.

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

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

[88] 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.

[89] 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.

[90] Kenneth C. Louden. *Programming Languages : Principles and Practice*. English. Pacific Grove, Calif: Brooks/Cole, 2010. ISBN: 978-0-534-95341-6 0-534-95341-7.

[91] 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.

[92] 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.

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

[94] 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.

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

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

[97] 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.

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

[99] 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.

[100] 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.

[101] Bartosz Milewski. “Category Theory for Programmers”. en. In: (), p. 498.