

1

TOPOLOGICAL ARTIST MODEL

2

HANNAH AIZENMAN

3

A DISSERTATION PROPOSAL SUBMITTED TO

4

THE GRADUATE FACULTY IN COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE

5

REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY,

6

THE CITY UNIVERSITY OF NEW YORK

7

COMMITTEE MEMBERS:

8

DR. MICHAEL GROSSBERG (ADVISOR), DR. ROBERT HARALICK, DR. LEV MANOVICH,

9

DR. HUY VO, DR. MARCUS HANWELL

10

JUNE 2021

11

Abstract

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

35 Contents

| | |
|--|----|
| 36 Abstract | ii |
| 37 1 Introduction | 1 |
| 38 2 Background | 3 |
| 39 2.1 Structure | 3 |
| 40 2.2 Tools | 6 |
| 41 2.3 Data | 8 |
| 42 2.4 Contribution | 9 |
| 43 3 Topological Equivariant Artist Model | 10 |
| 44 3.1 Data Space E | 11 |
| 45 3.1.1 Variables in Fiber Space F | 11 |
| 46 3.1.2 Measurement Scales: Monoid Actions | 13 |
| 47 3.1.3 Continuity of the Data | 16 |
| 48 3.1.4 Data Values | 19 |
| 49 3.1.5 Sheafs | 20 |
| 50 3.1.6 Applications | 21 |
| 51 3.2 Graphic Space H | 21 |
| 52 3.2.1 Idealized Display D | 22 |
| 53 3.2.2 Continuity of the Graphic S | 22 |
| 54 3.2.3 Graphic | 24 |
| 55 3.3 Artist | 25 |
| 56 3.3.1 Visual Fiber Bundle V | 27 |
| 57 3.3.2 Visual Encoders | 28 |
| 58 3.3.3 Visualization Assembly | 31 |
| 59 3.3.4 Assembly Q | 33 |
| 60 3.3.5 Assembly Template \hat{Q} | 37 |
| 61 3.3.6 Composition of Artists: $+$ | 39 |

| | | |
|----|--|-----------|
| 62 | 3.3.7 Equivalence class of artists | 41 |
| 63 | 4 Prototype: Matplottoy | 42 |
| 64 | 4.1 Scatter, Line, and Bar Artists | 44 |
| 65 | 4.2 Visual Encoders | 48 |
| 66 | 4.3 Data Model | 49 |
| 67 | 4.4 Case Study: Penguins | 51 |
| 68 | 5 Discussion | 56 |
| 69 | 5.1 Limitations | 57 |
| 70 | 5.2 Future Work | 58 |
| 71 | 6 Conclusion | 60 |

72 1 Introduction

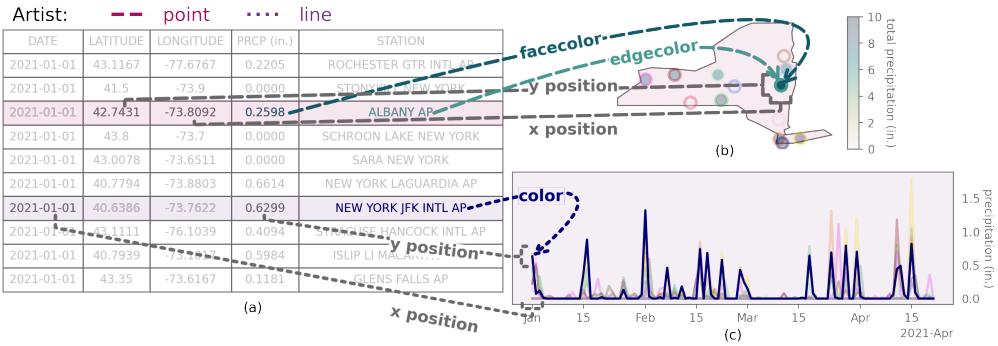


Figure 1: Building block visualization libraries implement independent reusable functions that map data to visual representations. For example, position encoding functions map the latitude and longitude values in the table (a) to locations on the map (b). These same encoding functions map dates and precipitation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

73 Visualizations are representations of data and therefore reflect something of the under-
 74 lying structure and semantics [1], whether through direct mappings from data into visual
 75 elements or via figurative representations that have meaning due to their similarity in shape
 76 to external concepts [2]. We define visualization components to be structure preserving
 77 maps from data to visual representations. As illustrated in Figure 1, visualizations map
 78 weather station precipitation data (a) into graphical elements such as points (b) and lines
 79 (c) and map individual data components (columns) into components of the visualization
 80 such as position or color. For example, the gray position encoder function converts the
 81 latitude and longitude to x and y positions, and the color encoders map temperature and
 82 station name to colors. A **point** function composites these encodings into attributes of a
 83 point in the map (a). The **line** function composites encoders that convert station name to
 84 color and date and precipitation to x and y positions into a piece of a line in the time series

85 plot (c). We identify the structure these maps must preserve as the *continuity* of the data
86 and the *equivariance* of the data and visual components.

87 We introduce a model of visualization components based on these *continuity* and *equivariance*
88 constraints and use this model to develop a design specification. Our specification
89 is targeted at building block libraries, which Wongsuphasawat defines as visualization tools
90 that provide independent functions that map components of data to visual representations
91 [3]. Just as the number of ways to arrange physical building blocks is solely constrained
92 by the size and shape of the blocks, we propose that the only restriction on how building
93 block library components can be composed are that the compositions preserve *continuity*
94 and *equivariance*. Independent, modular, components are inherently functional [4], so we
95 propose a functional architecture for our model. Doing so means the functional architecture
96 can be evaluated for correctness, the resulting code is likely to be shorter and clearer, and
97 the architecture is well suited to distributed, concurrent, and on demand tasks[5].

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

113 2 Background

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

121 2.1 Structure

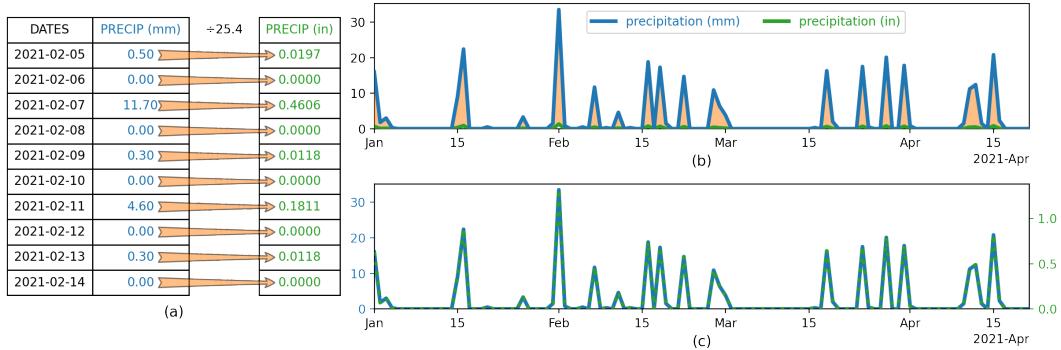


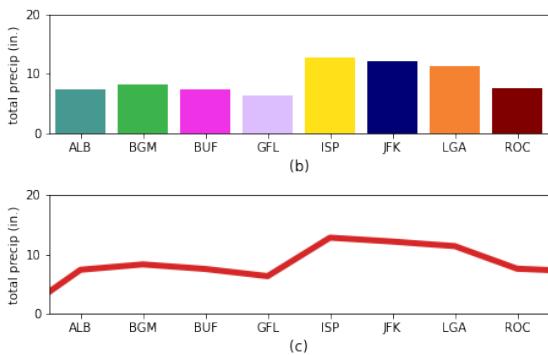
Figure 2: The table (a) lists precipitation in millimeters in blue. These values are converted to inches, in green, by dividing the millimeter values by 25.4. This conversion is a scaling action, represented by the orange arrows in (a). The *equivariant* action is the scaling in the time series (b),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

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

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

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

(a)



(b)

(c)

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

146 The notion that data is equivalent to visual representations when structure is preserved
147 serves as the basis for visualization best practices. When *continuity* is preserved, as in
148 the bar chart (b) in [Figure 3](#), then the graphic has not introduced new structure into the
149 data. In [Figure 3](#), the line plot (c) does not preserve *continuity* because the line connecting
150 the total precipitation of the stations implies that the stations are connected to each other
151 along a 1D continuous line. Bertin asserted that *continuity* was preserved by choosing
152 graphical marks that match the *continuity* of the data - for example discrete data is a
153 point, 1D continuous is the line, and 2D data is the area mark. Informally, Norman's
154 Naturalness Principal [15, 16] states that a visualization is easier to understand when the
155 properties of the visualization match the properties of the data. This principal is made more
156 concrete in Tufte's concept of graphical integrity, which is that a visual representation of
157 quantitative data must be directly proportional to the numerical quantities it represents (Lie
158 Principal), must have the same number of visual dimensions as the data, and should be well
159 labeled and contextualized, and not have any extraneous visual elements [17]. Expressivity,
160 as defined by Mackinlay, is a measure how much of the mathematical structure in the
161 data can be expressed in the visualizations; for example that ordered variables can be
162 mapped into ordered visual elements. We generalize these different codifications of structure
163 preserving encodings, proposing that a graphic is an equivalent representation of the data
164 when *continuity* and *equivariance* are preserved.

Structure

continuity How elements in the dataset are connected to each other, e.g. discrete

points, networked nodes, points on a continuous surface

equivariance if an action is applied to the data or the graphic—e.g. a rotation,

permutation, translation, or rescaling—there must be an equivalent action applied
on the other side of the transformation.

₁₆₅ **2.2 Tools**

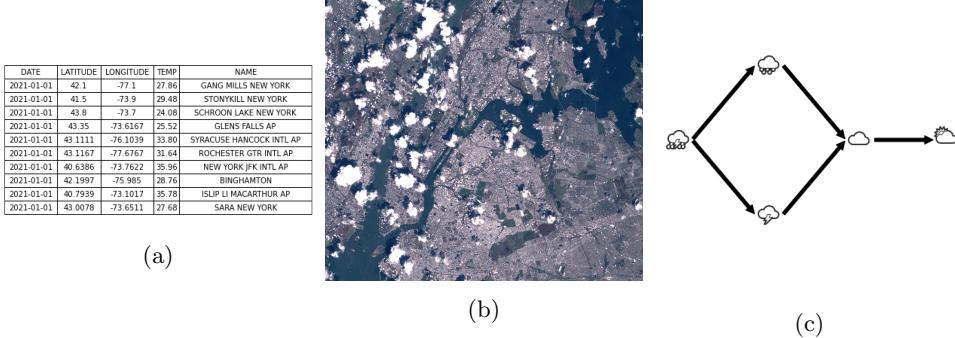


Figure 4: Visualization libraries, especially ones tied to specific domains, often are designed around a core data structure, such as tables Figure 4a, images Figure 4b, or networks Figure 4c.

₁₆₆ Most information visualization software design patterns are tuned to specific data structures
₁₆₇ and domains, as categorized by Heer and Agrawala [18]. For users who generally work in
₁₆₈ one domain—such relational tables (Figure 4a), images (Figure 4b), or graphs (Figure 4c)—
₁₆₉ well defined data space (and corresponding visual space [19]) often yields a coherent user
₁₇₀ experience [20]. This coherent experience is often not extensible; developers who want to
₁₇₁ build new visualizations on top of these libraries must work around the existing assumptions,
₁₇₂ sometimes in ways that break the architecture model of the libraries. For example tools
₁₇₃ influenced by APT that assume that data is a relational table integrate computation into
₁₇₄ the visualization pipeline. This is a wide array of tools, including Tableau [21–23] and
₁₇₅ the Grammar of Graphics [24] inspired ggplot [25], protovis [26], vega [27] and altair [28].
₁₇₆ Since these libraries represent data as a table, and computations on tables are well defined
₁₇₇ [29], these libraries implement computation as part of the visualization process. This is
₁₇₈ also true of tools that support images, such as the biology oriented ImageJ [30] and Napari
₁₇₉ [31] or the digital humanities ImageJ macro ImagePlot [32]. While these tools often have
₁₈₀ some support for visualizing non image components of the data, the architecture is oriented
₁₈₁ towards building plugins into the existing system [33] where the image is the core data
₁₈₂ structure. Tools like Gephi [34], Graphviz [35], and Networkx [36] are used to visualize
₁₈₃ and manipulate graphs. As with tables and images, extending network libraries to work

184 with other types of data either requires breaking the libraries internal model of how data is
 185 structured and what data transformations are allowed or implementing a model for other
 186 types of data structures alongside the network model. Our model aims to identify which
 187 computations are manipulations of the data and which are necessary for the visual encoding;
 188 for example data is often aggregated for a bar chart but it is not required, while a boxplot
 189 by definition requires computing distribution statistics [37]. Disentangling the computation
 190 from the visual transforms allows us to determine whether the visualization library should
 191 implement computations or if they should be implemented in data space.

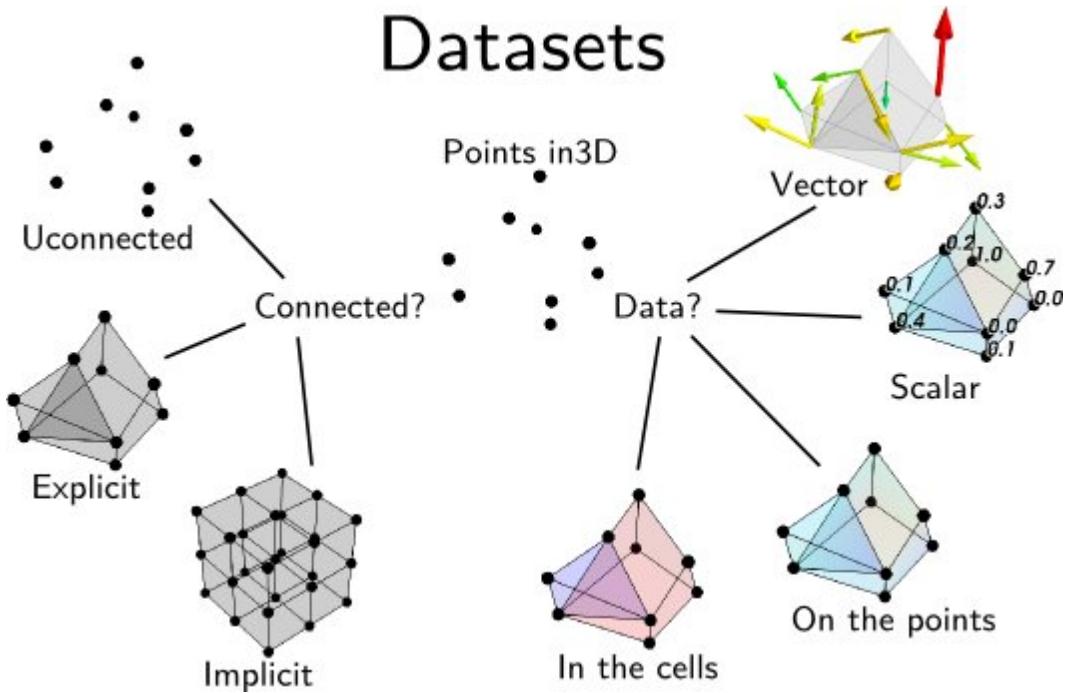


Figure 5: The *continuity* of the data is often used to classify data structures. For example, a relational database often consists of discrete unconnected points, while an image is an implicitly connected 2D grid. This image is from the Data Representation chapter of the MayaVi 4.7.2 documentation. [38]

192 Visualizations assume the structure of the input data, as described in Tory and Möller's
 193 taxonomy [39], which leads to many building block libraries implementing multiple models
 194 of data to support these different visualizations. For example, in Matplotlib, D3 [40] VTK
 195 [41, 42] and MayaVi [43], every plot is defined in terms of the *continuity* of the data it

expects as input. VTK has explicitly codified this in terms of *continuity* based data representations, as illustrated in figure 5. Downstream library developers impose some coherency by writing domain specific libraries with assumed data structures on top of the building block libraries—for example Seaborn [44] and Titan [45] assume a relational database, xarray [46] and ParaView [47] assume a data cube—but must work around the incoherencies in the building block libraries to do so. Our model navigates the tradeoff between coherency and extensibility by proposing functional composable well constrained visual components that take as input a structure aware data abstraction general enough to provide a common interface for many different types of data continuities.

2.3 Data

Fiber bundles were proposed by Butler as a core data structure for visualization because they encode data *continuity* separately from the components of the dataset [48, 49]. Butler’s model lacks a robust way of describing variables; therefore we encode a schema like description of the data in the fiber bundle using Spivak’s topological description of data types [50, 51]. In this work, we refer to the points of the dataset as *records*, as defined by Spivak. Each component of the record is a single object, such as a precipitation measurement, a station name, or an image. We generalize *component* to mean all objects in the dataset of a given type, such as all precipitation values or station names or images. The way in which these records are connected is the *continuity* or more generally topology.

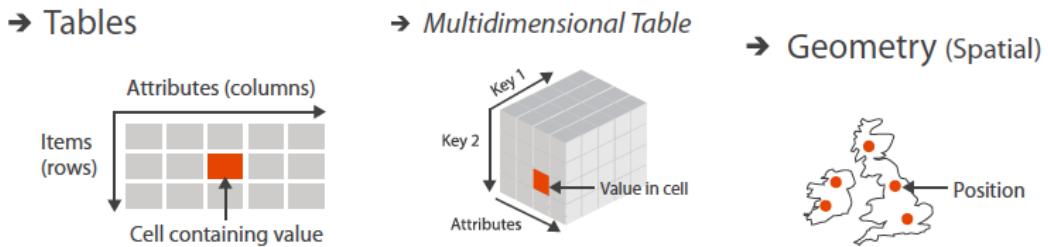


Figure 6: Values in a dataset have keys associated with them that describe where the value is in the dataset. These keys can be indexers or semantically meaningful; for example, in a table the keys are the variable name and the row ID. In the data cube, the keys is the row, column, and cell ID, and in the map the key is the position in the grid. Image is figure 2.8 in Munzner’s Visualization Analysis and Design [52]

215 The *continuity* can be described in some datasets by components of the dataset. This
216 is formalized by Munzner’s notion of metadata as *keys* into the data structure that return
217 associated *values* [53]. As shown in [Figure 6](#), keys can be labeled indexes, such as the
218 attribute name and row ID, or semantically significant physical entities such as locations on
219 a map. In contrast to Munzner’s model, in our model components may describe the keys
220 but are never themselves the keys; instead we propose that keys are points on a topological
221 space encoding the continuity of the data. This allows the metadata to be altered without
222 imposing new semantics on the underlying structure, for example by changing the coordinate
223 systems or time resolution. This value agnostic model also supports encoding datasets where
224 there may be multiple independent variables without having to assume any one variable is
225 inducing the change , for example measures of plant growth given variations in water,
226 sunlight, and time. For building block library developers, a non-semantic model of data
227 continuity allows for the implementation of components that can traverse data structures
228 without having to know the semantics of the data. Since these building block components
229 are by design *equivariant* and *continuity preserving*, domain specific library developers in
230 different domains that rely on the same continuity, for example 2D continuity, can use the
231 same components to build tools that can make domain specific assumptions.

232 2.4 Contribution

233 In this work, we present a framework for understanding visualization as equivariant contin-
234 uity preserving maps between topological spaces. Using this mathematical formalism, we
235 develop an architecture specification developers can use to implement components in build-
236 ing block visualization libraries that domain specific library developers can carry through
237 in the tools they build. Our work diverges from previous models of visualization and imple-
238 mentations of those models in that it contributes

- 239 1. formalization of the topology preserving relationship between data and graphic via
240 continuous maps [subsubsection 3.2.2](#)

- 241 2. formalization of property preservation from data component to visual representation
 242 as equivariant maps [subsubsection 3.3.2](#)
- 243 3. functional oriented visualization architecture built on the mathematical model to
 244 demonstrate the utility of the model [subsubsection 3.3.3](#)
- 245 4. prototype of the architecture built on Matplotlib's infrastructure to demonstrate the
 246 feasibility of the model. [subsection 4.1](#)
- 247 We validate our model by using it to re-design the artist and data access layer of Matplotlib.
 248 We evaluate whether the redesign is successful by comparing it to existing implementation,
 249 recreating existing domain specific functionality with the new components, and by imple-
 250 menting components that provide new functionality in Matplotlib. While much of this
 251 functionality is currently possible in Matplotlib, a functional approach allows us to imple-
 252 ment components in a more robust, modular, reliable way.

253 3 Topological Equivariant Artist Model

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

$$\mathcal{A} : \mathcal{E} \rightarrow \mathcal{H} \tag{1}$$

- 254 The *artist* \mathcal{A} is a map from data \mathcal{E} to graphic \mathcal{H} fiber bundles. To explain how the *artist*
 255 is a structure preserving map from data to graphic, we first model data ([subsection 3.1](#)) and
 256 graphics ([subsection 3.2](#)) as topological structures that encapsulate component types and
 257 continuity. We then discuss the functional maps from graphic to data ([subsubsection 3.2.2](#)),
 258 data components to visual components ([subsubsection 3.3.2](#)), and visual components into
 259 graphic ([subsubsection 3.3.3](#)) that make up the artist.

260 **3.1 Data Space E**

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

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

261 that binds the components of the data in F to the continuity of the data encoded in K .
262 Our use of fiber bundles builds on Butler's work proposing that fiber bundles should be
263 the common data abstraction for visualization data [48, 49]. The fiber bundle models the
264 properties of data component types F ([subsubsection 3.1.1](#)), the continuity of records K
265 ([subsubsection 3.1.3](#)), the collections of records ([subsubsection 3.1.4](#)), and the space E of
266 all possible datasets with these components and continuity. By definition fiber bundles are
267 locally trivial [54, 55], meaning that over a localized neighborhood U the total space is the
268 cartesian product $K \times F$.

269 **3.1.1 Variables in Fiber Space F**

To formalize the structure of the data components, we use Spivak's description of the schema [51] as a fiber bundle to bind the components of the fiber to variable names and data types. Spivak constructs a set \mathbb{U} that is the disjoint union of all possible objects of types $\{T_0, \dots, T_m\} \in \mathbf{DT}$, where \mathbf{DT} are the data types of the variables in the dataset. He then defines the single variable set \mathbb{U}_σ

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

which is \mathbb{U} restricted to objects of type T bound to variable name c . The \mathbb{U}_σ 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

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

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

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

which can also be written as

$$F = F_0 \times \dots \times F_i \times \dots \times F_n \quad (6)$$

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

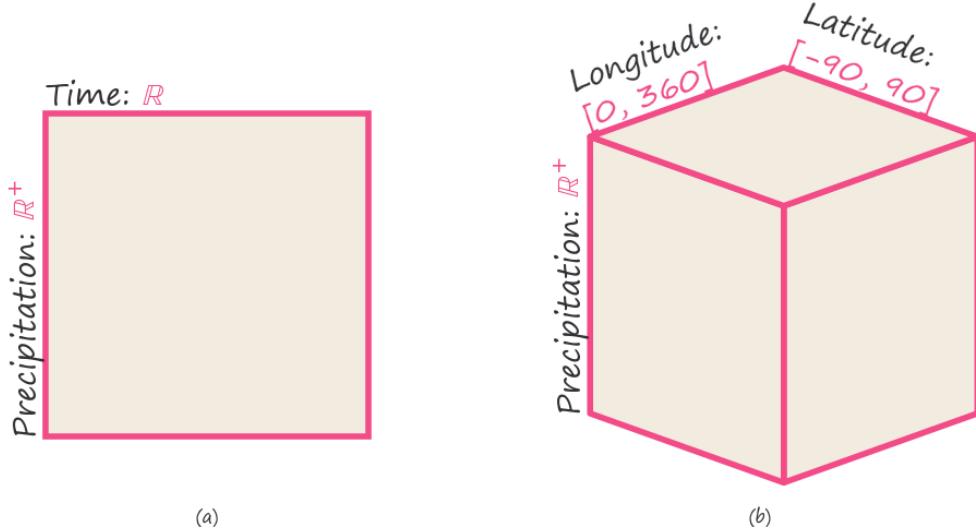


Figure 7: The fiber space is the cartesian product of the components. The 2D fiber $F = \mathbb{R} \times \mathbb{R}^+$ (a) encodes the properties of *time* and *precipitation* components. One dimension of the fiber encodes the range of possible values for the time component of the dataset, which is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

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

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

where the first component F_0 is the set of values specified by ($c = \text{time}$, $T = \text{datetime}$, $\mathbb{U}_\sigma = \mathbb{R}$) and F_1 is specified by ($c = \text{precipitation}$, $T = \text{float}$, $\mathbb{U}_\sigma = \mathbb{R}$) and is the set of values $\mathbb{U}_\sigma = \mathbb{R}$. In the 3D fiber (b) in [Figure 7](#), time is replaced with location. This location variable is of type `point` and has two components *latitude* and *longitude* $\{(lat, lon) \in \mathbb{R}^2 \mid -90 \leq lat \leq 90, 0 \leq lon \leq 360\}$. The fiber for this dataset is

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

271 with components ($c = \text{precipitation}$, $T = \text{float}$, $\mathbb{U}_\sigma = \mathbb{R}$), ($c = \text{latitude}$, $T = \text{float}$, $\mathbb{U}_\sigma =$
272 $[0, 360]$), and ($c = \text{longitude}$, $T = \text{float}$, $\mathbb{U}_\sigma = [-90, 90]$). By adapting Spivak's framework,
273 our model has a consistent way to describe the components of the data, no matter their
274 complexity.

275 **3.1.2 Measurement Scales: Monoid Actions**

276 Implementing expressive visual encodings requires formally describing the structure on the
277 components of the fiber. We conjecture that the structure can be formally defined as the
278 actions of a monoid on the component. We can also discuss complex structures since a core
279 property of monoids is composability [\[56\]](#). In doing so, we can specify the properties of the
280 component that must be preserved in a graphic representation.

A monoid [57] M is a set with a binary operation $* : M \times M \rightarrow M$ that satisfies the axioms:

associativity for all $a, b, c \in M$ $(a * b) * c = a * (b * c)$

identity for all $a \in M$, $e * a = a$

As defined on a component of F , a left monoid action [58, 59] of M_i is a set F_i with an action $\bullet : M \times F_i \rightarrow F_i$ with the properties:

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

identity for all $x \in F_i$, $e \in M_i$, $e \bullet x = x$

- ²⁸¹ We conjecture that given these properties, if there is a partial ordering on both sides of the
²⁸² action then the action is equivariant and monotonic.

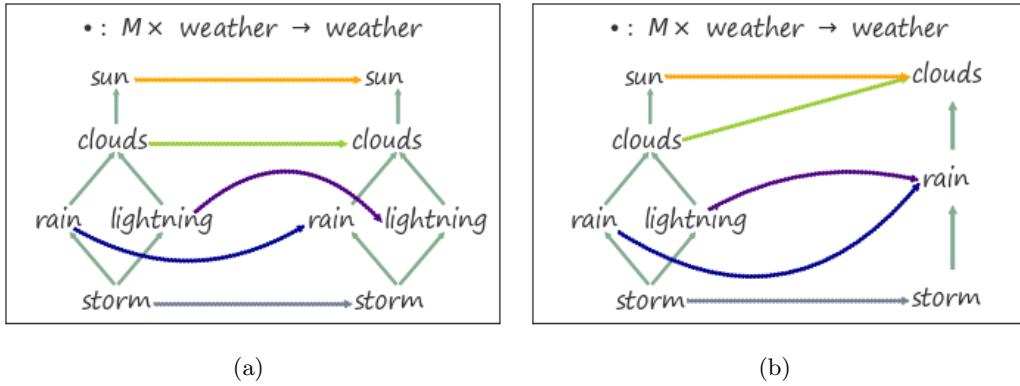


Figure 8: The action \bullet in Figure 8a is the arrows from the partial order diagram of weather states on the left to the diagram of weather states on the right. The action maps the weather states to themselves, thereby preserving the ordering defined by the monoid $*$ on both sides of the action.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

One example of monoids are partial orderings on a set, such as seen in [Figure 8](#). Each hasse diagram of the set of weather states describes an ordering on the set; the arrow goes from the lesser value to the greater one. For example, *storm* \leq *rain*. In [Figure 8](#), the action \bullet maps the elements of a set of weather states into itself by mapping them into other elements of the weather states. The action in [Figure 8a](#), represented as the arrows between the hasse diagrams of the weather states, maps the weather states to themselves; therefore the ordering of the weather states is identical on both sides of the action. The action \bullet in [Figure 8b](#) is a monotone map [60]

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

283 where the structure the action preserves is the relative, rather than exact, ordering. Since
284 groups are monoids with invertible operations, this definition of structure is broad enough
285 to include the Steven's measurement scales [61, 62].

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

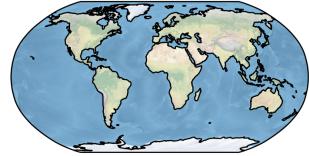
$$M = M_0 \times \dots \times M_i \times \dots \times \dots M_n \tag{7}$$

286 of each monoid M_i on F_i . The monoid is added to the specification of the fiber
287 $(c_i, T_i, \mathbb{U}_\sigma M_i)$

288 **3.1.3 Continuity of the Data**

| NAME | TEMP (°F) | PRCP (in.) |
|--------------------------|-----------|------------|
| NEW YORK LAGUARDIA AP | 61.00 | 0.4685 |
| BINGHAMTON | -12.00 | 0.0315 |
| NEW YORK JFK INTL AP | 49.00 | 0.7402 |
| ISLIP LI MACARTHUR AP | 11.00 | 0.0709 |
| SYRACUSE HANCOCK INTL AP | 13.00 | 0.0118 |

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{\sigma^2}}$$



(a)



(b)



(c)

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

289 The base space K provides a way to explicitly encode the continuity of the data, as de-
290 scribed in subsection 2.3. This explicit topology is a concise way of distinguishing between
291 visualizations that appear identical but assume different continuity, for example heat maps
292 and images. The base space K acts as an indexing space, as emphasized by Butler [48, 49],
293 to express how the records in E are connected to each other. As shown in Figure 9, K
294 can have any number of dimensions and can be continuous or discrete. Formally K is the
295 quotient space [63] of E meaning it is the finest space [64] such that every $k \in K$ has a
296 corresponding fiber F_k [63].

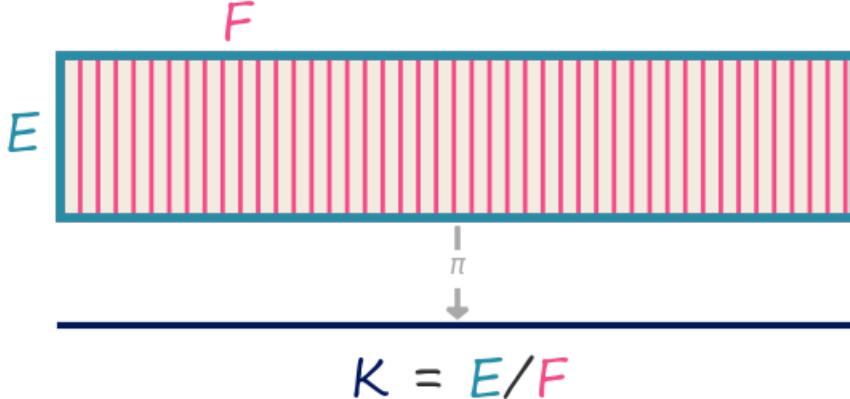


Figure 10: The total space E is divided into fiber segments F . The base space K acts as an index into the records in the fibers, such that every point k has a corresponding fiber F_k . The projection map π maps every fiber F_k to a point $k \in K$ in the base space.

297 In Figure 10, E is a rectangle divided by vertical fibers F , so the minimal K for which
 298 there is always a mapping $\pi : E \rightarrow K$ is the closed interval $[0, 1]$. While the total space
 299 E may have components in F that describe any given point $k \in K$, such as *time*, *latitude*,
 300 *longitude*, these labels are indexed into from K the same as any other components. In
 301 contrast to the structural *keys* with associated *values* proposed by Munzner [52], our model
 302 treats keys k as a pure reference to topology. Decoupling the keys from their semantics allows
 303 the components identifying the keys to be altered, which provides for a coordinate agnostic
 304 representation of the continuity and facilitates encoding of data where the independent
 305 variable may not be clear. For example total rainfall is dependent on time of day and how
 306 much rain has already fallen; therefore changing the coordinate system should have no effect
 307 on how the records are connected to each other, as illustrated in Figure 2 where precipitation
 308 in inches and millimeters yield equivalent line plots.

As with [Equation 6](#) and [Equation 7](#), we can decompose the total space into component bundles $\pi : E_i \rightarrow K$ where

$$\pi : E_1 \oplus \dots \oplus E_i \oplus \dots \oplus E_n \rightarrow K \quad (8)$$

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

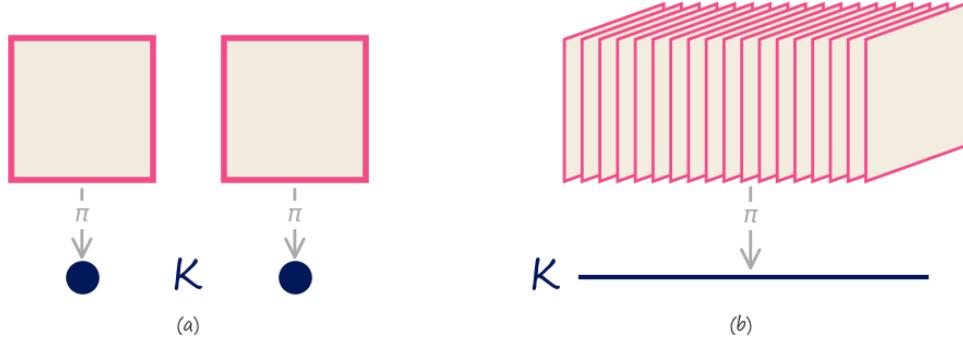


Figure 11: The fiber bundles in (a) and (b) encode the two component dataset from [Figure 7](#), with *(time, precipitation)* components, as having different continuities. The fiber bundle with discrete continuity (a) encodes the dataset as being a set of discrete records. The fiber bundle over the continuous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

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

318 **3.1.4 Data Values**

While the projection function $\pi : E \rightarrow K$ ties together the base space K with the fiber F , a section $\tau : K \rightarrow E$ 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 [51], K is a set of row ids, F is the columns, and the section τ returns the record r at a given key in K . For any fiber bundle, there exists a map

$$\begin{array}{ccc} F & \xhookrightarrow{\quad} & E \\ \pi \downarrow \lrcorner^{\tau} & & \\ K & & \end{array} \quad (9)$$

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

$$\tau(k) = (k, (g_{F_0}(k), \dots, g_{F_n}(k))) \quad (10)$$

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

$$\tau = (\tau_0, \dots, \tau_i, \dots, \tau_n) \quad (11)$$

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

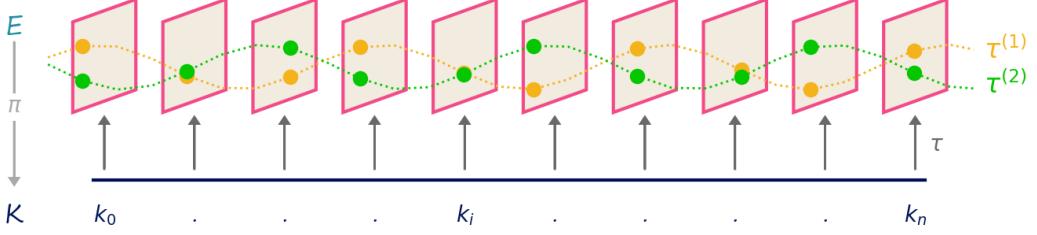


Figure 12: Fiber (time, precipitation) with a 1D continuous K defined on an interval $[0, n]$. The sections $\tau^{(1)}$ and $\tau^{(2)}$ are constrained such that the time variable must be monotonic, which means each section is a time series of precipitation values. They are included in the global set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

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

327 3.1.5 Sheafs

Dynamic visualizations require evaluating sections on different subspaces of K ; this can be achieved using a mathematical structure, called a sheaf \mathcal{O} , for defining collections of objects [65–67] on mathematical spaces. On the fiber bundle E , we can describe a sheaf as the collection of local sections $\iota^*\tau$

$$\begin{array}{ccc} \iota^*E & \xhookrightarrow{\iota^*} & E \\ \pi \downarrow \lrcorner \iota^*\tau & & \pi \downarrow \lrcorner \tau \\ U & \xhookrightarrow{\iota} & K \end{array} \quad (12)$$

328 which are sections of E pulled back over local neighborhood $U \subset E$ via the inclusion map
 329 $\iota : E \rightarrow U$. The collation of sections enabled by sheafs is necessary for navigation techniques
 330 such as pan and zoom [68] and dynamically updated visualizations such as sliding windows
 331 [69, 70].

332 **3.1.6 Applications**

333 Using fiber bundles as the data abstraction allows the model to describe widely used data
334 containers without sacrificing the semantic structure embedded in each container. For ex-
335 ample, the section can be any instance of a numpy array [71] that stores an image, such
336 as an image where the K is a 2D continuous plane and the F is $(\mathbb{R}^3, \mathbb{R}, \mathbb{R})$. In this fiber,
337 the \mathbb{R}^3 components encode color, and the other two components are the x and y positions
338 of the sampled data in the image. The continuity of the image is implicitly encoded in the
339 array as the index, so the position components encode the resolution. Instead of an image,
340 the numpy array could also store a 2D discrete table. The fiber may not change, but the K
341 would now be 0D discrete points. These different choices in topology indicate, for example,
342 what sorts of interpolation would be appropriate when visualizing the data. Labeled con-
343 tainers can also be described in this framework because of the schema like structure of the
344 fiber. One such example is a pandas series which stores a labeled list, another is a dataframe
345 [72] which has the structure of a relational table. A series could store the values of $\tau^{(1)}$ and
346 a second series could be $\tau^{(2)}$, while a dataframe would have multiple components and each
347 data frame would be a unique section τ . The ability to encode complexity in continuity
348 and components is particularly beneficial when working with N dimensional labeled data
349 containers. For example, an xarray [46] data cube that stores precipitation would be a sec-
350 tion of a fiber bundle with a K that is a continuous volume and components (*time, latitude,*
351 *longitude, precipitation*). This section does not need to resolve to values immediately and
352 instead can be an instance of a distributed data container, such as a dask array [73].

353 **3.2 Graphic Space H**

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

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

$$\begin{array}{ccc} D & \xhookrightarrow{\quad} & H \\ \pi \downarrow & \nearrow \rho & \\ S & & \end{array} \quad (13)$$

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

358 **3.2.1 Idealized Display D**

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

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

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

363 **3.2.2 Continuity of the Graphic S**

For a visualization component to preserve continuity, we propose that there must exist a structure preserving surjective map $\xi : S \rightarrow K$ from the data base space K to the graphic base space S . Formally, we require that K be a deformation retract [74] of S such that K and S have the same homotopy, meaning there is a continuous map from S to K [75]. The surjective map $\xi : S \rightarrow K$

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

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

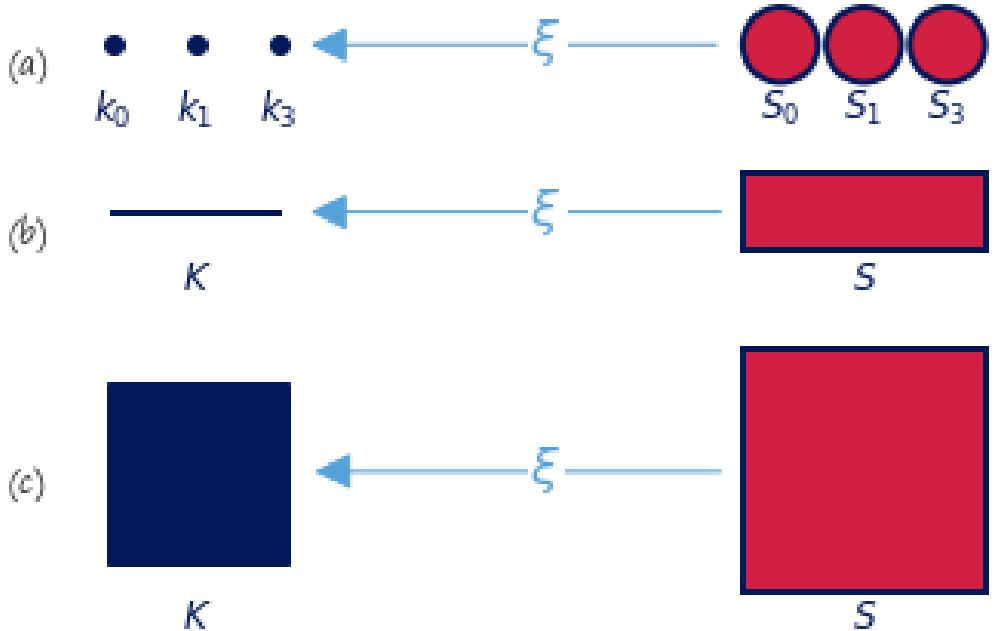


Figure 13: For a visualization component to preserve continuity, it must have a continuous surjective map $\xi : S \rightarrow K$ from graphic continuity to data continuity. The scatter (a) and line (b) graphic base spaces S have one more dimension of continuity than K so that S can encode physical aspects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

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

374 2D (a) to encode the connectivity of the pixels that visually represent the point. This is also
 375 the case with the line (b), which would be infinitely thin on screen if S was not thickened
 376 to 2D. This thickening is often not necessary when the dimensionality of K matches the
 377 dimensionality of the target space, for example if K is 2D and the display is a 2D screen
 378 (c). Since the mapping function ξ binds the graphic base space to the data base space, it
 379 can be used by interactive visualization components to look up the data associated with a
 380 region on screen. One example is to fill in details in a hover tooltip, another is to convert
 381 region selection (such as zooming) on S to a query on the data to access the corresponding
 382 record components on K .

383 3.2.3 Graphic

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

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

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

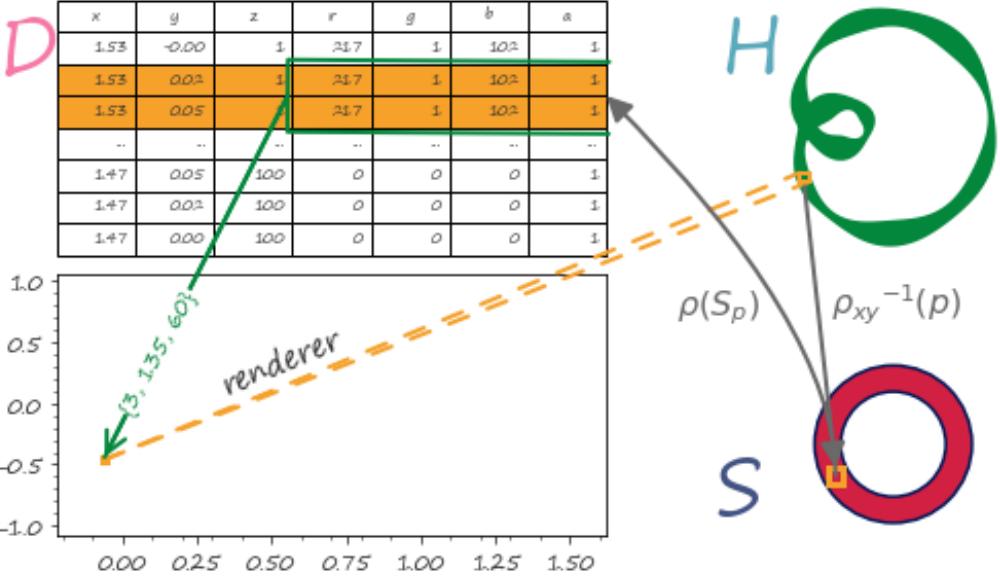


Figure 14: To render a graphic, a pixel p is selected in the display space, which is defined in the same coordinates as the x and y components in D via the renderer. Therefore, the pixel p maps to a region on H . In H the inverse mapping $\rho_{x,y}(p)$ returns a region $S_p \subset S$. $\rho(S_p)$ returns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

388 As shown in [Figure 14](#), a pixel p in the output space, drawn in yellow, is selected and
 389 mapped, via the renderer, into a region on H . The region on H corresponds to a region
 390 $S_p \subset S$ via the inverse mapping $\rho_{x,y}(p)$. The base space S is an annulus to match the
 391 topology of the graphic idealized in H . The section $\rho(S_p)$ then maps into the fiber D over
 392 S_p to obtain the set of points in D , here represented as a table, that correspond to that
 393 section. The integral over the pixel components of this set of points in the fiber yields the
 394 color of the pixel. In general, ρ is an abstraction of rendering. In very broad strokes ρ can
 395 be a specification such as PDF [\[76\]](#), SVG [\[77\]](#), or an OpenGL scene graph [\[78\]](#) or a rendering
 396 engine such as cairo [\[79\]](#) or AGG [\[80\]](#). Implementation of ρ is out of scope for this proposal.

397 3.3 Artist

We propose that visualization is structure preserving maps from data E to graphic H ; having described E in [subsection 3.1](#) and H in [subsection 3.2](#), we now define the visual

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

$$A : \mathcal{O}(E) \rightarrow \mathcal{O}(H) \quad (15)$$

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

$$A(m \cdot r) = \varphi(m) \cdot A(r) \quad (16)$$

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

$$\varphi : M \rightarrow M' \quad (17)$$

398 such that an action in data space $m \in M$ is equivalent to an action in graphic space
399 $\varphi(M) \in M'$.

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

$$\begin{array}{ccccccc} E & \xrightarrow{\nu} & V & \xleftarrow{\xi^*} & \xi^*V & \xrightarrow{Q} & H \\ & \searrow \pi & \downarrow \pi & & \xi^* \pi \downarrow & \swarrow \pi & \\ & & K & \xleftarrow{\xi} & S & & \end{array} \quad (18)$$

400 ξ^*V is the visual bundle V pulled back over S via the equivariant continuity map $\xi : S \rightarrow K$
401 introduced in [subsubsection 3.2.2](#). The functional decomposition of the visualization artist
402 in [Equation 18](#) facilitates building reusable components at each stage of the transformation
403 because the equivariance constraints are defined on ν , Q , and ξ . We name this map the

404 artist as that is the analogous part of the Matplotlib [7] architecture that builds visual
405 elements.

406 **3.3.1 Visual Fiber Bundle V**

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

$$P \xhookrightarrow{\quad} V \underset{\pi \downarrow}{\overset{\wedge}{\longrightarrow}} \mu \quad K \quad (19)$$

407 where μ is the visual variable encoding, as described by Bertin [9], of the data section τ .
408 The visual bundle V is the full design space [81] of possible parameters of a visualization
409 type, such as a scatter plot or line plot. For example, one section μ of V is a tuple of visual
410 values that specifies the visual characteristics of a part of a graphic.

| ν_i | μ_i | $\text{codomain}(\nu_i) \subset P_i$ |
|----------|--|---|
| 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

411 In [Table 1](#), the fiber components are specified by the visual parameter they are encod-
 412 ing. Multiple parameters can be encoded with the same transformation from data space
 413 to graphic space, for example x and y are both positions on a screen. Given a fiber of
 414 $\{x, y, color\}$ one possible section could be $\{.5, .5, (255, 20, 147)\}$. The $\text{codomain}(\nu_i)$ in [Ta-](#)
 415 [ble 1](#) specifies the libraries internal representation of visual variables and can be used to
 416 determine which monoids can act on P_i .

417 **3.3.2 Visual Encoders**

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

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

$$\{\nu_0, \dots, \nu_n\} : \{\tau_0, \dots, \tau_n\} \mapsto \{\mu_0, \dots, \mu_n\} \quad (20)$$

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

$$\begin{array}{ccc} E_i & \xrightarrow{\nu_i} & V_i \\ m_r \downarrow & & \downarrow m_v \\ E_i & \xrightarrow{\nu_i} & V_i \end{array} \quad (21)$$

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

$$\nu_i(m_r(E_i)) = \varphi(m_r)(\nu_i(E_i)) \quad (22)$$

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

426 The mapping from weather state to umbrella in [Figure 15a](#) is monotonic, and there-
427 fore we conjecture equivariant, because $\nu(rain) = \nu(storm)$ satisfies the monotonic con-

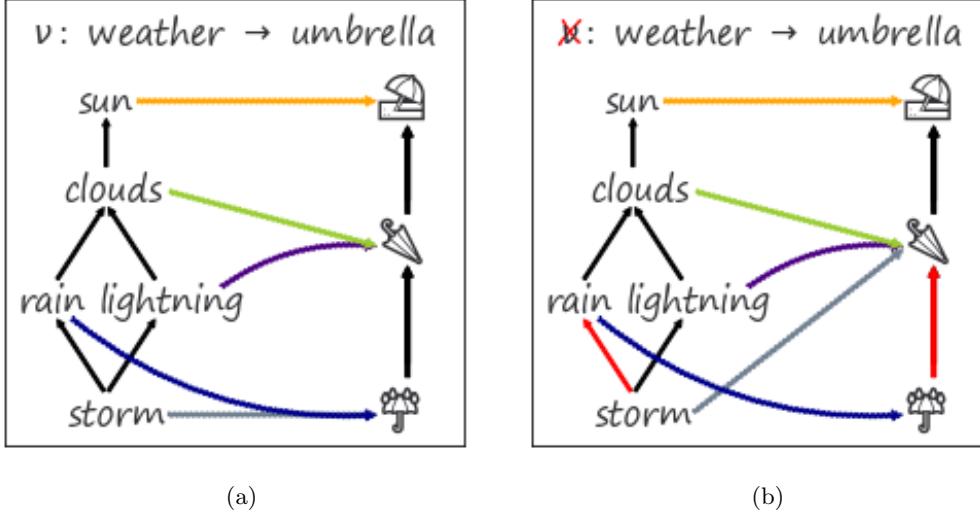


Figure 15: The ν mapping in Figure 15a represented by the colored arrows is monotonic, and therefore monoid equivariant, since $\nu(\text{storm}) = \nu(\text{rain})$ satisfies the condition $\nu(\text{storm}) \geq \nu(\text{rain})$. In contrast, the map from data component to visual component in Figure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

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

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

Table 2: Equivariance constraints for the Stevens' measurement scales[82]

436 The Stevens measurement types [61], listed in [Table 2](#), are specified in terms of groups,
 437 which are monoids with invertible operations[83]. We generalize to monoids to account for
 438 limitations in the types of data that can be described with the Stevens' scales [84, 85]

439 **3.3.3 Visualization Assembly**

440 Having described the maps to components in [subsubsection 3.3.2](#), we now specify the assem-
 441 bly function \hat{Q} that composites components in V into a graphic in H . Since the component
 442 transforms ν are equivariant, the equivariance constraints carry through to \hat{Q} . We specify
 443 these constraints to guide the implementation of library components responsible for gener-
 444 ating graphics.

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

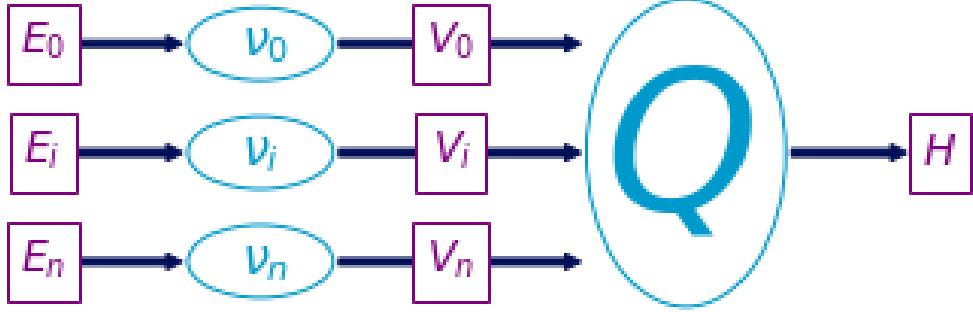


Figure 16: The transform functions ν_i convert data $\tau_i \in E$ to visual characteristics $\mu_i \in V$. These visual components μ_i are then assembled by Q into a graphic $\rho \in H$.

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



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

456
 457 the visual representations of the components $\Gamma(V) = X$ and the graphic $Q(\Gamma(V)) = Y$

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

$$Q(\mu) = Q(\mu') \implies Q(m \circ \mu) = Q(m \circ \mu') \quad (23)$$

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

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

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

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

where the path [86] γ from k to j is a continuous function from the interval $[0, 1]$. We define the glyph as the graphic generated by $Q(S_j)$

$$H \xrightleftharpoons[\rho(S_j)]{} S_j \xrightleftharpoons[\xi^{-1}(J)]{\xi(s)} J_k \quad (25)$$

465 such that for every glyph there is at least one corresponding region on K , in keeping with
 466 the definition of glyph as any visually distinguishable element put forth by Ziemkiewicz and
 467 Kosara [87]. The primitive point, line, and area marks [9, 88] are specially cased glyphs.

468 3.3.4 Assembly Q

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

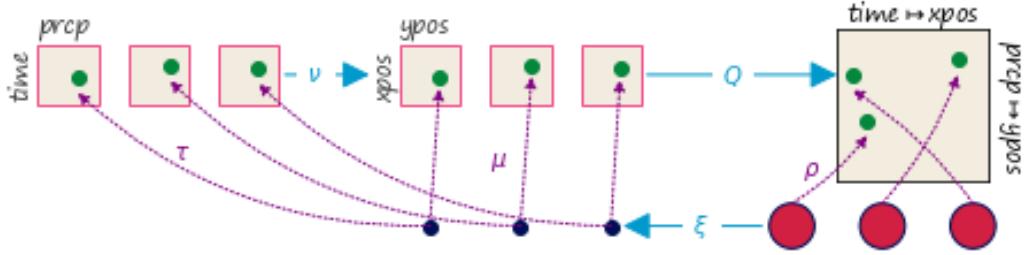


Figure 18: The data is discrete points (time, precipitation). Via ν these are converted to (xpos, ypos) and pulled over discrete S via ξ^* . The pulled back visual section ν is composed with the assembly function $\hat{Q} \circ \nu = \rho$ to produce the instructions to make the graphic ρ . The graphic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

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

$$(26) \quad Q(xpos, ypos)(\alpha, \beta)$$

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

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

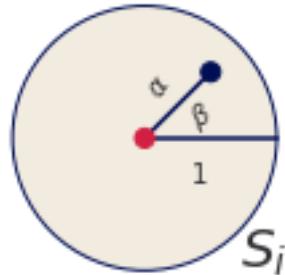


Figure 19: The simplest form of the scatter plot takes as input the expected position of the marker shape ($xpos, ypos$). The marker shape is determined by the polar coordinates (α, β) on the disc; these coordinates dictate whether anything is drawn at that region of S . To obtain the color of the pixel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

474 The position of this swatch of color is computed relative to the location on the disc
 475 $(\alpha, \beta) \in S_k$ as shown in Figure 19. The region α, β is scaled by a constant size and shifted
 476 by $xpos$ and $ypos$. This computation yields the values (x, y) that map into D and have a
 477 corresponding function $\rho(s) = (x, y, 0, 0, 0)$ which colors the point (x, y) black.

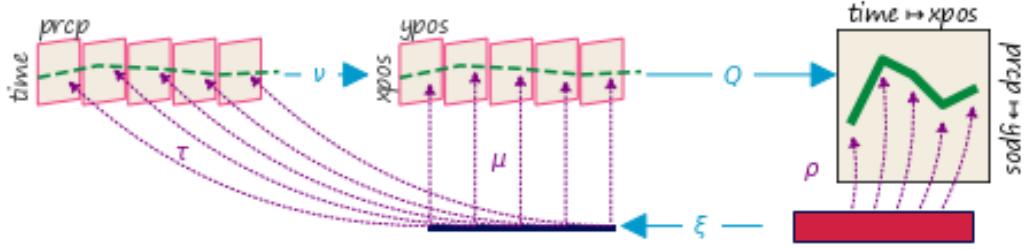


Figure 20: The line fiber (*time, precipitation*) is thickened with the derivative (*time', precipitation'*) because that information will be necessary to figure out the tangent to the point to draw a line. This is because the line needs to be pushed perpendicular to the tangent of (*xpos, ypos*). The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

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

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

$$\begin{aligned} |n| &= \sqrt{n_1^2(\xi(\alpha)) + n_2^2(\xi(\alpha))} \\ \hat{n}_1 &= \frac{n_1(\xi(\alpha))}{|n|}, \quad \hat{n}_2 = \frac{n_2(\xi(\alpha))}{|n|} \end{aligned}$$

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

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

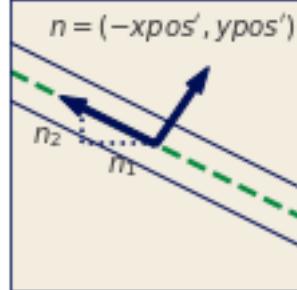


Figure 21: The *xpos* and *ypos* variables give the position of the line in screen space, but render an infinitely thin line. To draw equidistant lines parallel to $(xpos, ypos)$, defined by the distance (n_1, n_2) , requires the derivatives $(n_1 = xpos', n_2 = ypos')$. The position $(xpos, ypos)$ and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

480 As shown in [Figure 21](#), line needs to know the tangent of the data to draw an envelope
 481 above and below each $(xpos, ypos)$ such that the line appears to have a thickness; therefore
 482 the artist takes as input the jet bundle [89, 90] $\mathcal{J}^2(E)$ which is the data E and the first
 483 and second derivatives of E . The indexing map $\xi(\alpha)$ finds the point in K corresponding
 484 to the region in S at coordinate α . The section τ on the k that corresponds to the region
 485 in S returns the position $xpos$, $ypos$ and the derivatives \hat{n}_1, \hat{n}_2 . The derivatives are then
 486 multiplied by a width parameter to specify the thickness of the line. This is then used to
 487 determine the color of the pixel at (x, y) .

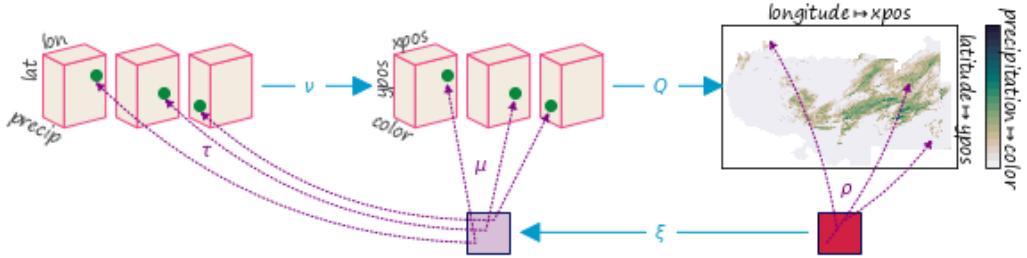


Figure 22: Via ξ the artist maps from a point (x, y) on the screen to a corresponding point on K . This maps into F via τ . These data points are converted to visual points via ν and then Q assembles the $(xpos, ypos, color)$ parameters into attributes of each pixel.

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

$$Q(xpos, ypos, color)(\alpha, \beta) \quad (28)$$

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

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

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

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

491 **3.3.5 Assembly Template \hat{Q}**

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

$$\begin{array}{ccccc}
 E & \xrightarrow{\nu} & V & \xleftarrow{\xi^*} & \xi^*V \xrightarrow{Q} H \\
 & \searrow \pi & \downarrow \mu & \uparrow \xi^*\pi & \swarrow \pi \\
 & K & \xleftarrow{\xi} S & &
 \end{array} \tag{29}$$

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

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

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

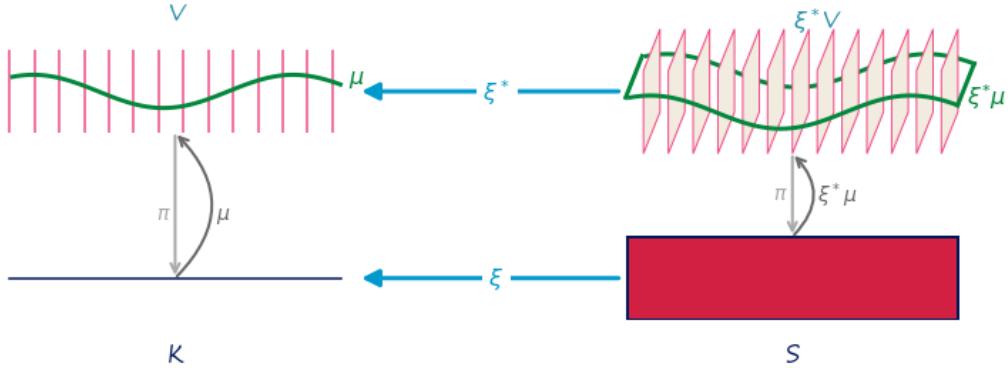


Figure 23: Because the pullback of the visual bundle ξ^*V is the replication of a μ over all points s that map back to a single k , we can construct a \hat{Q} on μ over k that will fabricate the Q for the equivalent region of s associated to that k

494 As shown in Figure 23, given the section $\xi^*\mu$ pulled back from μ and the point $s \in \xi^{-1}(k)$,
495 there is mapping from section $\xi^*\mu$ over s to μ over k . This means that the pulled back section
496 $\xi^*\mu(s) = \xi^*(\mu(k))$ is the section μ copied over all s such that $\xi^*\mu$ is identical for all s where
497 $\xi(s) = k$. In Figure 23 each dot on P is equivalent to the line on $P^*\mu$.

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

$$\rho(s) := Q(\xi^*\mu)(s) \quad (31)$$

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

$$\hat{Q}(\mu(k))(s) := Q((\xi^*\mu)(s)) \quad (32)$$

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

can then factor s out of [Equation 32](#), which yields

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

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

3.3.6 Composition of Artists: +

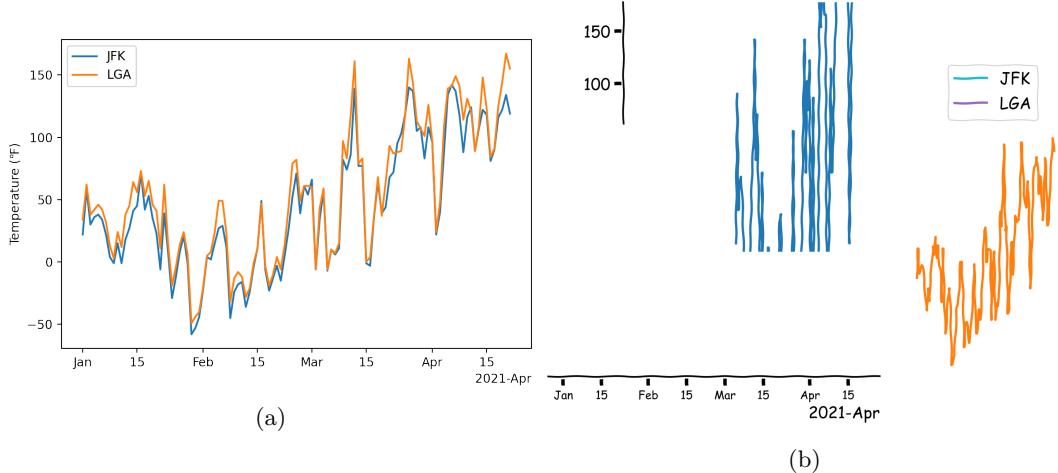


Figure 24: In [Figure 24a](#), these artists are composited before being added to the image. Disjoint union of E aligns the two time series with the x and y axis so all these elements use a shared coordinate system. A more complex composition dictates that the legend is connected to the E such that it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

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

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

$$+ := \bigsqcup_{i \in I} E_i \quad (34)$$

506 defines a simple composition of artists. For example, in [Figure 24a](#) the data is joined via
 507 disjoint union; doing so aligns the components in F such the ν to the same component in
 508 P targets the same coordinate system. In [Figure 24b](#), these artists are all added to the
 509 image independently of the other and therefore there are no constraints on where they are
 510 placed in the image. When artists share a base space $K_2 \hookrightarrow K_1$, a composition operator
 511 can be defined such that the artists are acting on different components of the same section.
 512 This type of composition is important for visualizations where elements update together in
 513 a consistent way, such as multiple views [91, 92] and brush-linked views [93, 94].

514 3.3.7 Equivalence class of artists

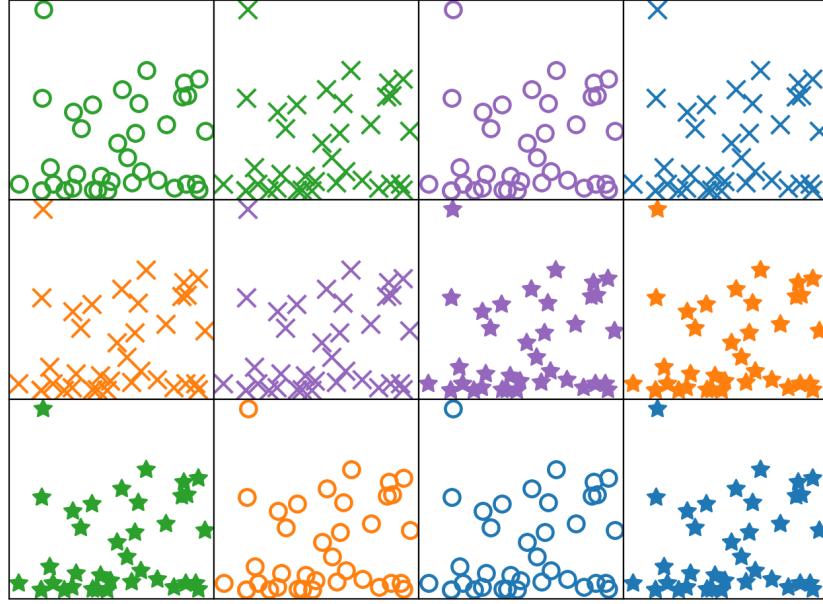


Figure 25: Each scatter plot is generated via a unique artist function A_i , but they only differ in aesthetic styling. Therefore, these artists are all members of an equivalence class $A_i \in A'$

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

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

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

521 4 Prototype: Matplottoy

522 To evaluate the feasibility of the model described in [section 3](#), we built prototypes of a
 523 `point`, `line`, and `bar` artist. We make use of the Matplotlib Figure and Axes artists [6,
 524 7] so that we can initially focus on the data to graphic transformations and exploit the
 525 Matplotlib transform stack to convert data coordinates into screen coordinates. While the
 526 artist is specified in a fully functional manner in [Equation 18](#), we implement the prototype
 527 in a heavily object oriented manner. This is done to manage function inputs, especially
 528 parameters that are passed through to methods that are structurally functional.

```

1 fig, ax = plt.subplots()
2 artist = ArtistClass(E, V)
3 ax.add_artist(artist)

```

529 Building on the current Matplotlib Artists, which construct an internal representa-
 530 tion of the graphic, `ArtistClass` acts as an equivalence class artist A' as described
 531 in [Figure 3.3.7](#). The visual bundle V is specified as the `V` dictionary of the form
 532 `{parameter:(variable name, encoder)}` where parameter is a component in P , variable
 533 is a component in F , and the ν encoders are passed in as functions or callable objects. The
 534 data bundle E is passed in as a `E` object. By binding data and transforms to A' inside
 535 `__init__`, the `draw` method is a fully specified artist A as defined in [Equation 15](#).

```

1  class ArtistClass(matplotlib.artist.Artist): #A'
2      def __init__(self, E, V, *args, **kwargs):
3          # properties that are specific to the graphic
4          self.E = E
5          self.V = V
6          super().__init__(*args, **kwargs)
7
8      def q_hat(self, **args):
9          # set the properties of the graphic
10
11     def draw(self, renderer):
12         # returns  $K$ , indexed on fiber then key
13         tau = self.E.view(self.axes)
14         # visual channel encoding applied component wise
15         mu = {p: nu(tau(c))
16               for p, (c, nu) in self.V.items()}
17         self.q_hat(**mu)
18         # pass configurations off to the renderer
19         super().draw(renderer)

```

536 The data in section τ is fetched via a `view` method on the data because the input to the
 537 artist is a section on E . The `view` method takes the `axes` attribute because it provides the
 538 region in graphic coordinates S that can be used to query back into data to select a subset
 539 as described in [subsubsection 3.1.5](#). We require that the `view` method be atomic so that
 540 we do not risk race conditions. Atomicity means that the values cannot change after the
 541 method is called in `draw` until a new call to `draw`[29], which ensures the integrity of the
 542 section.

543 The ν functions are then applied to the data, as described in [Equation 20](#), to generate
 544 the visual section μ that here is the object V . The conversion from data to visual space is
 545 simplified here to directly show that it is the encoding ν applied to the component. The
 546 `q_hat` function that is \hat{Q} , as defined in [Equation 33](#), is responsible for generating a repre-
 547 sentation such that it could be serialized to recreate a static version of the graphic. The last
 548 step in the artist function is handing itself off to the renderer. The extra `*arg`, `**kwargs`
 549 arguments in `__init__`, `draw` are artifacts of how these objects are currently implemented.

550 4.1 Scatter, Line, and Bar Artists

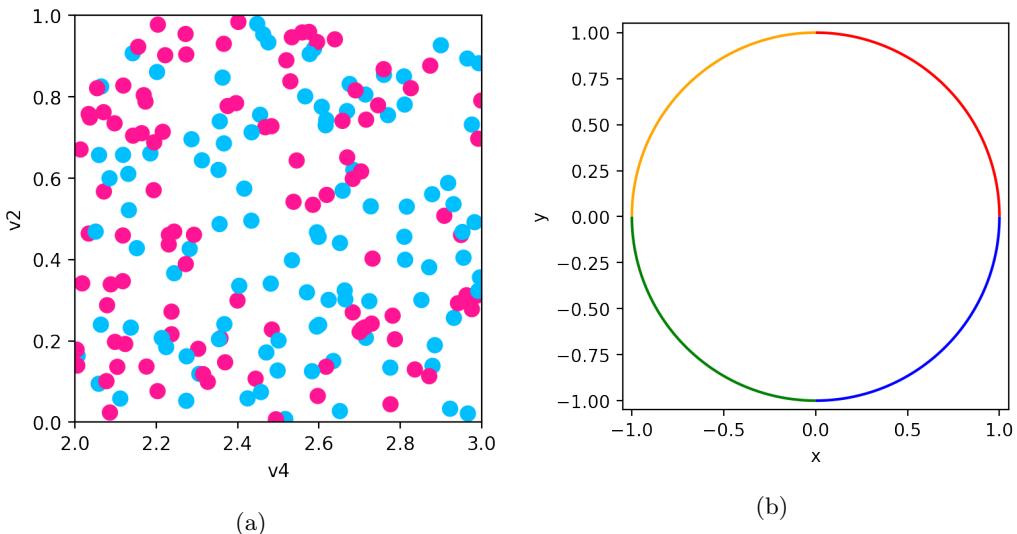


Figure 26: Scatter plot and line plot implemented using `Point` and `Line` artists and fiber bundle inspired data models. Matplotlib is used for the rendering.

551 The figure in [Figure 26a](#) is a scatter plot, as described by [Equation 26](#). This is implemented
 552 via a `Line` object where the scatter marker shape is fixed as a circle, and the visual fiber
 553 components are x and y position and the facecolor and size of the marker. We only show the
 554 `q_hat` function here because the `__init__`, `draw` are inherited from the prototype artist
 555 `ArtistClass`.

556 The `view` method repackages the data as a fiber component indexed table of vertices.
 557 Even though the `view` is fiber indexed, each vertex at an index k has corresponding values
 558 in section $\tau(k_i)$. This means that all the data on one vertex maps to one glyph.

```

1 class Point(ArtistClass, mcollections.Collection):  

2     def q_hat(self, x, y, s, facecolors): #\hat{Q}  

3         # construct geometries of circle glyphs  

4         self._paths = [mpath.Path.circle((xi,yi), radius=si)  

5                         for (xi, yi, si) in zip(x, y, s)]  

6         # set attributes of glyphs, these are vectorized  

7         # circles and facecolors are lists of the same size  

8         self.set_facecolors(facecolors)

```

559 In `q_hat`, the μ components are used to construct the vector path of each circular marker
 560 with center (x,y) and size x and set the colors of each circle. This is done via the
 561 `Path.circle` object.

```

1 class Line(ArtistClass, mcollections.LineCollection):  

2     def q_hat(self, x, y, color): #\hat{Q}  

3         #assemble line marks as set of segments  

4         segments = [np.vstack((vx, vy)).T for vx, vy  

5                         in zip(x, y)]  

6         self.set_segments(segments)  

7         self.set_color(color)

```

562 To generate [Figure 26b](#), the `Line` artist `view` method returns a table of edges. Each edge
 563 consists of (x,y) points sampled along the line defined by the edge and information such as
 564 the color of the edge. As with `Point`, the data is then converted into visual variables. In
 565 `q_hat`, described by [Equation 27](#), this visual representation is composed into a set of line

566 segments, where each segment is the array generated by `np.vstack((vx, vy))`. Then the
 567 colors of each line segment are set. The colors are guaranteed to correspond to the correct
 568 segment because of the atomicity constraint on the view. The implementation of line in
 569 Matplotlib does not have this functionality because it has no notion of rows and columns
 570 of a table, and therefore no notion of a τ . Instead, line is assumed to be points along one
 571 edge and therefore has only one color, and the user is responsible for aligning the x and y
 572 components and colors along the implicit K over which they are plotted.

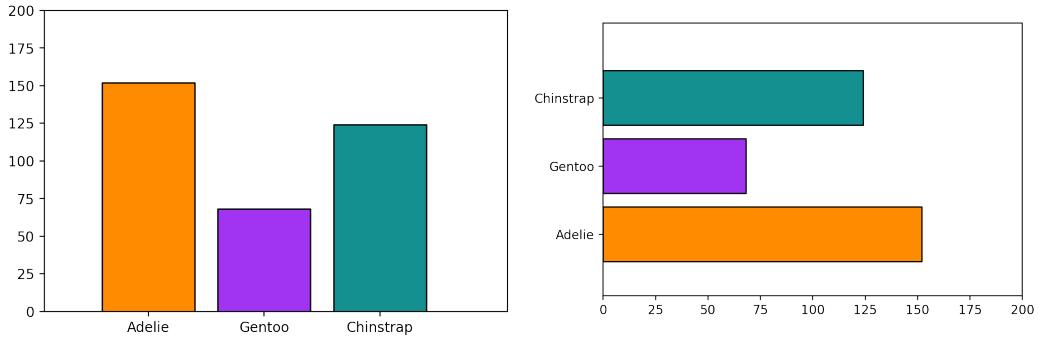


Figure 27: Frequency of Penguin types visualized as discrete bars.

573 The bar charts in figure 27 are generated with a `Bar` artist. The artist has required
 574 visual parameters P of (position, length), and an additional parameter `orientation` which
 575 controls whether the bars are arranged vertically or horizontally. This parameter only applies
 576 holistically to the graphic and never to individual data parameters, and highlights how the
 577 model encourages explicit differentiation between parameters in V and graphic parameters
 578 applied directly to \hat{Q} .

```

1 class Bar(ArtistClass, mcollections.Collection):
2     def __init__(self, E, V, orientation, *args, **kwargs):
3         """
4             orientation: str
5                 v: bars aligned along x axis, heights on y
6                 h: bars aligned along y axis, heights on x

```

```

7      """
8
9         self.orientation = orientation
10        super().__init__(*args, **kwargs) # set E & V
11
12    def q_hat(self, position, length, floor, width,
13              facecolors, edgecolors, offset):
14
15        position = position + offset
16
17        def make_bars(xval, xoff, yval, yoff):
18
19            return [(x, y), (x, y+yo), (x+xo, y+yo), (x+xo, y), (x, y)]
20
21            for (x, xo, y, yo) in zip(xval, xoff, yval, yoff)]
22
23            #build bar glyphs based on graphic parameter
24
25            if self.orientation in {'vertical', 'v'}:
26
27                verts = make_bars(position, width, floor, length)
28
29            elif self.orientation in {'horizontal', 'h'}:
30
31                verts = make_bars(floor, length, position, width)
32
33
34            self._paths = [mpath.Path(xy, closed=True) for xy in verts]
35
36            self.set_edgecolors(edgecolors)
37
38            self.set_facecolors(facecolors)

```

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

584 **4.2 Visual Encoders**

585 The visual parameter serves as the dictionary key because the visual representation is con-
586 structed from the encoding applied to the data $\mu = \nu \circ \tau$. For the scatter plot, the mappings
587 for the visual fiber components $P = (x, y, facecolors, s)$ are defined as

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

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

```
1  #\nu_i(m_r(E_i)) = \varphi(m_r)(\nu_i(E_i))
2  def test_nominal(values, encoder):
3      m1 = list(zip(values, encoder(values)))
4      random.shuffle(values)
5      m2 = list(zip(values, encoder(values)))
6      assert sorted(m1) == sorted(m2)
```

591 As described in [Equation 22](#), a test for equivariance can be implemented trivially. The test
592 shown here is the nominal test described in [Table 2](#) because `Categorical` is intended to
593 encode nominal data. Equivariance tests are currently factored out of the artist for clarity.

594 **4.3 Data Model**

595 The data input into the `Artist` will often be a wrapper class around an existing data
596 structure. This wrapper object must specify the fiber components F and connectivity K
597 and have a `view` method that returns an atomic object that encapsulates τ . To support
598 specifying the fiber bundle, we define a `FiberBundle` data class [95]

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

599 that asks the user to specify the the properties of F and the K connectivity as either discrete
600 vertices or continuous data along edges. To generate the scatter plot and the line plot, the
601 distinction is in the `tau` method that is the section.

```
1 class PointData:
2     def __init__(self):
3         self.FB = FiberBundle({'tables': ['vertex']},
4                               {'v1': float, 'v2': str, 'v3': float})
5     def tau(self, k):
6         return # tau evaluated at one point k
7
8 class LineData:
9     def __init__(self):
10        self.FB = FiberBundle({'tables': ['edge']},
11                             {'x': float, 'y': float, 'color': str})
12     def tau(self, k):
13         return # tau evaluated on interval k
```

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

```

1 def view(self, axes):
2     table = defaultdict(list)
3     for k in self.keys():
4         table['index'].append(k)
5         for (name, value) in zip(self.FB.fiber.keys(),
6                               self.tau(k)[1]):
7             table[name].append(value)
8
9     return table

```

605 In both cases, the `view` method packages the data into a data structure that the artist can
 606 unpack via data component name, akin to a table with column names when K is 0 or 1 D.

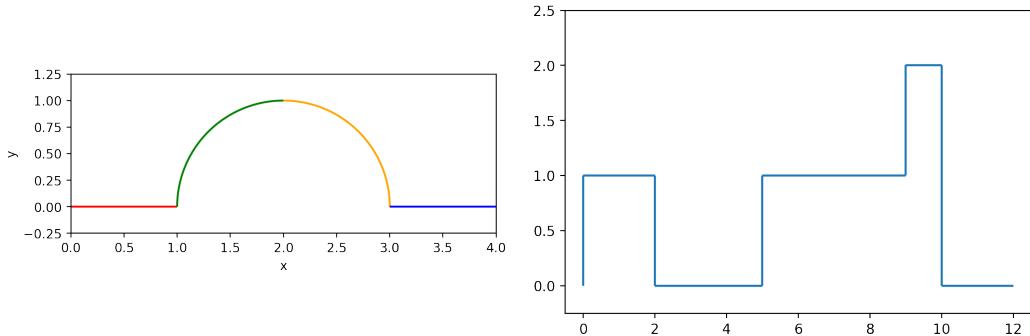


Figure 28: Continuous and discontinuous lines as defined via the same data model, and generated with the same A' Line

607 The graphics in figure Figure 28 are made using the `Line` artist and the `GraphData`
 608 data source where if told that the data is connected, the data source will check for that
 609 connectivity by constructing an adjacency matrix. The multicolored line is a connected
 610 graph of edges with each edge function evaluated on 100 samples,

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

611 which is an arbitrary choice made to display a smooth curve. The axes can also be used
612 to choose an appropriate number of samples. In contrast, the stair chart only needs to be
613 evaluated at the edges of the interval

```
1 GraphData(FB, edges, vertices, num_samples=2, connect=False)
```

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

617 4.4 Case Study: Penguins

618 Building on the Bar artist in subsection 4.1, we implement grouped bar charts as these do
619 not exist out of the box in the current version of Matplotlib. Instead, grouped bar charts
620 are often achieved via looping over an implementation of bar, which forces the user to keep
621 track which values are mapped to a single visual element and how that is achieved. For this
622 case study, we use the Palmer Penguins dataset [96, 97], packaged as a pandas data frame
623 [98] since that is a very commonly used Python labeled data structure.

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

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

624 The wrapper is very thin because there is explicitly only one section, which is the pen-
625 guins dataframe.

```

1  class DataFrame:
2
3      def __init__(self, dataframe):
4          self.FB = FiberBundle(K = {'tables':['vertex']},
5                               F = dict(dataframe.dtypes))
6
7          self._tau = dataframe.iloc
8
9          self._view = dataframe

```

626 Since the aim for this wrapper is to be very generic, here the fiber is set by querying the
 627 dataframe for its metadata. The `dtypes` are a list of column names and the datatype of
 628 the values in each column; this is the minimal amount of information the model requires to
 629 verify constraints. The pandas indexer is a key valued set of discrete vertices, so there is no
 630 need to repackage for the data interface.

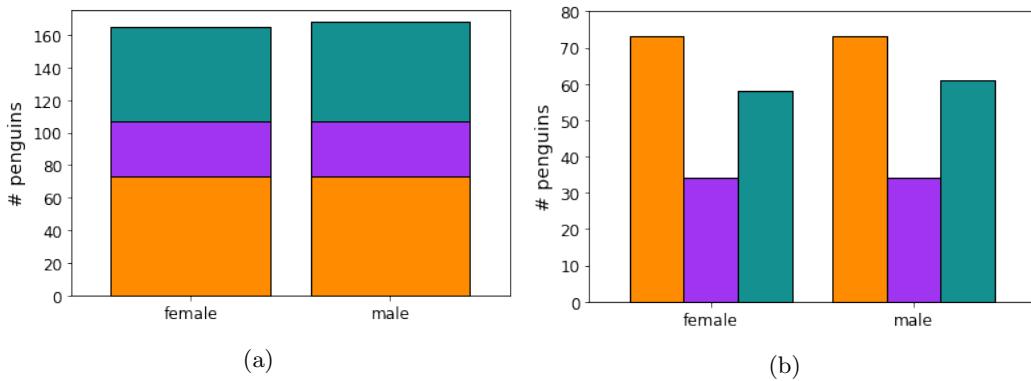


Figure 29: Penguin count disaggregated by island and species

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

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

```
1 class StackedBar(martist.Artist):
2     def __init__(self, E, V, MV, orientation='v', *args, **kwargs):
3         """
4             Parameters
5             -----
6
7             orientation: str, optional
8                 vertical: bars aligned along x axis, heights on y
9                 horizontal: bars aligned along y axis, heights on x
10            """
11
12     super().__init__(*args, **kwargs)
13     self.E = E
14     self.orientation = orientation
15     self.V = V
16     self.MV = MV
17
18     def q_hat(self):
19         tau = self.data.view(self.axes)
20         self.children = [] # list of bars to be rendered
21         floor = 0
22         for group in self.MV:
23             # pull out the specific group transforms
24             bar = Bar(self.E, {**group, **self.V, 'floor':floor},
```

```

24                     self.orientation, transform=self.axes.transData)
25
26             self.children.append(bar)
27
28             floor += view[group['length']['name']]
29
30
31     def draw(self, renderer, *args, **kwargs):
32
33         # all the visual conversion gets pushed to child artists
34
35         self.assemble()
36
37         #self._transform = self.children[0].get_transform()
38
39         for artist in self.children:
40
41             artist.draw(renderer, *args, **kwargs)

```

640 The visual transformations are passed through to Bar; therefore, the draw method does
641 not do any visual transformations. In StackedBar the view is used to adjust the floor for
642 every subsequent bar chart, since a stacked bar chart is bar chart area marks concatenated
643 together in the length parameter. In contrast, GroupedBar does not even need the view, but
644 instead keeps track of the relative position of each group of bars in the visual only variable
645 offset.

```

1  class GroupedBar(martist.Artist):
2
3      def q_hat(self):
4
5          self.children = [] # list of bars to be rendered
6
7          ngroups = len(self.mtransforms)
8
9
10         for gid, group in enumerate(self.mtransforms):
11
12             group.update(self.transforms)
13
14             width = group.get('width', .8)
15
16             gwidth = width/ngroups
17
18             offset = gid/ngroups*width

```

```

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

```

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

```

1 transforms = {'position': {'name':'sex',
2                               'encoder': position.Nominal({'female':0, 'male':1})}}
3 group_transforms =  [{'length': {'name':'Adelie'},
4                               'facecolors': {'name':'Adelie_s', 'encoder':cmap}},
5                               {'length': {'name':'Chinstrap'},
6                               'facecolors': {'name':'Chinstrap_s', 'encoder':cmap}},
7                               {'length': {'name':'Gentoo'},
8                               'facecolors': {'name':'Gentoo_s', 'encoder':cmap}}]

```

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

```

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

```

653 but differ in assembly \hat{Q} . By decomposing the architecture into data, visual encoding,
 654 and assembly steps, we are able to build components that are more flexible and also more self
 655 contained than the existing code base. While very rough, this API demonstrates that the

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

664 5 Discussion

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

682 **5.1 Limitations**

683 The TEAM model is a specification visualization library developers can use to implement
684 structure preserving library components. Implementing a TEAM based architecture involves
685 developers explicitly describing the structure and continuity of the data and the structure
686 and continuity the artist expect. TEAM does not provide a framework for recommending
687 visualizations to the user, therefore effectiveness [12, 99] is out of scope. But, automatic
688 recommendation tools could be built using TEAM components.

689 While TEAM specifies the components, the developers building libraries using TEAM
690 components decide which compositions of components are semantically correct for the do-
691 main. For this reason, TEAM does not include data space transforms, as are incorporated
692 into libraries like Tableau or ggplot, instead leaving choice of computations to implemen-
693 tors of the data object. TEAM’s intentional ignorance of semantics also means it cannot
694 evaluate whether a figurative glyph [2] is a semantically correct choice, but it can enforce
695 equivariance constraints of glyphs generated from data components enforcing equivariance
696 of figurative glyphs [2] generated from data components [100, 101]. TEAM also allows
697 graphics to have a lower dimensional continuity than the source data when a retraction map
698 from one continuity to the other exists. For example, TEAM components could transform
699 1D continuous segments into 0D discrete elements, e.g. bar charts or scatter plots. As with
700 computations, it is the role of the domain specific library to determine which figurative
701 glyphs and continuity downgrades are appropriate.

702 The prototype is deeply tied to Matplotlib’s existing architecture, so it has not yet been
703 worked through how the model generalizes to libraries such as R graphics [102], VTK, and
704 D3. TEAM has only been tested using PNG files rendered with AGG [80], but is expected
705 to work with all the file types Matplotlib currently supports, including svg, pdf, and eps.
706 We have not yet addressed how this framework interfaces with high performance rendering
707 libraries such as openGL [78] that implement different models of ρ .

708 **5.2 Future Work**

709 More work is needed to formalize the composition operators, equivalence class A' , and the
710 mathematical model of interactivity. We also need to implement artists that demonstrate
711 that the model can underpin a minimally viable library, foremost an image [103, 104], a heat
712 map [105, 106], and an inherently computational artist such as a boxplot [37]. In summary,
713 the proposed scope of work is

| work period | milestones & tasks |
|---------------------|--|
| April - July 2021 | <p>prepare and submit conference presentation on new functionality enabled by model for <i>SciPy</i>:</p> <p>artists that do not inherit from existing Matplotlib artists, computational artists such as histograms, non tabular data, composite interactive artist</p> |
| June - Sept 2021 | <p>prepare and submit theory paper on interactivity to <i>TCVG or Eurovis 2022</i>:</p> <p>fully work out and describe math of addition operators and lookups from graphic to data space, implement brush linked artist (shared base space) and artist that exploits sheafs</p> |
| May - Nov 2021 | <p>prepare and submit applications paper on high dimensional to <i>TCVG</i>:</p> <p>math and implementation of computational artists, concurrent artists and data sources, non-trivial data bundles</p> |
| Aug 2021 - Feb 2022 | <p>prepare and submit systems paper on building domain specific libraries based on this model to <i>Infoviz 2022</i>:</p> <p>domain specific structured data, composite artists, inference of meta data components, mathematical notion of a visualization (labeled, multiple artists, etc)</p> |
| March 2022 | <p>dissertation writing:</p> <p>synthesize previous work on climate data, compile topological equivariant artist model work</p> |
| April 2022 | defense 59 |

Table 4

714 In acknowledgement that the schedule is optimistic, this work has various scales of data
715 applications. We plan to apply this model to datasets with complex continuities, such as
716 the trajectories of rats running around a maze and the positions of their limbs. We also
717 potentially can look at large scale biology or climate datasets. The data applications could
718 be further integrated with topological [107] and functional [108] data analysis methods.
719 Since this model formalizes notions of structure preservation, it can serve as a good base
720 for tools that assess quality metrics [109] or invariance [13] of visualizations with respect
721 to graphical encoding choices. This specification of structure could also be used to develop
722 a serialization structure that could then be used to allow Matplotlib to interface with other
723 visualization libraries such as open GL via shared serialization protocol. While this paper
724 formulates visualization in terms of monoidal action homomorphisms between fiber bundles,
725 the model lends itself to a categorical formulation [60, 110] that could be further explored.

726 6 Conclusion

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

738 References

- 739 [1] Michael Friendly. “A Brief History of Data Visualization”. en. In: *Handbook of Data*
740 *Visualization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 15–56. ISBN:
741 978-3-540-33036-3 978-3-540-33037-0. DOI: [10.1007/978-3-540-33037-0_2](https://doi.org/10.1007/978-3-540-33037-0_2).
- 742 [2] L. Byrne, D. Angus, and J. Wiles. “Acquired Codes of Meaning in Data Visualization
743 and Infographics: Beyond Perceptual Primitives”. In: *IEEE Transactions on Visual-
744 ization and Computer Graphics* 22.1 (Jan. 2016), pp. 509–518. ISSN: 1077-2626. DOI:
745 [10.1109/TVCG.2015.2467321](https://doi.org/10.1109/TVCG.2015.2467321).
- 746 [3] Krist Wongsuphasawat. *Navigating the Wide World of Data Visualization Libraries
747 (on the Web)*. 2021.
- 748 [4] J. Hughes. “Why Functional Programming Matters”. In: *The Computer Journal* 32.2
749 (Jan. 1989), pp. 98–107. ISSN: 0010-4620. DOI: [10.1093/comjnl/32.2.98](https://doi.org/10.1093/comjnl/32.2.98).
- 750 [5] Zhenjiang Hu, John Hughes, and Meng Wang. “How Functional Programming Mat-
751 tered”. In: *National Science Review* 2.3 (Sept. 2015), pp. 349–370. ISSN: 2095-5138.
752 DOI: [10.1093/nsr/nwv042](https://doi.org/10.1093/nsr/nwv042).
- 753 [6] J. D. Hunter. “Matplotlib: A 2D Graphics Environment”. In: *Computing in Science
754 Engineering* 9.3 (May 2007), pp. 90–95. ISSN: 1558-366X. DOI: [10.1109/MCSE.2007.55](https://doi.org/10.1109/MCSE.2007.55).
- 756 [7] John Hunter and Michael Droettboom. *The Architecture of Open Source Applications
757 (Volume 2): Matplotlib*. <https://www.aosabook.org/en/matplotlib.html>.
- 758 [8] A. Sarikaya et al. “What Do We Talk About When We Talk About Dashboards?”
759 In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (Jan. 2019),
760 pp. 682–692. ISSN: 1941-0506. DOI: [10.1109/TVCG.2018.2864903](https://doi.org/10.1109/TVCG.2018.2864903).
- 761 [9] Jacques Bertin. *Semiology of Graphics : Diagrams, Networks, Maps*. English. Red-
762 lands, Calif.: ESRI Press, 2011. ISBN: 978-1-58948-261-6 1-58948-261-1.
- 763 [10] Wilson Stothers. *The Klein View of Geometry: Similarity Group*. <https://www.maths.gla.ac.uk/wwws/cabripages/>

- 764 [11] Jock Mackinlay. “Automating the Design of Graphical Presentations of Relational
765 Information”. In: *ACM Transactions on Graphics* 5.2 (Apr. 1986), pp. 110–141. ISSN:
766 0730-0301. DOI: [10.1145/22949.22950](https://doi.org/10.1145/22949.22950).
- 767 [12] Jock Mackinlay. “Automatic Design of Graphical Presentations”. English. PhD The-
768 sis. Stanford, 1987.
- 769 [13] G. Kindlmann and C. Scheidegger. “An Algebraic Process for Visualization Design”.
770 In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (Dec. 2014),
771 pp. 2181–2190. ISSN: 1941-0506. DOI: [10.1109/TVCG.2014.2346325](https://doi.org/10.1109/TVCG.2014.2346325).
- 772 [14] Ricky Shadrach. *Introduction to Groups*. <https://www.mathsisfun.com/sets/groups-introduction.html>. 2017.
- 773 [15] “Naturalness Principle - InfoVis:Wiki”. In: *InfoVis:Wiki* ().
- 774 [16] Donald A. Norman. *Things That Make Us Smart: Defending Human Attributes in
775 the Age of the Machine*. USA: Addison-Wesley Longman Publishing Co., Inc., 1993.
776 ISBN: 0-201-62695-0.
- 777 [17] Edward R. Tufte. *The Visual Display of Quantitative Information*. English. Cheshire,
778 Conn.: Graphics Press, 2001. ISBN: 0-9613921-4-2 978-0-9613921-4-7 978-1-930824-13-
779 3 1-930824-13-0.
- 780 [18] J. Heer and M. Agrawala. “Software Design Patterns for Information Visualization”.
781 In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (2006), pp. 853–
782 860. DOI: [10.1109/TVCG.2006.178](https://doi.org/10.1109/TVCG.2006.178).
- 783 [19] E. H. Chi. “A Taxonomy of Visualization Techniques Using the Data State Reference
784 Model”. In: *IEEE Symposium on Information Visualization 2000. INFOVIS 2000.*
785 *Proceedings*. Oct. 2000, pp. 69–75. DOI: [10.1109/INFVIS.2000.885092](https://doi.org/10.1109/INFVIS.2000.885092).
- 786 [20] Jeffrey Heer and Michael Bostock. “Declarative Language Design for Interactive Vi-
787 sualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.6
788 (Nov. 2010), pp. 1149–1156. ISSN: 1077-2626. DOI: [10.1109/TVCG.2010.144](https://doi.org/10.1109/TVCG.2010.144).

- 790 [21] C. Stolte, D. Tang, and P. Hanrahan. “Polaris: A System for Query, Analysis, and
791 Visualization of Multidimensional Relational Databases”. In: *IEEE Transactions on*
792 *Visualization and Computer Graphics* 8.1 (Jan. 2002), pp. 52–65. ISSN: 1941-0506.
793 DOI: [10.1109/2945.981851](https://doi.org/10.1109/2945.981851).
- 794 [22] Pat Hanrahan. “VizQL: A Language for Query, Analysis and Visualization”. In:
795 *Proceedings of the 2006 ACM SIGMOD International Conference on Management*
796 *of Data*. SIGMOD ’06. New York, NY, USA: Association for Computing Machinery,
797 2006, p. 721. ISBN: 1-59593-434-0. DOI: [10.1145/1142473.1142560](https://doi.org/10.1145/1142473.1142560).
- 798 [23] J. Mackinlay, P. Hanrahan, and C. Stolte. “Show Me: Automatic Presentation for
799 Visual Analysis”. In: *IEEE Transactions on Visualization and Computer Graphics*
800 13.6 (Nov. 2007), pp. 1137–1144. ISSN: 1941-0506. DOI: [10.1109/TVCG.2007.70594](https://doi.org/10.1109/TVCG.2007.70594).
- 801 [24] Leland Wilkinson. *The Grammar of Graphics*. en. 2nd ed. Statistics and Computing.
802 New York: Springer-Verlag New York, Inc., 2005. ISBN: 978-0-387-24544-7.
- 803 [25] Hadley Wickham. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New
804 York, 2016. ISBN: 978-3-319-24277-4.
- 805 [26] M. Bostock and J. Heer. “Protovis: A Graphical Toolkit for Visualization”. In: *IEEE*
806 *Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1121–
807 1128. ISSN: 1941-0506. DOI: [10.1109/TVCG.2009.174](https://doi.org/10.1109/TVCG.2009.174).
- 808 [27] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. “Declarative Inter-
809 action Design for Data Visualization”. en. In: *Proceedings of the 27th Annual ACM*
810 *Symposium on User Interface Software and Technology*. Honolulu Hawaii USA: ACM,
811 Oct. 2014, pp. 669–678. ISBN: 978-1-4503-3069-5. DOI: [10.1145/2642918.2647360](https://doi.org/10.1145/2642918.2647360).
- 812 [28] Jacob VanderPlas et al. “Altair: Interactive Statistical Visualizations for Python”.
813 en. In: *Journal of Open Source Software* 3.32 (Dec. 2018), p. 1057. ISSN: 2475-9066.
814 DOI: [10.21105/joss.01057](https://doi.org/10.21105/joss.01057).
- 815 [29] Jeffrey D. Ullman and Jennifer. Widom. *A First Course in Database Systems*. En-
816 glish. Upper Saddle River, NJ: Pearson Prentice Hall, 2008. ISBN: 0-13-600637-X
817 978-0-13-600637-4.

- 818 [30] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. “NIH Image to
819 ImageJ: 25 Years of Image Analysis”. In: *Nature Methods* 9.7 (July 2012), pp. 671–
820 675. ISSN: 1548-7105. DOI: [10.1038/nmeth.2089](https://doi.org/10.1038/nmeth.2089).
- 821 [31] Nicholas Sofroniew et al. *Napari/Napari: 0.4.5rc1*. Zenodo. Feb. 2021. DOI: [10.5281/zenodo.4533308](https://doi.org/10.5281/zenodo.4533308).
- 822 [32] Software Studies. *Culturevis/Imageplot*. Jan. 2021.
- 823 [33] *Writing Plugins*. en. <https://imagej.net/Writing-plugins>.
- 824 [34] Mathieu Bastian, Sébastien Heymann, and Mathieu Jacomy. “Gephi: An Open
825 Source Software for Exploring and Manipulating Networks”. en. In: *Proceedings of
826 the International AAAI Conference on Web and Social Media* 3.1 (Mar. 2009). ISSN:
827 2334-0770.
- 828 [35] John Ellson et al. “Graphviz—Open Source Graph Drawing Tools”. In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Berlin, Heidelberg:
829 Springer Berlin Heidelberg, 2002, pp. 483–484. ISBN: 978-3-540-45848-7.
- 830 [36] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. “Exploring Network Struc-
831 ture, Dynamics, and Function Using NetworkX”. In: *Proceedings of the 7th Python
832 in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman.
833 Pasadena, CA USA, 2008, pp. 11–15.
- 834 [37] Hadley Wickham and Lisa Stryjewski. “40 Years of Boxplots”. In: *The American
835 Statistician* (2011).
- 836 [38] *Data Representation in Mayavi*. <https://docs.enthought.com/mayavi/mayavi/data.html>.
- 837 [39] M. Tory and T. Moller. “Rethinking Visualization: A High-Level Taxonomy”. In:
838 *IEEE Symposium on Information Visualization*. 2004, pp. 151–158. DOI: [10.1109/INFVIS.2004.59](https://doi.org/10.1109/INFVIS.2004.59).
- 839 [40] M. Bostock, V. Ogievetsky, and J. Heer. “D³ Data-Driven Documents”. In: *IEEE
840 Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pp. 2301–
841 2309. ISSN: 1941-0506. DOI: [10.1109/TVCG.2011.185](https://doi.org/10.1109/TVCG.2011.185).

- 845 [41] Marcus D. Hanwell et al. “The Visualization Toolkit (VTK): Rewriting the Rendering
846 Code for Modern Graphics Cards”. en. In: *SoftwareX* 1-2 (Sept. 2015), pp. 9–12. ISSN:
847 23527110. DOI: [10.1016/j.softx.2015.04.001](https://doi.org/10.1016/j.softx.2015.04.001).
- 848 [42] Berk Geveci et al. “VTK”. In: *The Architecture of Open Source Applications* 1 (2012),
849 pp. 387–402.
- 850 [43] P. Ramachandran and G. Varoquaux. “Mayavi: 3D Visualization of Scientific Data”.
851 In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 40–51. ISSN: 1558-366X.
852 DOI: [10.1109/MCSE.2011.35](https://doi.org/10.1109/MCSE.2011.35).
- 853 [44] Michael Waskom and the seaborn development team. *Mwaskom/Seaborn*. Zenodo.
854 Sept. 2020. DOI: [10.5281/zenodo.592845](https://doi.org/10.5281/zenodo.592845).
- 855 [45] Brian Wylie and Jeffrey Baumes. “A Unified Toolkit for Information and Scientific
856 Visualization”. In: *Proc.SPIE*. Vol. 7243. Jan. 2009. DOI: [10.1117/12.805589](https://doi.org/10.1117/12.805589).
- 857 [46] Stephan Hoyer and Joe Hamman. “Xarray: ND Labeled Arrays and Datasets in
858 Python”. In: *Journal of Open Research Software* 5.1 (2017).
- 859 [47] James Ahrens, Berk Geveci, and Charles Law. “Paraview: An End-User Tool for
860 Large Data Visualization”. In: *The visualization handbook* 717.8 (2005).
- 861 [48] D. M. Butler and M. H. Pendley. “A Visualization Model Based on the Mathematics
862 of Fiber Bundles”. en. In: *Computers in Physics* 3.5 (1989), p. 45. ISSN: 08941866.
863 DOI: [10.1063/1.168345](https://doi.org/10.1063/1.168345).
- 864 [49] David M. Butler and Steve Bryson. “Vector-Bundle Classes Form Powerful Tool
865 for Scientific Visualization”. en. In: *Computers in Physics* 6.6 (1992), p. 576. ISSN:
866 08941866. DOI: [10.1063/1.4823118](https://doi.org/10.1063/1.4823118).
- 867 [50] David I Spivak. *Databases Are Categories*. en. Slides. June 2010.
- 868 [51] David I Spivak. “SIMPLICIAL DATABASES”. en. In: (), p. 35.
- 869 [52] Tamara Munzner. *Visualization Analysis and Design*. AK Peters Visualization Series.
870 CRC press, Oct. 2014. ISBN: 978-1-4665-0891-0.

- 871 [53] Tamara Munzner. “Ch 2: Data Abstraction”. In: *CPSC547: Information Visualiza-*
872 *tion, Fall 2015-2016* ().
- 873 [54] E.H. Spanier. *Algebraic Topology*. McGraw-Hill Series in Higher Mathematics.
874 Springer, 1989. ISBN: 978-0-387-94426-5.
- 875 [55] “Locally Trivial Fibre Bundle”. In: *Encyclopedia of Mathematics* ().
- 876 [56] Brent A Yorgey. “Monoids: Theme and Variations (Functional Pearl)”. en. In: (),
877 p. 12.
- 878 [57] “Monoid”. en. In: *Wikipedia* (Jan. 2021).
- 879 [58] “Semigroup Action”. en. In: *Wikipedia* (Jan. 2021).
- 880 [59] nLab authors. “Action”. In: (Mar. 2021).
- 881 [60] Brendan Fong and David I. Spivak. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. en. First. Cambridge University Press, July
882 2019. ISBN: 978-1-108-66880-4 978-1-108-48229-5 978-1-108-71182-1. DOI: [10.1017/9781108668804](https://doi.org/10.1017/9781108668804).
- 885 [61] S. S. Stevens. “On the Theory of Scales of Measurement”. In: *Science* 103.2684 (1946),
886 pp. 677–680. ISSN: 00368075, 10959203.
- 887 [62] W A Lea. “A Formalization of Measurement Scale Forms”. en. In: (), p. 44.
- 888 [63] “Quotient Space (Topology)”. en. In: *Wikipedia* (Nov. 2020).
- 889 [64] Professor Denis Auroux. “Math 131: Introduction to Topology”. en. In: (), p. 113.
- 890 [65] Robert W. Ghrist. *Elementary Applied Topology*. Vol. 1. Createspace Seattle, 2014.
- 891 [66] Robert Ghrist. “Homological Algebra and Data”. In: *Math. Data* 25 (2018), p. 273.
- 892 [67] David Urbanik. “A Brief Introduction to Schemes and Sheaves”. en. In: (), p. 16.
- 893 [68] Dmitry Nekrasovski et al. “An Evaluation of Pan & Zoom and Rubber Sheet
894 Navigation with and without an Overview”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’06. New York, NY, USA:
895 Association for Computing Machinery, 2006, pp. 11–20. ISBN: 1-59593-372-7. DOI:
896 [10.1145/1124772.1124775](https://doi.org/10.1145/1124772.1124775).

- 898 [69] Michael S. Crouch, Andrew McGregor, and Daniel Stubbs. “Dynamic Graphs in the
899 Sliding-Window Model”. In: *European Symposium on Algorithms*. Springer, 2013,
900 pp. 337–348.
- 901 [70] Chia-Shang James Chu. “Time Series Segmentation: A Sliding Window Approach”.
902 In: *Information Sciences* 85.1 (July 1995), pp. 147–173. ISSN: 0020-0255. DOI: [10.1016/0020-0255\(95\)00021-G](#).
- 903 [71] Charles R Harris et al. “Array Programming with NumPy”. In: *Nature* 585.7825
904 (2020), pp. 357–362.
- 905 [72] Jeff Reback et al. *Pandas-Dev/Pandas: Pandas 1.0.3*. Zenodo. Mar. 2020. DOI: [10.5281/zenodo.3715232](#).
- 906 [73] Matthew Rocklin. “Dask: Parallel Computation with Blocked Algorithms and Task
907 Scheduling”. In: *Proceedings of the 14th Python in Science Conference*. Vol. 126.
908 Citeseer, 2015.
- 909 [74] “Retraction (Topology)”. en. In: *Wikipedia* (July 2020).
- 910 [75] Eric W. Weisstein. *Homotopy*. en. <https://mathworld.wolfram.com/Homotopy.html>.
911 Text.
- 912 [76] Tim Bierenz, Richard Cohn, and Calif.) Adobe Systems (Mountain View. *Portable
913 Document Format Reference Manual*. Citeseer, 1993.
- 914 [77] A. Quint. “Scalable Vector Graphics”. In: *IEEE MultiMedia* 10.3 (July 2003), pp. 99–
915 102. ISSN: 1941-0166. DOI: [10.1109/MMUL.2003.1218261](#).
- 916 [78] George S. Carson. “Standards Pipeline: The OpenGL Specification”. In: *SIGGRAPH
917 Comput. Graph.* 31.2 (May 1997), pp. 17–18. ISSN: 0097-8930. DOI: [10.1145/271283.271292](#).
- 918 [79] *Cairographics.Org*.
- 919 [80] Maxim Shemanarev. *Anti-Grain Geometry*.

- 923 [81] S. K. Card and J. Mackinlay. “The Structure of the Information Visualization Design
924 Space”. In: *Proceedings of VIZ '97: Visualization Conference, Information Visuali-*
925 *zation Symposium and Parallel Rendering Symposium*. Oct. 1997, pp. 92–99. DOI:
926 [10.1109/INFVIS.1997.636792](https://doi.org/10.1109/INFVIS.1997.636792).
- 927 [82] *Measurement Scales and Statistics: Resurgence of an Old Misconception.* - PsycNET.
928 <https://psycnet.apa.org/doiLanding?doi=10.1037%2F0033-2909.87.3.564>.
- 929 [83] Christian Remling. *Algebra (Math 5353/5363) Lecture Notes*. Lecture Notes. Uni-
930 versity of Oklahoma.
- 931 [84] H. M. Johnson. “Pseudo-Mathematics in the Mental and Social Sciences”. In: *The*
932 *American Journal of Psychology* 48.2 (1936), pp. 342–351. ISSN: 00029556. DOI: [10.2307/1415754](https://doi.org/10.2307/1415754).
- 933 [85] M. A. Thomas. *Mathematization, Not Measurement: A Critique of Stevens' Scales of*
934 *Measurement*. en. SSRN Scholarly Paper ID 2412765. Rochester, NY: Social Science
935 Research Network, Oct. 2014. DOI: [10.2139/ssrn.2412765](https://doi.org/10.2139/ssrn.2412765).
- 936 [86] “Connected Space”. en. In: *Wikipedia* (Dec. 2020).
- 937 [87] Caroline Ziemkiewicz and Robert Kosara. “Embedding Information Visualization
938 within Visual Representation”. In: *Advances in Information and Intelligent Systems*.
939 Ed. by Zbigniew W. Ras and William Ribarsky. Berlin, Heidelberg: Springer Berlin
940 Heidelberg, 2009, pp. 307–326. ISBN: 978-3-642-04141-9. DOI: [10.1007/978-3-642-04141-9_15](https://doi.org/10.1007/978-3-642-04141-9_15).
- 941 [88] Sheelagh Carpendale. *Visual Representation from Semiology of Graphics by J. Bertin*.
942 en.
- 943 [89] “Jet Bundle”. en. In: *Wikipedia* (Dec. 2020).
- 944 [90] Jana Musilová and Stanislav Hronek. “The Calculus of Variations on Jet Bundles
945 as a Universal Approach for a Variational Formulation of Fundamental Physical
946 Theories”. In: *Communications in Mathematics* 24.2 (Dec. 2016), pp. 173–193. ISSN:
947 2336-1298. DOI: [10.1515/cm-2016-0012](https://doi.org/10.1515/cm-2016-0012).

- 950 [91] Yael Albo et al. “Off the Radar: Comparative Evaluation of Radial Visualization
951 Solutions for Composite Indicators”. In: *IEEE Transactions on Visualization and*
952 *Computer Graphics* 22.1 (Jan. 2016), pp. 569–578. ISSN: 1077-2626. DOI: [10.1109/TVCG.2015.2467322](https://doi.org/10.1109/TVCG.2015.2467322).
- 953
- 954 [92] Z. Qu and J. Hullman. “Keeping Multiple Views Consistent: Constraints, Validations,
955 and Exceptions in Visualization Authoring”. In: *IEEE Transactions on Visualization*
956 *and Computer Graphics* 24.1 (Jan. 2018), pp. 468–477. ISSN: 1941-0506. DOI: [10.1109/TVCG.2017.2744198](https://doi.org/10.1109/TVCG.2017.2744198).
- 957
- 958 [93] Richard A. Becker and William S. Cleveland. “Brushing Scatterplots”. In: *Techno-*
959 *metrics* 29.2 (May 1987), pp. 127–142. ISSN: 0040-1706. DOI: [10.1080/00401706.1987.10488204](https://doi.org/10.1080/00401706.1987.10488204).
- 960
- 961 [94] Andreas Buja et al. “Interactive Data Visualization Using Focusing and Linking”. In:
962 *Proceedings of the 2nd Conference on Visualization ’91. VIS ’91.* Washington, DC,
963 USA: IEEE Computer Society Press, 1991, pp. 156–163. ISBN: 0-8186-2245-8.
- 964 [95] *Data Classes*. <https://docs.python.org/3/library/dataclasses.html>.
- 965 [96] Kristen B. Gorman, Tony D. Williams, and William R. Fraser. “Ecological Sexual
966 Dimorphism and Environmental Variability within a Community of Antarctic Pen-
967 guins (Genus Pygoscelis)”. In: *PLOS ONE* 9.3 (Mar. 2014), e90081. DOI: [10.1371/journal.pone.0090081](https://doi.org/10.1371/journal.pone.0090081).
- 968
- 969 [97] Allison Marie Horst, Alison Presmanes Hill, and Kristen B Gorman. *Palmerpen-*
970 *guins: Palmer Archipelago (Antarctica) Penguin Data*. Manual. 2020. DOI: [10.5281/zenodo.3960218](https://doi.org/10.5281/zenodo.3960218).
- 971
- 972 [98] Muhammad Chenariyan Nakhaee. *Mcnakhaee/Palmerpenguins*. Jan. 2021.
- 973 [99] John M Chambers et al. *Graphical Methods for Data Analysis*. Vol. 5. Wadsworth
974 Belmont, CA, 1983.
- 975 [100] F. Beck. “Software Feathers Figurative Visualization of Software Metrics”. In: *2014*
976 *International Conference on Information Visualization Theory and Applications*
977 (*IVAPP*). Jan. 2014, pp. 5–16.

- 978 [101] Lydia Byrne, Daniel Angus, and Janet Wiles. “Figurative Frames: A Critical Vocab-
979 uary for Images in Information Visualization”. In: *Information Visualization* 18.1
980 (Aug. 2017), pp. 45–67. ISSN: 1473-8716. DOI: [10.1177/1473871617724212](https://doi.org/10.1177/1473871617724212).
- 981 [102] Paul Murrell. *R Graphics, Third Edition*. 3rd. Chapman & Hall/CRC, 2018.
982 ISBN: 1-4987-8905-6.
- 983 [103] Robert B Haber and David A McNabb. “Visualization Idioms: A Conceptual Model
984 for Scientific Visualization Systems”. In: *Visualization in scientific computing* 74
985 (1990), p. 93.
- 986 [104] Charles D Hansen and Chris R Johnson. *Visualization Handbook*. Elsevier, 2011.
- 987 [105] Leland Wilkinson and Michael Friendly. “The History of the Cluster Heat Map”.
988 In: *The American Statistician* 63.2 (May 2009), pp. 179–184. ISSN: 0003-1305. DOI:
989 [10.1198/tas.2009.0033](https://doi.org/10.1198/tas.2009.0033).
- 990 [106] Toussaint Loua. *Atlas Statistique de La Population de Paris*. J. Dejey & cie, 1873.
- 991 [107] C. Heine et al. “A Survey of Topology-Based Methods in Visualization”. In: *Computer
992 Graphics Forum* 35.3 (June 2016), pp. 643–667. ISSN: 0167-7055. DOI: [10.1111/cgf.12933](https://doi.org/10.1111/cgf.12933).
- 993 [108] James O Ramsay. *Functional Data Analysis*. Wiley Online Library, 2006.
- 995 [109] Enrico Bertini, Andrada Tatu, and Daniel Keim. “Quality Metrics in High-
996 Dimensional Data Visualization: An Overview and Systematization”. In: *IEEE
997 Transactions on Visualization and Computer Graphics* 17.12 (2011), pp. 2203–2212.
- 998 [110] Bartosz Milewski. “Category Theory for Programmers”. en. In: (), p. 498.