

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

# Abstract

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

# Contents

36	<b>Abstract</b>	<b>ii</b>
37	<b>1 Introduction</b>	<b>1</b>
38	<b>2 Background</b>	<b>2</b>
39	2.1 Structure: <i>equivariance &amp; continuity</i> . . . . .	2
40	2.2 Tools . . . . .	5
41	2.3 Data . . . . .	8
42	2.4 Contribution . . . . .	9
43	<b>3 Topological Artist Model</b>	<b>10</b>
44	3.1 Data Space $E$ . . . . .	11
45	3.1.1 Variables in Fiber Space $F$ . . . . .	12
46	3.1.2 Measurement Scales: Monoid Actions . . . . .	14
47	3.1.3 Continuity of the Data $K$ . . . . .	15
48	3.1.4 Data $\tau$ . . . . .	18
49	3.1.5 Applications to Data Containers . . . . .	19
50	3.2 Graphic Space $H$ . . . . .	20
51	3.2.1 Idealized Display $D$ . . . . .	20
52	3.2.2 Continuity of the Graphic $S$ . . . . .	21
53	3.2.3 Graphic $\rho$ . . . . .	23
54	3.3 Artist . . . . .	24
55	3.3.1 Visual Fiber Bundle $V$ . . . . .	25
56	3.3.2 Visual Encoders $\nu$ . . . . .	27
57	3.3.3 Graphic Assembler $Q$ . . . . .	29
58	3.3.4 Assembly $Q$ . . . . .	31
59	3.3.5 Assembly factory $\hat{Q}$ . . . . .	34
60	3.3.6 Sheafs . . . . .	36
61	3.3.7 Composition of Artists: $+$ . . . . .	36

62	3.3.8 Equivalence class of artists $A'$ . . . . .	37
63	<b>4 Prototype Implementation: Matplottoy</b>	<b>38</b>
64	4.1 Artist Class $A'$ . . . . .	39
65	4.2 Encoders $\nu$ . . . . .	45
66	4.3 Data $E$ . . . . .	47
67	4.4 Case Study: Penguins . . . . .	52
68	<b>5 Discussion</b>	<b>56</b>
69	5.1 Limitations . . . . .	57
70	5.2 Future Work . . . . .	58
71	<b>6 Conclusion</b>	<b>59</b>

# 1 Introduction

Building block level visualization libraries implement the functions that transform data into some component of the visual representation, providing a collection of components and utilities that can be combined to create a visualization [1]. Often domain specific tools are built out of these building block libraries, meaning the building block libraries must provide components general enough to satisfy a wide variety of data and visual representation needs. Specifically they must satisfy confirmatory and exploratory visualization needs [2], in scientific and information (formal and informal[3]) visualization domains. To satisfy ever developing visualization needs, new components can be incorporated into the library in sometimes ad-hoc ways that can lead to API incoherency and inconsistent behavior. We propose that like physical building blocks, building block libraries should provide a collection of well defined pieces that can be composed in whichever ways the blocks fit together. **figure of what we mean by building block?** We specify that for a valid visualization block is a structure preserving transformation from data to visual space, and we define structure in terms of *continuity* and *equivariance*. We then use this model to develop a design specification for the components of a building block visualization library. The notion of self contained, inherently modular, building blocks lends itself naturally to a functional paradigm of visualization [4]. We adopt a functional model for a redesign because the lack of side effects means functional architecture can be evaluated for correctness, functional programs tend to be shorter and clearer, and are well suited to distributed, concurrent, and on demand tasks[5].

This work is strongly motivated by the needs of the Matplotlib[6, 7] visualization library. One of the most widely used visualization libraries in Python, since 2002 new components and features have been added in a some what adhoc, sometimes hard to maintain, manner. Particularly, each new component carries its own implicit notion of how it believes the data is structured-for example if the data is a table, cube, image, or network - that is then expressed in the API for that component. In turn, this yields an inconsistent API for interfacing with the data, for example when updating streaming visualizations or constructing dashboards[8].

100 This entangling of data model with visual transform also yields inconsistencies in how visual  
101 component transforms, e.g. shape or color, are supported. We propose that these issues can  
102 be ameliorated via a redesign of the functions that convert data to graphics, named *Artists* in  
103 Matplotlib, in a manner that reliably enforces *continuity* and *equivariance* constraints. We  
104 evaluate our functional model by implementing new artists in Matplotlib that are specified  
105 via *equivariance* and *continuity* constraints. We then use the common data model introduced  
106 by the model to demonstrate how plotting functions can be consolidated in a way that makes  
107 clear whether the difference is in expected data structure, visual component encoding, or  
108 the resulting graphic.

## 109 2 Background

110 The notion that visualization is structure preserving maps from data to visual representation  
111 is neither a new formalism nor a new implementation goal; this work bridges the formalism  
112 and implementation in a functional manner with a topological approach at a building blocks  
113 library level to propose a new model that can be used to guide library development.

### 114 2.1 Structure: *equivariance & continuity*

115 Visual representations of data, by definition, reflect something of the underlying structure  
116 and semantics[9], whether through direct mappings from data into visual elements or via  
117 figurative representations that have meaning due to their similarity in shape to external  
118 concepts [10]. The components of a visual representation were first codified by Bertin[11],  
119 who introduced a notion of structure preservation that we formally describe in terms of  
120 *equivariance* and *continuity*.

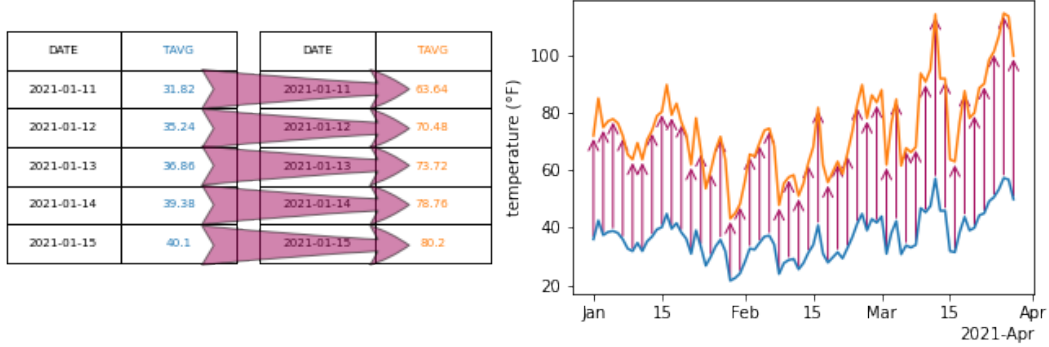


Figure 1: The data in blue is scaled by a factor of two, yielding the data in orange. To preserve *equivariance*, the blue line plot representation of the unscaled data is also scaled by a factor of two, yielding the orange line plot that is equivalent to the scaled data.

Bertin proposes that there are classes of visual encodings—such as position, shape, color, and texture—that when mapped to from specific types of measurement, quantitative or qualitative, will preserve the properties of that measurement type. For example, in Figure 1, the data and visual representation are scaled by equivalent factors of two, resulting in the change illustrated in the shift from blue to orange data and lines. The idea of equivariance is formally defined as the mapping of a binary operator from the data domain to the visual domain in Mackinlay’s *A Presentation Tool* (APT) model [12, 13]. The algebraic model of visualization proposed by Kindlmann and Scheidegger uses equivariance to refer generally to invertible binary transformations [14], which are mathematical groups [15]. 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.

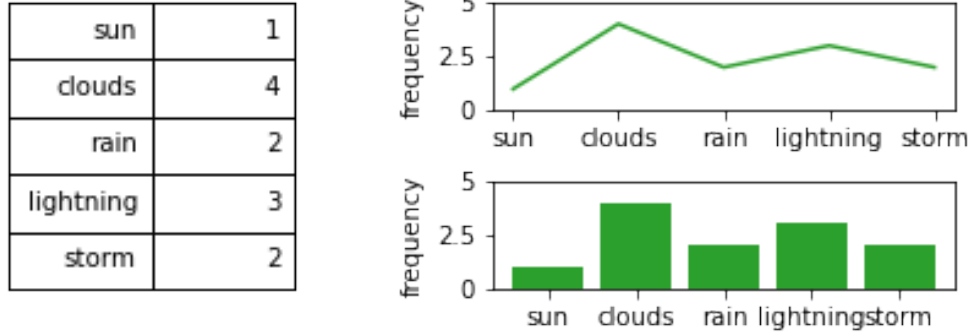


Figure 2: The line plot does not preserve *continuity* because it implies that the discrete records are connected to each other, while the bar plot is *continuity* preserving because it visually represents the records as independent data points.

Bertin proposes that the visual encodings be composited into graphical marks that match the *continuity* of the data - for example discrete data is a point, 1D continuous is the line, and 2D data is the area mark. In Figure 2, the line plot does not preserve continuity because the line connecting the discrete categories implies that the frequency of weather events is sampled from a continuous interval and the categories are points on that interval. **transition statement**

#### Structure

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

The measure of how much of the structure of the data the graphic encodes is a concept Mackinlay termed expressiveness, while the graphic’s effectiveness describes how much design choices are made in deference to perceptual saliency [16–19]. When the properties of the representation match the properties of the data, then the visualization is easier to understand according to Norman’s Naturalness Principal[20]. These ideas are combined into Tufte’s notion of graphical integrity, which is that a visual representation of quantitative



146 data must be directly proportional to the numerical quantities it represents (Lie Principal),  
 147 must have the same number of visual dimensions as the data, and should be well labeled  
 148 and contextualized, and not have any extraneous visual elements [21].

## 149 2.2 Tools

150 One of the reasons we developed a new formalism rather than adopting the architecture of  
 151 an existing library is that most information visualization software design patterns, as cate-  
 152 gorized by Heer and Agrawala[22], are tuned to very specific data structures. This in turn re-  
 153 stricts the design space of visual algorithms that display information (the visualization types  
 154 the library supports) since the algorithms are designed such that the structure of data is as-  
 155 sumed, as described in Tory and Möller’s taxonomy [**ToryRethinkingVisualization2004**].  
 156 In proposing a new architecture, we contrast the trade offs libraries make, describe different  
 157 types of data continuity, and discuss metrics by which a visualization library is traditionally  
 158 evaluated.

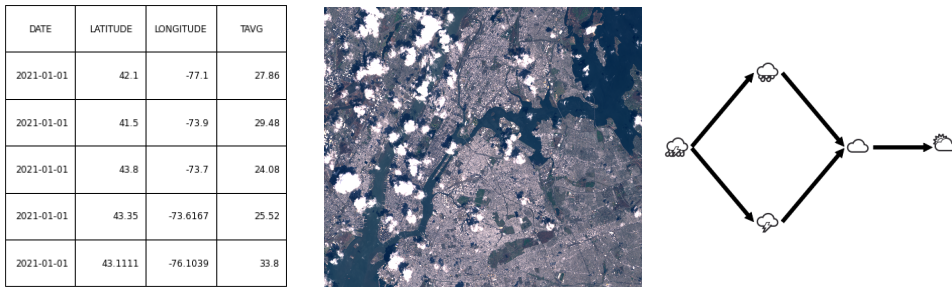


Figure 3: Visualization libraries, especially ones tied to specific domains, tend to be architected around a core data structure, such as tables, images, or networks.

159 One extensive family of relational table based libraries are those based on Wilkinson’s  
 160 Grammar of Graphics (GoG) [23], including ggplot[24], protovis[25] and D3 [26], vega[27]  
 161 and altair[28]. The restriction to tables in turn restricts the native design space to visu-  
 162 alizations suited to tables. Since the data space and graphic space is very well defined in  
 163 this grammar, it lends itself to a declarative interface [29]. This grammar oriented approach  
 164 allows users to describe how to compose visual elements into a graphical design [1], while

165 we are proposing a framework for building those elements. An example of this distinction  
166 is that the GoG grammar includes computation and aggregation of the table as part of the  
167 grammar, while we propose that most computations are specific to domains and only try to  
168 describe them when they are specifically part of the visual encoding - for example mapping  
169 data to a color. Disentangling the computation from the visual transforms allows us to  
170 determine whether the visualization library needs to handle them or if they can be more  
171 efficiently computed by the data container.

172 A different class of user facing tools are those that support images, such as ImageJ[30]  
173 or Napari[31]. These tools often have some support for visualizing non image components  
174 of a complex data set, but mostly in service to the image being visualized. These tools  
175 are ill suited for general purpose libraries that need to support data other than images  
176 because the architecture is oriented towards building plugins into the existing system [32]  
177 where the image is the core data structure. Even the digital humanities oriented ImageJ  
178 macro ImagePlot[33], which supports some non-image aggregate reporting charts, is still  
179 built around image data as the primary input.

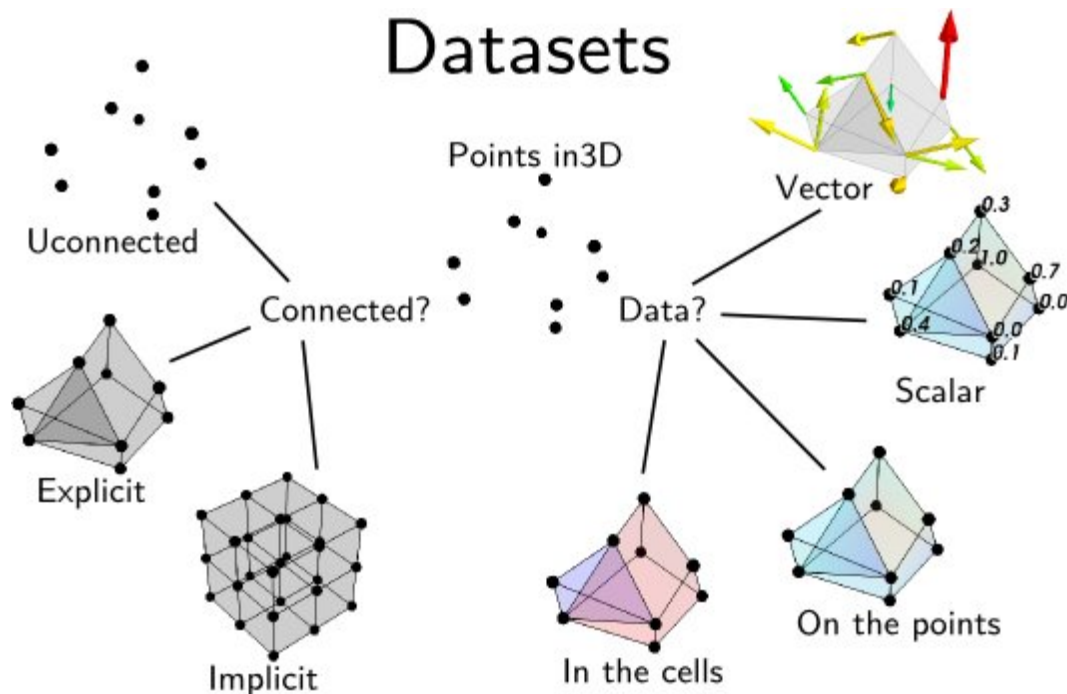


Figure 4: One way to describe data is by the connectivity of the points in the dataset. A database for example is often discrete unconnected points, while an image is an implicitly connected 2D grid. This image is from the Data Representation chapter of the MayaVi 4.7.2 documentation.[\[34\]](#)

There are also visualization tools where there is no single core structure, and instead internally carry around many different representations of data. Matplotlib, has this structure, as does VTK [\[35, 36\]](#) and its derivatives such as MayaVi[\[37\]](#) and extensions such as ParaView[\[38\]](#) and the infoviz themed Titan[\[39\]](#). Where GoG and ImageJ type libraries have very consistent APIs for their visualization tools because the data structure is the same, the APIs for visualizations in VTK and Matplotlib are significantly dependent on the structure of the data it expects. VTK has explicitly codified this in terms of continuity based data representations, as illustrated in figure 4. This in turn means that every new type of visualization must carry implicit assumptions about data structure in how it interfaces with the input data. This has lead to poor API consistency and brittle code as every visualization type has a very different point of view on how the data is structured. This API choice particularly breaks down when the same dataset is fed into visualizations

with different assumptions about structure or into a dashboard consisting of different types of visualization[8, 40] because there is no consistent way to update the data and therefore no consistent way of guaranteeing that the views stay in sync. Our model is a structure dependent formalism, but then also provides a core representation of that structure that is abstract enough to provide a common interface for many different types of visualization.

## 2.3 Data

Discrete and continuous data and their attributes form a discipline independent design space [3], so one of the drivers of this work was to facilitate building libraries that could natively support domain specific data containers that do not make assumptions about data continuity. As shown in figure 4, there are many types of connectivity. A database typically consists of unconnected records, while an image is an implicit 2D grid and a network is some sort of explicitly connected graph. These data structures typically contain not only the measurements or values of the data, but also domain specific semantic information such as that the data is a map or an image that a modern visualization library could exploit if this information was exposed to the API.

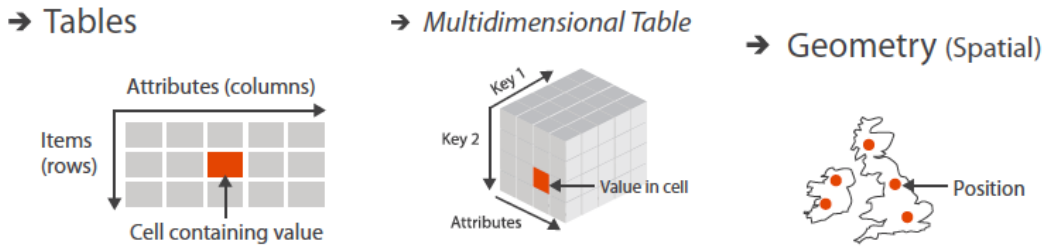


Figure 5: Image is figure 2.8 in Munzner’s Visualization Analysis and Design[19]

As shown in figure 4, there are many distinct ways of encoding each specific type of structure, while as mentioned in section 2.2 APIs are clearer when structured around a common data representation. Fiber bundles were proposed by Butler as one such representation because they encode the continuity of the data separately from the types of variables and are flexible enough to support discrete and ND continuous datasets [41, 42]. Since Butler’s model lacks a robust way of describing variables, we fold in Spivak’s Simplicial

213 formulation of databases [43, 44] so that we can encode a schema like description of the  
 214 data in the fiber bundle. In this work we will refer to the points of the dataset as *records*  
 215 to indicate that a point can be a vector of heterogenous elements. Each *component* of the  
 216 record is a single object, such as a temperature measurement, a color value, or an image.  
 217 We also generalize *component* to mean all objects in the dataset of a given type, such as  
 218 all temperatures or colors or images. The way in which these records are connected is the  
 219 *connectivity*, *continuity*, or more generally *topology*.

#### definitions

**records** points, observations, entries

**components** variables, attributes, fields

**connectivity** how the records are connected to each other

220 Often this topology has metadata associated with it, describing for example dependent  
 221 variables and the independent variables they are dependent on, or information about the  
 222 structure of the data such as when and where the measurement was taken. Building on  
 223 the idea of metadata as *keys* and their associated *values* proposed by Munzner [45], we  
 224 propose that information rich metadata are part of the components and instead the values  
 225 are keyed on coordinate free structural ids. In contrast to Munzner’s model where the  
 226 semantic meaning of the key is tightly coupled to the position of the value in the dataset,  
 227 our model considers keys to be a pure reference to topology. This allows the metadata  
 228 to be altered, for example by changing the coordinate systems or time resolution, without  
 229 imposing new semantics on the underlying structure.

## 2.4 Contribution

231 This work presents a mathematical model of the transformation from data to graphic rep-  
 232 resentation and a proof of concept implementation. Specifically, the contributions of this  
 233 work are

- 234 1. a formal description of the topology preserving relationship between data and graphic  
235 via continuous maps
- 236 2. a formal description of the property preservation from data component to visual rep-  
237 resentation as equivariant maps that carry a homomorphism of monoid actions
- 238 3. abstraction of data structure using fiber bundles with schema like fibers to encode  
239 components and topology
- 240 4. algebraic sum operator such that more complex visualizations can be built from simple  
241 ones
- 242 5. a functional oriented visualization tool architecture built on the mathematical model  
243 to demonstrate the utility of the model
- 244 6. a prototype of the architecture built on Matplotlib’s infrastructure to demonstrate the  
245 feasibility of the model

246 In contrast to mathematical models of visualization that aim to evaluate visualization design,  
247 we propose a topological framework for building tools to build visualizations. We defer  
248 judgement of expressivity and effectiveness to developers building domain specific tools, but  
249 provide them the framework to do so.

### 250 3 Topological Artist Model

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

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

that carries a homomorphism of monoid actions  $\varphi : M \rightarrow M'$  [46], which are discussed in detail in section 3.1.2. Given  $M$  on data  $\mathcal{E}$  and  $M'$  on graphic  $\mathcal{H}$ , we propose that artists  $\mathcal{A}$  are equivariant maps

$$\mathcal{A}(m \cdot r) = \varphi(m) \cdot \mathcal{A}(r) \quad (2)$$

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

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

### 3.1 Data Space $E$

Building on Butler’s proposal of using fiber bundles as a common data representation structure for visualization data[41, 42], a fiber bundle is a tuple  $(E, K, \pi, F)$  defined by the projection map  $\pi$

$$F \hookrightarrow E \xrightarrow{\pi} K \quad (3)$$

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

By definition fiber bundles are locally trivial[47, 48], meaning that over a localized neighborhood we can dispense with extra structure on  $E$  and focus on the components and continuity. We use fiber bundles as the data model because they are inclusive enough to express all the types of data described in section 2.3.

### 3.1.1 Variables in Fiber Space $F$

To formalize the structure of the data components, we use notation introduced by Spivak [44] that binds the components of the fiber to variable names. This allows us to describe the components in a schema like way. Spivak constructs a set  $\mathbb{U}$  that is the disjoint union of all possible objects of types  $\{T_0, \dots, T_m\} \in \mathbf{DT}$ , where  $\mathbf{DT}$  are the data types of the variables in the dataset. He then defines the single variable set  $\mathbb{U}_\sigma$

$$\begin{array}{ccc} \mathbb{U}_\sigma & \longrightarrow & \mathbb{U} \\ \pi_\sigma \downarrow & & \downarrow \pi \\ C & \xrightarrow{\sigma} & \mathbf{DT} \end{array} \quad (4)$$

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

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

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

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

which is equivalent to

$$F = F_0 \times \dots \times F_i \times \dots \times F_n \quad (7)$$

which allows us to decouple  $F$  into components  $F_i$ .



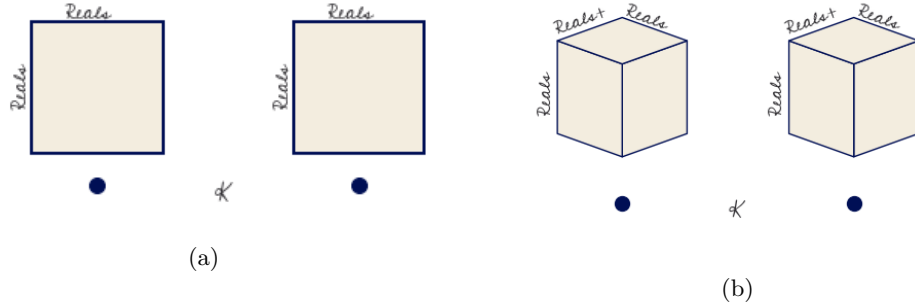


Figure 6: These two datasets have the same base space  $K$  of discrete points, but figure 6a has fiber  $F = \mathbb{R} \times \mathbb{R}$  which is (time, temperature) while figure 6b has fiber  $\mathbb{R} \times \mathbb{R}^+ \times \mathbb{R}$  which is (time, wind=(speed, direction))

For example, the data in figure 6a is a pair of times and °K temperature measurements taken at those times. Time is a positive number of type `datetime` which can be resolved to floats  $\mathbb{U}_{\text{datetime}} = \mathbb{R}$ . Temperature values are real positive numbers  $\mathbb{U}_{\text{float}} = \mathbb{R}^+$ . The fiber is

$$\mathbb{U} = \mathbb{R} \times \mathbb{R}^+ \quad (8)$$

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  $(c = \text{temperature}, T = \text{float}, \mathbb{U}_\sigma = \mathbb{R}^+)$  and is the set of values  $\mathbb{U}_\sigma = \mathbb{R}^+$ . In figure 6b, temperature is replaced with wind. This wind variable is of type `wind` and has two components speed and direction  $\{(s, d) \in \mathbb{R}^2 \mid 0 \leq s, 0 \leq d \leq 360\}$ . Therefore, the fiber is

$$F = \mathbb{R}^+ \times \mathbb{R}^2 \quad (9)$$

270 such that  $F_1$  is specified by  $(c = \text{wind}, T = \text{wind}, \mathbb{U}_\sigma = \mathbb{R}^2)$ . As illustrated in figure 6,  
 271 Spivak's framework provides a consistent way to describe potentially complex components  
 272 of the input data.

### 273 3.1.2 Measurement Scales: Monoid Actions

274 Implementing expressive visual encodings requires formally describing the structure on the  
 275 components of the fiber, which we define by the actions of a monoid on the component. In  
 276 doing so, we specify the properties of the component that must be preserved in a graphic  
 277 representation. While structure on a set of values is often described algebraically as oper-  
 278 ations or through the actions of a group, for example Steven’s scales [49], we generalize to  
 279 monoids to support more component types. Monoids are also commonly found in functional  
 280 programming because they specify compositions of transformations [50, 51].

A monoid [52]  $M$  is a set with an associative binary operator  $*$  :  $M \times M \rightarrow M$ . A  
 monoid has an identity element  $e \in M$  such that  $e * a = a * e = a$  for all  $a \in M$ . As  
 defined on a component of  $F$ , a left monoid action [53, 54] 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  $x \in F_i, e \in M_i$ ,  $e \bullet x = x$

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

$$M = M_0 \times \dots \times M_i \times \dots \times \dots M_n \quad (10)$$

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

283 Steven’s described the measurement scales[49, 55] in terms of the monoid actions on  
 284 the measurements: nominal data is permutable, ordinal data is monotonic, interval data  
 285 is translatable, and ratio data is scalable [56]. For example, given an arbitrary interval  
 286 scale fiber component  $(c = \text{temperature}, T = \text{float}, \mathbb{U}_\sigma = \mathbb{R})$  with with arbitrary monoid  
 287 translation actions chosen for this example:

- 288 • monoid operator addition  $* = +$
- 289 • monoid operations:  $f : x \mapsto x + 1^\circ C$ ,  $g : x \mapsto x + 2^\circ C$

290 • monoid action operator composition  $\bullet = \circ$

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

$$\begin{array}{ccc}
 \mathbb{R} & & \\
 x+1^\circ \downarrow & \searrow (x+1^\circ C) \circ (x+2^\circ C) & \\
 \mathbb{R} & \xrightarrow{x+2^\circ C} & \mathbb{R}
 \end{array} \tag{11}$$

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

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

298 1.  $f(rain) = drizzle$ ,

299 2.  $f(drizzle) = mist$

300 3.  $f(mist) = mist$

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

### 308 3.1.3 Continuity of the Data $K$

309 The base space  $K$  is way to express how the records in  $E$  are connected to each other, for  
 310 example if they are discrete points or if they lie in a 2D continous surface. Connectivity

311 type is assumed in the choice of visualization, for example a line plot implies 1D continuous  
 312 data, but an explicit representation allows for verifying that the topology of the graphic  
 313 representation is equivalent to the topology of the data.



Figure 7: The topological base space  $K$  encodes the connectivity of the data space, for example if the data is independent points or on a plane or a sphere

314 As illustrated in figure 7,  $K$  is akin to an indexing space into  $E$  that describes the  
 315 structure of  $E$ .  $K$  can have any number of dimensions and can be continuous or discrete.

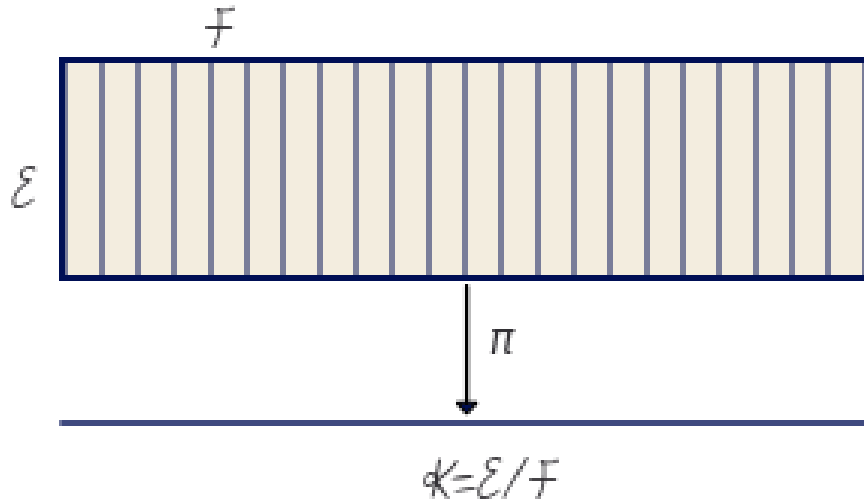


Figure 8: The base space  $E$  is divided into fiber segments  $F$ . The base space  $K$  acts as an index into the records in the fibers.

Formally  $K$  is the quotient space [57] of  $E$  meaning it is the finest space[58] such that every  $k \in K$  has a corresponding fiber  $F_k$ [57]. In figure 8,  $E$  is a rectangle divided by vertical fibers  $F$ , so the minimal  $K$  for which there is always a mapping  $\pi : E \rightarrow K$  is the closed interval  $[0, 1]$ . As with fibers and monoids, we can decompose the total space into components  $\pi : E_i \rightarrow K$  where

$$\pi : E_1 \oplus \dots \oplus E_i \oplus \dots \oplus E_n \rightarrow K \quad (12)$$

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

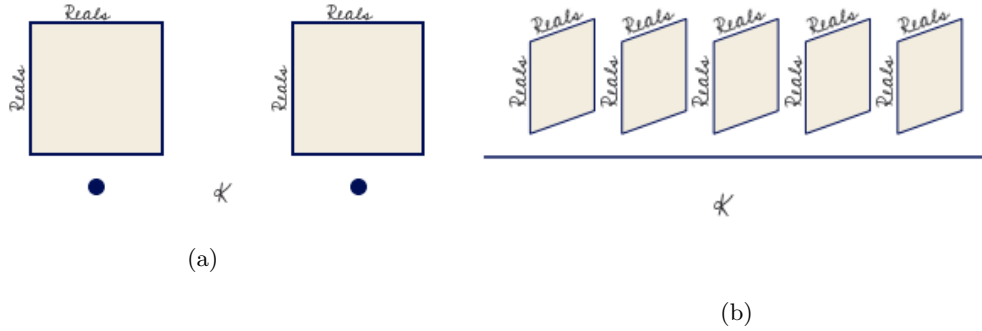


Figure 9: These two datasets have the same (time, temperature) fiber. In figure 9a the total space  $E$  is discrete over points  $k \in K$ , meaning the records in the fiber are also discrete. In figure 9b  $E$  lies over the continuous interval  $K$ , meaning the records in the fiber are sampled from a continuous space.

318 The datasets in figure 9 have the same fiber of (temperature, time). In figure 9a the  
 319 fibers lie over discrete  $K$  such that the records in the datasets in the fiber bundles are  
 320 discrete. The same fiber in figure 9b lies over a continuous interval  $K$  such that the records  
 321 are samples from a continuous function defined on  $K$ . By encoding this continuity in the  
 322 model as  $K$  the data model now explicitly carries information about its structure such  
 323 that the implicit assumptions of the visualization algorithms are now explicit. The explicit

324 topology is a concise way of distinguishing visualizations that appear identical, for example  
 325 heatmaps and images.

#### 326 3.1.4 Data $\tau$

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 [44],  $K$  is a set of row ids,  $F$  is the columns, and the section  $\tau$  returns the record  $r$  at a given key in  $K$ . For any fiber bundle, there exists a map

$$\begin{array}{ccc} F & \hookrightarrow & E \\ & & \pi \downarrow \nearrow \tau \\ & & K \end{array} \quad (13)$$

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

$$\tau(k) = (k, (g_{F_0}(k), \dots, g_{F_n}(k))) \quad (14)$$

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, we can decompose  $\tau$  as

$$\tau = (\tau_0, \dots, \tau_i, \dots, \tau_n) \quad (15)$$

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

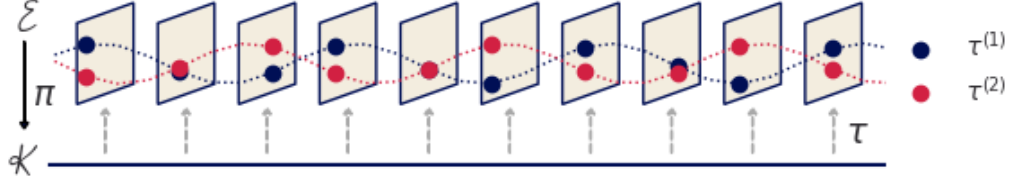


Figure 10: Fiber (time, temperature) with an interval  $K$  basespace. The sections  $\tau^{(1)}$  and  $\tau^{(2)}$  are constrained such that the time variable must be monotonic, which means each section is a timeseries of temperature values. They are included in the global set of sections  $\tau^{(1)}, \tau^{(2)} \in \Gamma(E)$

In the example in figure 10, the fiber is *(time, temperature)* as described in figure 6 and the base space is the interval  $K$ . The section  $\tau^{(1)}$  resolves to a series of monotonically increasing in time records of (time, temperature) values. Section  $\tau^{(2)}$  returns a different timeseries of (time, temperature) values. Both sections are included in the global set of sections  $\tau^{(1)}, \tau^{(2)} \in \Gamma(E)$ .

### 3.1.5 Applications to Data Containers

This model provides a common formalism for widely used data containers without sacrificing the semantic structure embedded in each container. For example, the section can be any instance of a univariate numpy array[59] that stores an image. This could be a section of a fiber bundle where  $K$  is a 2D continuous plane and the  $F$  is  $(\mathbb{R}^3, \mathbb{R}, \mathbb{R})$  where  $\mathbb{R}^3$  is color, and the other two components are the x and y positions of the sampled data in the image. This position information is already implicitly encoded in the array as the index and the resolution of the image being stored. Instead of an image, the numpy array could also store a 2D discrete table. The fiber would not change, but the  $K$  would now be 0D discrete points. These different choices in topology indicate, for example, what sorts of interpolation would be appropriate when visualizing the data.

There are also many types of labeled containers that can richly be described in this framework because of the schema like structure of the fiber. For example, a pandas series which stores a labeled list, or a dataframe[60] which stores a relational table. A series could

store the values of  $\tau^{(1)}$  and a second series could be  $\tau^{(2)}$ . We could also fatten the fiber to hold two temperature series, such that a section would be an instance of a dataframe with a time column and two temperature columns. While the series and dataframe explicitly have a time index column, they are components in our model and the index is assumed to be data independent references such as hashvalues, virtual memory locations, or random number keys.

Where this model particularly shines are N dimensional labeled data structures. For example, an xarray[61] data that stores temperature field could have a  $K$  that is a continuous volume and the components would be the temperature and the time, latitude, and longitude the measurements were sampled at. A section can also be an instance of a distributed data container, such as a dask array [62]. As with the other containers,  $K$  and  $F$  are defined in terms of the index and dtypes of the components of the array. Because our framework is defined in terms of the fiber, continuity, and sections, rather than the exact values of the data, our model does not need to know what the exact values are until the renderer needs to fill in the image.

## 3.2 Graphic Space $H$

We introduce a graphic bundle to hold the essential information necessary to render a graphical design constructed by the artist. As with the data, we can represent the target graphic as a section  $\rho$  of a bundle  $(H, S, \pi, D)$ . The graphic bundle  $H$  consists of a base  $S$  (3.2.1) that is a thickened form of  $K$  a fiber  $D$  (3.2.2) that is an idealized display space, and sections  $\rho$  (3.2.3) that encode a graphic where the visual characteristics are fully specified.

### 3.2.1 Idealized Display $D$

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

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



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

### 3.2.2 Continuity of the Graphic $S$

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

Formally, we require that  $K$  be a deformation retract[63] of  $S$  so that  $K$  and  $S$  have the same homotopy. The surjective map  $\xi : S \rightarrow K$

$$\begin{array}{ccc} E & & H \\ \pi \downarrow & & \pi \downarrow \\ K & \xleftarrow{\xi} & S \end{array} \quad (17)$$

goes from region  $s \in S_k$  to its associated point  $s$ . This means that if  $\xi(s) = k$ , the record at  $k$  is copied over the region  $s$  such that  $\tau(k) = \xi^* \tau(s)$  where  $\xi^* \tau(s)$  is  $\tau$  pulled back over  $S$ .

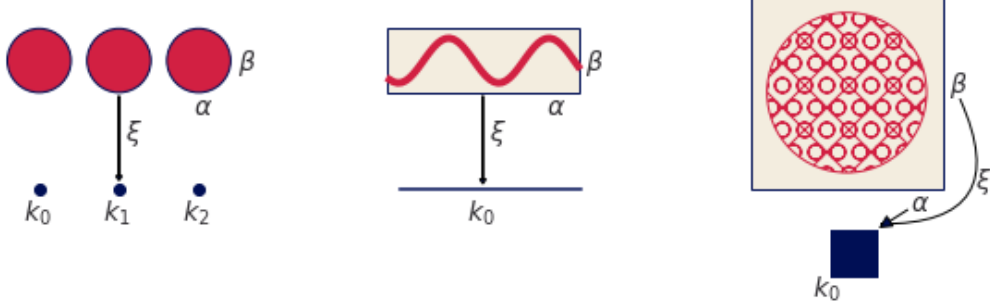


Figure 11: The scatter and line graphic base spaces have one more dimension of continuity than  $K$  so that  $S$  can encode physical aspects of the glyph, such as shape (a circle) or thickness. The image has the same dimension in  $S$  as in  $K$ .

When  $K$  is discrete points and the graphic is a scatter plot, each point  $k \in K$  corresponds to a 2D disk  $S_k$  as shown in figure 11. In the case of 1D continuous data and a line plot, the region  $\beta$  over a point  $\alpha_i$  specifies the thickness of the line in  $S$  for the corresponding  $\tau$  on  $k$ . The image has the same dimensions in data space and graphic space such that no extra dimensions are needed in  $S$ .

The mapping function  $\xi$  provides a way to identify the part of the visual transformation that is specific to the connectivity of the data rather than the values; for example it is common to flip a matrix when displaying an image. The  $\xi$  mapping is also used by interactive visualization components to look up the data associated with a region on screen. One example is to fill in details in a hover tooltip, another is to convert region selection (such as zooming) on  $S$  to a query on the data to access the corresponding record components on  $K$ .

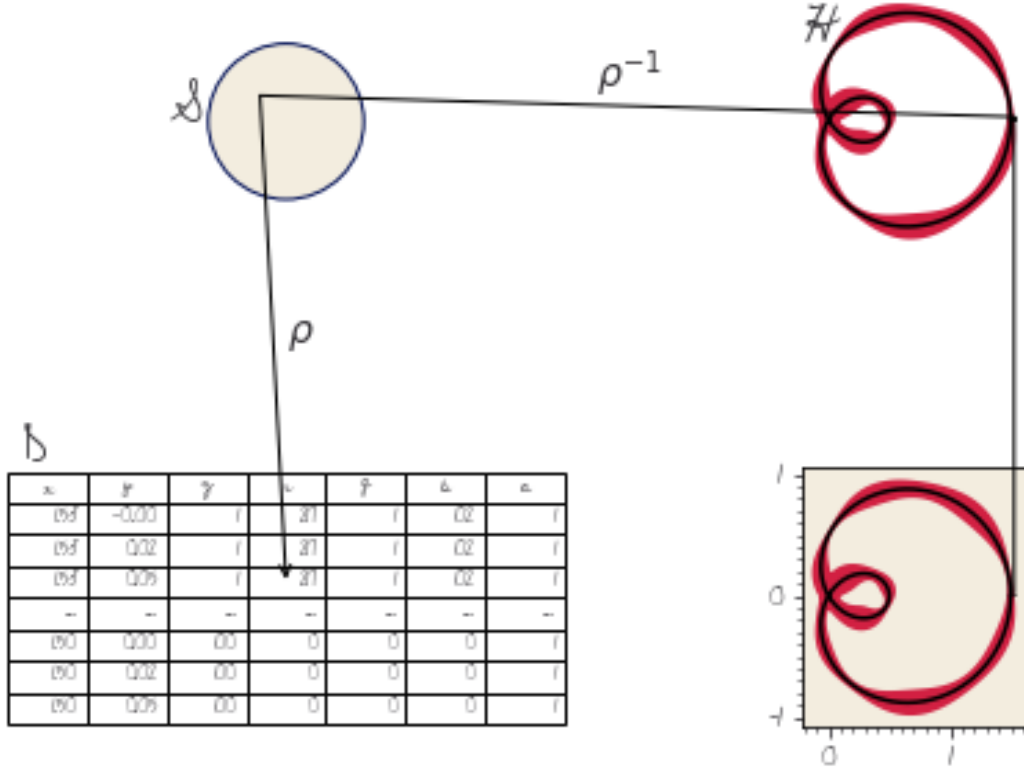


Figure 12: To render a graphic, a pixel  $p$  is selected in the display space, which is defined in the same coordinates as the  $x$  and  $y$  components in  $D$ . The inverse mapping  $\rho_{xy}^{-1}(p)$  returns a region  $S_p \subset S$ .  $\rho(S_p)$  returns the list of elements  $(x, y, r, g, b) \in D$  that lie over  $S_p$ . The integral over the  $(r, g, b)$  elements is the color of the pixel.

399 This section describes how we go from a graphic in an idealized prerender space to a rendered  
 400 image, where the graphic is the section  $\rho : S \rightarrow H$ . It is sufficient to sketch out how an  
 401 arbitrary pixel would be rendered, where a pixel  $p$  in a real display corresponds to a region  
 402  $S_p$  in the idealized display. To determine the color of the pixel, we aggregate the color values  
 403 over the region via integration.

404 For a 2D screen, the pixel is defined as a region  $p = [y_{top}, y_{bottom}, x_{right}, x_{left}]$  of the  
 405 rendered graphic. Since the  $x$  and  $y$  in  $p$  are in the same coordinate system as the  $x$  and  $y$

406 components of  $D$  the inverse map of the bounding box  $S_p = \rho_{xy}^{-1}(p)$  is a region  $S_p \subset S$ .  
 407 To compute the color, we integrate on  $S_p$

$$r_p = \iint_{S_p} \rho_r(s) ds^2 \quad (18)$$

$$g_p = \iint_{S_p} \rho_g(s) ds^2 \quad (19)$$

$$b_p = \iint_{S_p} \rho_b(s) ds^2 \quad (20)$$

408 As shown in figure 12, a pixel  $p$  in the output space is selected and inverse mapped into  
 409 the corresponding region  $S_p \subset S$ . This triggers a lookup of the  $\rho$  over the region  $S_p$ , which  
 410 yields the set of elements in  $D$  that specify the  $(r, g, b)$  values corresponding to the region  
 411  $p$ . The color of the pixel is then obtained by taking the integral of  $\rho_{rgb}(S_p)$ .

412 In general,  $\rho$  is an abstraction of rendering. In very broad strokes  $\rho$  can be a specification  
 413 such as PDF[64], SVG[65], or an openGL scene graph[66]. Alternatively,  $\rho$  can be a rendering  
 414 engine such as cairo[67] or AGG[AntiGrainGeometry]. Implementation of  $\rho$  is out of  
 415 scope for this work,

### 416 3.3 Artist

We propose that the transformation from data to visual representation can be described as a structure preserving map from one topological space to another. We name this map the artist as that is the analogous part of the Matplotlib[7] architecture that builds visual elements. The topological artist  $A$  is a monoid equivariant sheaf map from the sheaf on a data bundle  $E$  which is  $\mathcal{O}(E)$  to the sheaf on the graphic bundle  $H$ ,  $\mathcal{O}(H)$ .

$$A : \mathcal{O}(E) \rightarrow \mathcal{O}(H) \quad (21)$$

417 Sheafs are a mathematical object with restriction maps that define how to glue  $\tau$  over local  
 418 neighborhoods  $U \subseteq K$ , discussed in section ??, such that the  $A$  maps are consistent over

continuous regions of  $K$ . While  $A$  can usually construct graphical elements solely with the data in  $\tau$ , some visualizations, such as line, may also need some finite number  $n$  of derivatives, which is captured by the jet bundle  $\mathcal{J}^n$  [68, 69] with  $\mathcal{J}^0(E) = E$ . In this work, we at most need  $\mathcal{J}^2(E)$  which is the value at  $\tau$  and its first and second derivatives; therefore the artist takes as input the jet bundle  $E' = \mathcal{J}^2(E)$ .

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

$$\begin{array}{ccccccc}
 E' & \xrightarrow{\nu} & V & \xleftarrow{\xi^*} & \xi^*V & \xrightarrow{Q} & H \\
 & \searrow \pi & \downarrow \pi & & \xi^* \pi \downarrow & \swarrow \pi & \\
 & & K & \xleftarrow{\xi} & S & & 
 \end{array} \tag{22}$$

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

### 3.3.1 Visual Fiber Bundle $V$

We introduce a visual bundle  $V$  to store the visual representations the artist needs to assemble into a graphic. The visual bundle  $(V, K, \pi, P)$  has section  $\mu : V \rightarrow K$  that resolves to a visual variable in the fiber  $P$ . The visual bundle  $V$  is the latent space of possible parameters of a visualization type, such as a scatter or line plot. We define  $P$  in terms of the parameters of a visualization libraries compositing functions; for example table 1 is a sample of the fiber space for Matplotlib [6].

$\nu_i$	$\mu_i$	$\text{codomain}(\nu_i) \subset P_i$
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

438 A section  $\mu$  is a tuple of visual values that specifies the visual characteristics of a part of  
 439 the graphic. For example, given a fiber of  $\{xpos, ypos, color\}$  one possible section could be  
 440  $\{.5, .5, (255, 20, 147)\}$ . The  $\text{codomain}(\nu_i)$  determines the monoid actions on  $P_i$ . These fiber  
 441 components are implicit in the library, by making them explicit as components of the fiber  
 442 we can build consistent definitions and expectations of how these parameters behave.

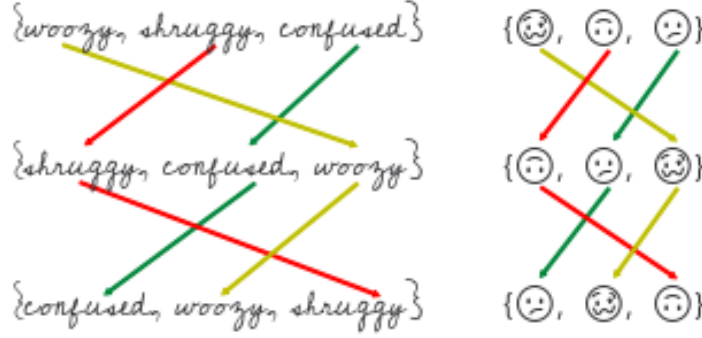


Figure 13: In this artist,  $\nu$  maps the strings to the emojis. This  $\nu$  is equivariant because the monoid actions (which are represented by the colored arrows) are the same on both the  $\tau$  input and  $\mu$  output sets.

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

$$\{\nu_0, \dots, \nu_n\} : \{\tau_0, \dots, \tau_n\} \mapsto \{\mu_0, \dots, \mu_n\} \quad (23)$$

444 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  
 445 there is a monoid  $M_i'$  on  $V_i$ , then there is a monoid homomorphism from  $\varphi : M_i \rightarrow M_i'$   
 446 that  $\nu$  must preserve. As mentioned in section 3.1.2, we choose monoid actions as the basis  
 447 for equivariance because they define the structure on the fiber components.

A validly constructed  $\nu$  is one where the diagram of the monoid transform  $m$  commutes such that

$$\begin{array}{ccc} E_i & \xrightarrow{\nu_i} & V_i \\ m_r \downarrow & & \downarrow m_v \\ E_i & \xrightarrow{\nu_i} & V_i \end{array} \quad (24)$$

In general, the data fiber  $F_i$  cannot be assumed to be of the same type as the visual fiber  $P_i$  and the actions of  $M$  on  $F_i$  cannot be assumed to be the same as the actions of  $M'$  on  $P$ ; therefore an equivariant  $\nu_i$  must satisfy the constraint

$$\nu_i(m_r(E_i)) = \varphi(m_r)(\nu_i(E_i)) \quad (25)$$

such that  $\varphi$  maps a monoid action on data to a monoid action on visual elements. However, we can construct a monoid action of  $M$  on  $P_i$  that is compatible with a monoid action of  $M$  on  $F_i$ . We can compose the monoid actions on the visual fiber  $M' \times P_i \rightarrow P_i$  with the homomorphism  $\varphi$  that takes  $M$  to  $M'$ . This allows us to define a monoid action on  $P$  of  $M$  that is  $(m, v) \rightarrow \varphi(m) \bullet v$ . Therefore, without a loss of generality, we can assume that an action of  $M$  acts on  $F_i$  and on  $P_i$  compatibly such that  $\varphi$  is the identity function.

On example of an equivariant  $\nu$  is illustrated in figure 13, which is a mapping from **Strings** to symbols. The data is an example of a Steven's nominal measurement set, which is defined as having on it permutation group actions

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

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

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

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

$$\nu(x + c) = \nu(x) + c \quad (28)$$



while for ratio data, there must be equivalent scaling[56]

$$\nu(xc) = \nu(x) * c \quad (29)$$

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

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

$$.5 \neq .5 + 2 \quad (31)$$

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

### 457 3.3.3 Graphic Assembler $Q$

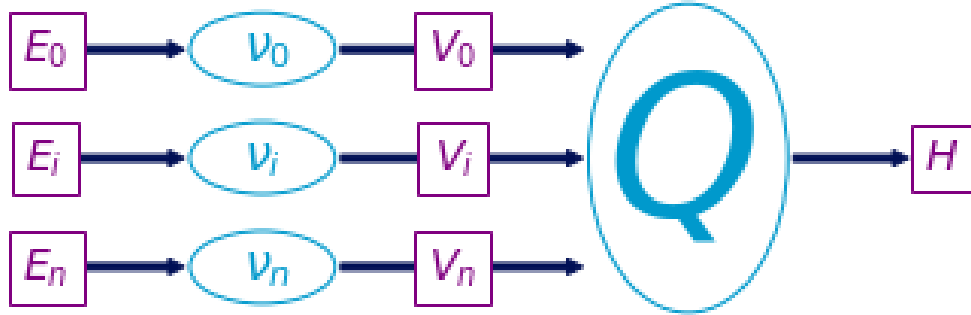


Figure 14:  $\nu_i$  functions convert data  $\tau_i$  to visual characteristics  $\mu_i$ , then  $Q$  assembles  $\mu_i$  into a graphic  $\rho$  such that there is a map  $\xi$  preserving the continuity of the data.  $\rho$  applied to a region of connected components  $S_j$  generates a part of a graphic, for example the point graphical mark.

458 As shown in figure 14, the assembly function  $Q$  combines the fiber  $F_i$  wise  $\nu$  transforms into  
 459 a graphic in  $H$ . Together,  $\nu$  and  $Q$  are a map-reduce operation: map the data into their  
 460 visual encodings, reduce the encodings into a graphic. As with  $\nu$  the constraint on  $Q$  is  
 461 that for every monoid action on the input  $\mu$  there is corresponding monoid action on the  
 462 output  $\rho$ .

While  $\rho$  generates the entire graphic, we will restrict the discussion of  $Q$  to generation of sections of a glyph. We formally describe a glyph as  $Q$  applied to the regions  $k$  that map back to a set of path connected components  $J \subset K$  as input:

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

where the path[70]  $\gamma$  from  $k$  to  $j$  is a continuous function from the interval  $[0,1]$ . We define the glyph as the graphic generated by  $Q(S_j)$

$$H \xrightleftharpoons[\rho(S_j)]{} S_j \xrightleftharpoons[\xi^{-1}(J)]{\xi(s)} J_k \quad (33)$$

463 such that for every glyph there is at least one corresponding region on  $K$ . This is in keeping  
 464 with the definition of glyph as any differentiable element put forth by Ziemkiewicz and  
 465 Kosara[71]. The primitive point, line, and area marks[11, 72] are specially cased glyphs.

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

Lets call the visual representations of the components  $\Gamma(V) = X$  and the graphic  $Q(\Gamma(V)) = Y$ . If for elements of the monoid  $m \in M$  and for all  $\mu, \mu' \in X$ , we define

the monoid action on  $X$  so that it is by definition equivariant

$$Q(\mu) = Q(\mu') \implies Q(m \circ \mu) = Q(m \circ \mu') \quad (34)$$

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

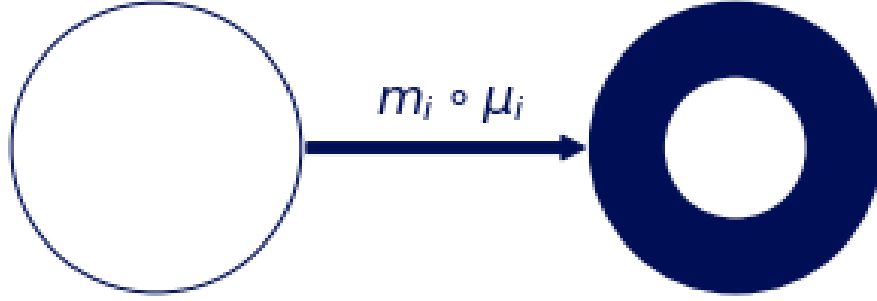


Figure 15: These two glyphs are generated by the same  $Q$  function. The monoid action  $m_i$  on edge thickness  $\mu_i$  of the first glyph yields the thicker edge  $\mu_i'$  in the second glyph.

476 The glyph in figure 15 has the following characteristics  $P$  specified by  $(xpos, ypos, color, thickness)$   
 477 such that one section is  $\mu = (0, 0, 0, 1)$  and  $Q(\mu) = \rho$  generates a piece of the thin hollow  
 478 circle. The equivariance constraint on  $Q$  is that the action  $m = (e, e, e, x + 2)$ , where  $e$  is  
 479 identity, translates  $\mu$  to  $\mu' = (e, e, e, 3)$ . The corresponding action on  $\rho$  causes  $Q(\mu')$  to be  
 480 the thicker circle in figure 15.

#### 481 3.3.4 Assembly $Q$

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

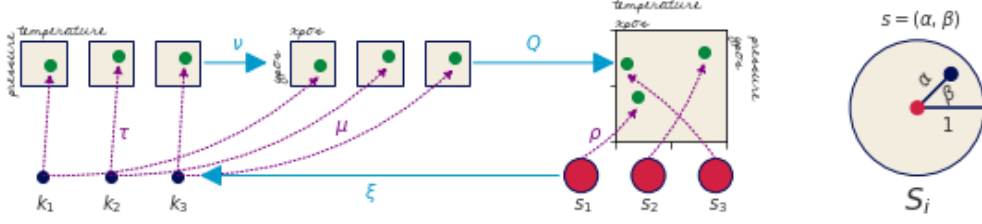


Figure 16: The data is discrete points (temperature, time). Via  $\nu$  these are converted to (xpos, ypos) and pulled over discrete  $S$ . These values are then used to parameterize  $\rho$  which returns a color based on the parameters (xpos,ypos) and position  $\alpha, \beta$  on  $S_k$  that  $\rho$  is evaluated on.

The scatter plot in figure 16 can be defined as  $Q(xpos, ypos)(\alpha, \beta)$  where color  $\rho_{RGB} = (0, 0, 0)$  is defined as part of  $Q$  and  $s = (\alpha, \beta)$  defines the region on  $S$ . The position of this swatch of color can be computed relative to the location on the disc  $S_k$  as shown in figure 16:

$$x = size * \alpha \cos(\beta) + xpos \quad (35)$$

$$y = size * \alpha \sin(\beta) + ypos \quad (36)$$

such that  $\rho(s) = (x, y, 0, 0, 0)$  colors the point (x,y) black. Here *size* can either be defined inside  $Q$  or it could also be a parameter in  $V$  that is passed along with (xpos, ypos). As seen in figure 16, a scatter has a direct mapping from a region on  $S_k$  to its corresponding  $k$ .

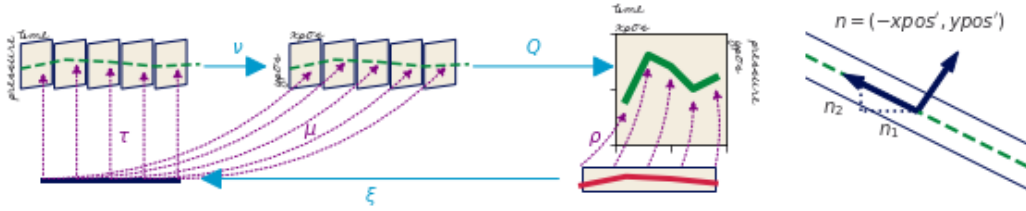


Figure 17: The line fiber (*time*, *temp*) is thickened with the derivative (*time'*, *temperature'*) because that information will be necessary to figure out the tangent to the point to draw a line. This is because the line needs to be pushed perpendicular to the tangent of (xpos, ypos). The data is converted to visual characteristics (xpos, ypos). The  $\alpha$  coordinates on  $S$  specifies the position of the line, the  $\beta$  coordinate specifies thickness.

In contrast to the scatter, the line plot  $Q(xpos, \hat{n}_1, ypos, \hat{n}_2)(\alpha, \beta)$  shown in fig 17 has a  $\xi$  function that is not only parameterized on  $k$  but also on the  $\alpha$  distance along  $k$  and corresponding region in  $S$ . The line also exemplifies the need for the jet since the line needs to know the tangent of the data to draw an envelope above and below each  $(xpos, ypos)$  such that the line appears to have a thickness. The magnitude of the slope is

$$|n| = \sqrt{n_1^2 + n_2^2} \quad (37)$$

such that the normal is

$$\hat{n}_1 = \frac{n_1}{|n|}, \quad \hat{n}_2 = \frac{n_2}{|n|} \quad (38)$$

which yields components of  $\rho$

$$x = xpos(\xi(\alpha)) + width * \beta \hat{n}_1(\xi(\alpha)) \quad (39)$$

$$y = ypos(\xi(\alpha)) + width * \beta \hat{n}_2(\xi(\alpha)) \quad (40)$$

487 where  $(x,y)$  look up the position  $\xi(\alpha)$  on the data. At that point, we also look up the the  
 488 derivatives  $\hat{n}_1, \hat{n}_2$  which are then multiplied by a *width* parameter to specify the thickness.  
 489 As with the *size* parameter in scatter, *width* can be defined in  $Q$  or as a component of  $V$ .

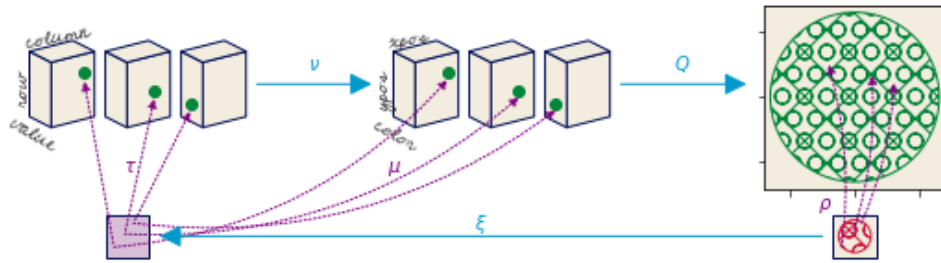


Figure 18: The only visual parameter an image requires is color since  $\xi$  encodes the mapping between position in data and position in graphic.

The image  $Q(xpos, ypos, color)$  in figure 18 is a direct lookup into  $\xi : S \rightarrow K$ . Since  $K$  is 2D continuous space, the indexing variables  $(\alpha, \beta)$  define the distance along the space.

This is then used by  $\xi$  to map into  $K$  to lookup the color values

$$R = R(\xi(\alpha, \beta)) \quad (41)$$

$$G = G(\xi(\alpha, \beta)) \quad (42)$$

$$B = B(\xi(\alpha, \beta)) \quad (43)$$

490 that the data values have been mapped into. In the case of an image, the indexing mapper  
 491  $\xi$  may do some translating to a convention expected by  $Q$ , for example reorientng the array  
 492 such that the first row in the data is at the bottom of the graphic.

### 493 3.3.5 Assembly factory $\hat{Q}$

494 The graphic base space  $S$  is not accessible in many architectures, including Matplotlib;  
 495 instead we can construct a factory function  $\hat{Q}$  over  $K$  that can build a  $Q$ . As shown in  
 496 eq 22,  $Q$  is a bundle map  $Q : \xi^*V \rightarrow H$  where  $\xi^*V$  and  $H$  are both bundles over  $S$ .

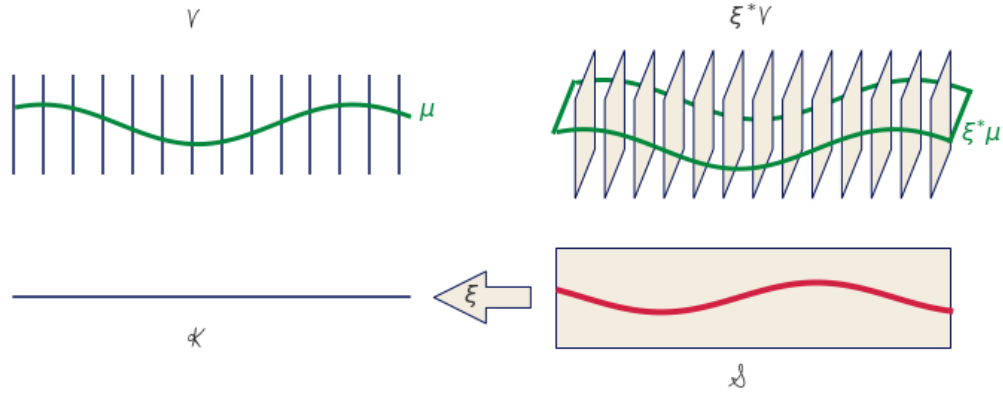


Figure 19: The pullback of the visual bundle  $\xi^*V$  is the replication of a  $\mu$  over all points  $s$  that map back to a single  $k$ . Because the  $\mu$  is the same, we can construct a  $\hat{Q}$  on  $\mu$  over  $k$  that will fabricate the  $Q$  for the equivalent region of  $s$  associated to that  $k$

The preimage of the continuity map  $\xi^{-1}(k) \subset S$  is such that many graphic continuity points  $s \in S_K$  go to one data continuity point  $k$ ; therefore, by definition the pull back of  $\mu$

$$\xi^*V|_{\xi^{-1}(k)} = \xi^{-1}(k) \times P \quad (44)$$

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

This copying is illustrated in figure 19, where the 1D fiber  $P \hookrightarrow V$  over  $K$  is copied repeatedly to become the 2D fiber  $P^*\mu \hookrightarrow \xi^*V$  with identical components over  $S$ . Given the section  $\xi^*\mu$  pulled back from  $\mu$  and the point  $s \in \xi^{-1}(k)$ , there is a direct map from  $\mu$  on a point  $k$ , there is a direct map from the visual section over data base space  $(k, \mu(k)) \mapsto (s, \xi^*\mu(s))$  to the visual section  $\xi^*\mu$  over graphic base space. This map means that the pulled back section  $\xi^*\mu(s) = \xi^*(\mu(k))$  is the section  $\mu$  copied over all  $s$ . This means that  $\xi^*\mu$  is identical for all  $s$  where  $\xi(s) = k$ , which is illustrated in figure 19 as each dot on  $P$  is equivalent to the line intersection  $P^*\mu$ .

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

$$\rho(s) := Q(\xi^*\mu)(s) \quad (45)$$

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

$$\hat{Q}(\mu(k))(s) := Q((\xi^*\mu)(s)) \quad (46)$$

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

510 Factoring out  $s$  from equation 46 yields  $\hat{Q}(\mu(k)) = Q(\xi^*\mu)$  where  $Q$  is no longer bound  
 511 to input but  $\hat{Q}$  is still defined in terms of  $K$ . In fact,  $\hat{Q}$  is a map from visual space to  
 512 graphic space  $\hat{Q} : \Gamma(V) \rightarrow \Gamma(H)$  locally over  $k$  such that it can be evaluated on a single  
 513 visual record  $\hat{Q} : \Gamma(V_k) \rightarrow \Gamma(H|_{\xi^{-1}(k)})$ . This allows us to construct a  $\hat{Q}$  that only depends  
 514 on  $K$ , such that for each  $\mu(k)$  there is part of  $\rho|_{\xi^{-1}(k)}$ . The construction of  $\hat{Q}$  allows us  
 515 to retain the functional map reduce benefits of  $Q$  without having to majorly restructure  
 516 the existing pipeline for libraries that delegate the construction of  $\rho$  to a back end such as  
 517 Matplotlib.

### 518 3.3.6 Sheafs

The restriction maps of a sheaf describe how local  $\tau$  can be glued into larger sections [73, 74]. As part of the definition of local triviality, there is an open neighborhood  $U \subset K$  for every  $k \in K$ . We can define the inclusion map  $\iota : U \rightarrow K$  which pulls  $E$  over  $U$

$$\begin{array}{ccc} \iota^*E & \xrightarrow{\iota^*} & E \\ \pi \downarrow \Big) \iota^*\tau & & \pi \downarrow \Big) \tau \\ U & \xrightarrow{\iota} & K \end{array} \quad (47)$$

519 such that the pulled back  $\iota^*\tau$  only contains records over  $U \subset K$ . By gluing  $\iota^*\tau$  together, the  
 520 sheaf is putting a continuous structure on local sections which allows for defining a section  
 521 over a subset in  $K$ . That section over subset  $K$  maps to the graphic generated by  $A$  for  
 522 visualizations such as sliding windows[75, 76] streaming data, or navigation techniques such  
 523 as pan and zoom[77].

### 524 3.3.7 Composition of Artists: +

525 To build graphics that are the composites of multiple artists, we define a simple addition  
 526 operator that is the disjoint union of fiber bundles  $E$ . For example, a scatter plot  $E_1$  and a  
 527 line plot  $E_2$  have different  $K$  that are mapped to separate  $S$ . To fully display both graphics,  
 528 the composite graphic  $A_1 + A_2$  needs to include all records on both  $K_1$  and  $K_2$ , which are the  
 529 sections on the disjoint union  $K_1 \sqcup K_2$ . This in turn yields disjoint graphics  $S_1 \sqcup S_2$  rendered



530 to the same image. Constraints can be placed on the disjoint union such as that the fiber  
531 components need to have the same  $\nu$ position encodings or that the position  $\mu$  need to be in a  
532 specified range. There is a second type of composition where  $E_1$  and  $E_2$  share a base space  
533  $K_2 \hookrightarrow K_1$  such that the the artists can be considered to be acting on different components  
534 of the same section. This type of composition is important for creating visualizations where  
535 elements need to update together in a consistent way, such as multiple views [78, 79] and  
536 brush-linked views[80, 81].

### 537 3.3.8 Equivalence class of artists $A'$

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

545 For example, a scatter plot of red circles is the output of one artist, a scatter plot of green  
546 squares the output of another. These two artists are equivalent since their only difference  
547 is in the literal visual encodings (color, shape). Shape and color could also be defined in  $Q$   
548 but the position must come from the fiber  $P = (xpos, ypos)$  since fundamentally a scatter  
549 plot is the plotting of one position against another[9]. We also use this criteria to identify  
550 derivative types, for example the bubble chart[21] is a type of scatter where by definition the  
551 glyph size is mapped from the data. The criteria for equivalence class membership serves  
552 as the basis for evaluating invariance[kindlmann2014algebraic].

## 553 4 Prototype Implementation: Matplottoy

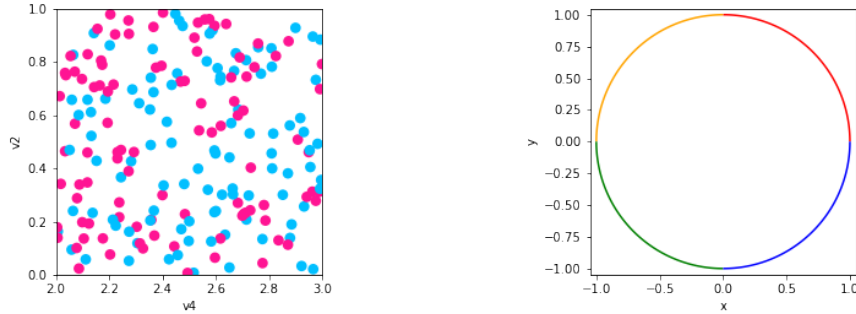


Figure 20: Scatter plot and line plot implemented using prototype artists and data models, building on Matplotlib rendering.

554 To prototype our model, we implemented the artist classes for the scatter and line plots  
 555 shown in figure 20 because they differ in every attribute: different visual channels  $\nu$  that  
 556 composite to different marks  $Q$  with different continuities  $\xi$ . We make use of the Matplotlib  
 557 figure and axes artists [6, 7] so that we can initially focus on the data to graphic transfor-  
 558 mations. We also exploit the Matplotlib transform stack to transform data coordinates into  
 559 screen coordinates. To generate the images in figure 20, we instantiate **fig**, **ax** artists that  
 560 will contain the new **Point**, **Line** primitive objects we implemented based on our topology  
 561 model.

---

```

1 fig, ax = plt.subplots()
2 artist = Point(data, transforms)
3 ax.add_artist(artist)

```

---



---

```

1 fig, ax = plt.subplots()
2 artist = Line(data, transforms)
3 ax.add_artist(artist)

```

---

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

## 572 4.1 Artist Class $A'$

573 The artist is the piece of the Matplotlib architecture that constructs an internal representa-  
 574 tion of the graphic that the render then uses to draw the graphic. In the prototype artist,  
 575 `transform` is a dictionary of the form `{parameter:(variable, encoder)}` where parame-  
 576 ter is a component in  $P$ , variable is a component in  $F$ , and the  $\nu$  encoders are passed in as  
 577 functions or callable objects. The data bundle  $E$  is passed in as a `data` object. By binding  
 578 data and transforms to  $A'$  inside `__init__`, the `draw` method is a fully specified artist  $A$ .

---

```

1  class ArtistClass(matplotlib.artist.Artist):
2      def __init__(self, data, transforms, *args, **kwargs):
3          # properties that are specific to the graphic but not the channels
4          self.data = data
5          self.transforms = transforms
6          super().__init__(*args, **kwargs)
7
8      def assemble(self, **args):
9          # set the properties of the graphic
10

```

```

11     def draw(self, renderer):
12         # returns K, indexed on fiber then key
13         # is passed the
14         view = self.data.view(self.axes)
15         # visual channel encoding applied fiberwise
16         visual = {p: t['encoder'](view[t['name']])
17                  for p, t in self.transforms.items()}
18         self.assemble(**visual)
19         # pass configurations off to the renderer
20         super().draw(renderer)

```

---

579 The data is fetched in section  $\tau$  via a `view` method on the data because the input to the  
580 artist is a section on  $E$ . The `view` method takes the `axes` attribute because it provides the  
581 region in graphic coordinates  $S$  that we can use to query back into data to select a subset  
582 as discussed in section 3.3.6. The  $\nu$  functions are then applied to the data to generate the  
583 visual section  $\mu$  that here is the object `visual`. The conversion from data to visual space is  
584 simplified here to directly show that it is the encoding  $\nu$  applied to the component. In the  
585 full implementation, we allow for fixed visual parameter, such as setting a constant color  
586 for all sections, by verifying that the named component is in  $F$  before accessing the data.  
587 If the data component name is not in  $F$  this is interpreted to mean this component is a  
588 thickening of  $V$  that could be pulled back to  $E$  via an inverse identity  $\nu$ .

589 The components of the visual object, denoted by the Python unpacking convention  
590 `**visual` are then passed into the `assemble` function that is  $\hat{Q}$ . This assembly function  
591 is responsible for generating a representation such that it could be serialized to recreate a  
592 static version of the graphic. Although `assemble` could be implemented outside the class  
593 such that it returns an object the artist could then parse to set attributes, the attributes are  
594 directly set here to reduce indirection. This artist is not optimized because we prioritized  
595 demonstrating the separability of  $\nu$  and  $\hat{Q}$ . The last step in the artist function is handing

596 itself off to the renderer. The extra `*arg, **kwargs` arguments in `__init__`, `draw` are  
 597 artifacts of how these objects are currently implemented in Matplotlib.

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

---

```

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

```

---

603 The `view` method repackages the data as a fiber component indexed table of vertices. Even  
 604 though the `view` is fiber indexed, each vertex at an index  $k$  has corresponding values in  
 605 section  $\tau(k_i)$ . This means that all the data on one vertex maps to one glyph. To ensure the  
 606 integrity of the section, `view` must be atomic. This means that the values cannot change  
 607 after the method is called in `draw` until a new call in `draw`. We put this constraint on the  
 608 return of the `view` method so that we do not risk race conditions.

609 This table is converted to a table of visual variables and is then passed into `assemble`.  
 610 In `assemble`, the  $\mu$  components are used to construct the vector path of each circular  
 611 marker with center `(x,y)` and size `x` and set the colors of each circle. This is done via the  
 612 `Path.circle` object. As mentioned in sections ?? and 3.3.3, this assembly function could  
 613 as easily be implemented such that it was fed one  $\tau(k)$  at a time.

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

---

```

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

```

---

617 In the `Line` artist, `view` returns a table of edges. Each edge consists of (x,y) points sampled  
 618 along the line defined by the edge and information such as the color of the edge. As with  
 619 `Point`, the data is then converted into visual variables. In `assemble`, this visual represen-  
 620 tation is composed into a set of line segments, where each segment is the array generated  
 621 by `np.vstack((vx, vy))`. Then the colors of each line segment are set. The colors are  
 622 guaranteed to correspond to the correct segment because of the atomicity constraint on  
 623 `view`.

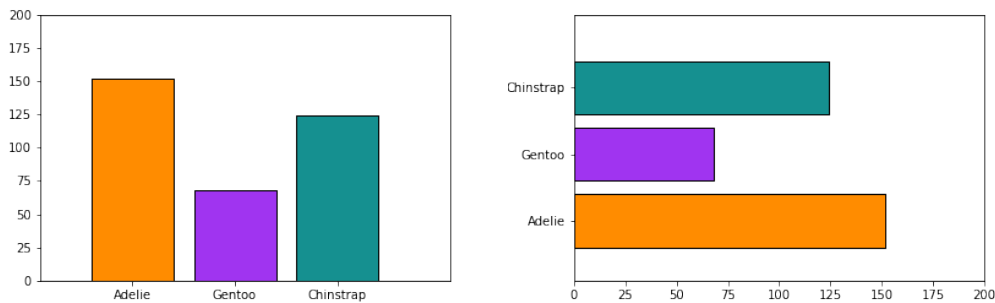


Figure 21: Frequency of Penguin types visualized as discrete bars.

624 The bar charts in figure 21 are generated with a `Bar` artist. The artist has required  
 625 visual parameters  $P$  of (position, length), and an additional parameter `orientation` which  
 626 controls whether the bars are arranged vertically or horizontally. This parameter only applies  
 627 holistically to the graphic and never to individual data parameters, and highlights how the  
 628 model encourages explicit differentiation between parameters in  $V$  and graphic parameters  
 629 applied directly to  $\hat{Q}$ .

---

```

1  class Bar(mcollections.Collection):
2      def __init__(self, data, transforms, orientation='v', *args, **kwargs):
3          """
4              orientation: str, optional
5                  v: bars aligned along x axis, heights on y
6                  h: bars aligned along y axis, heights on x
7          """
8          self.orientation = orientation
9          super().__init__(*args, **kwargs)
10         self.data = data
11         self.transforms = copy.deepcopy(transforms)
12
13     def assemble(self, position, length, floor=0, width=0.8,
14                 facecolors='CO', edgecolors='k', offset=0):
15         #set some defaults
16         width = itertools.repeat(width) if np.isscalar(width) else width
17         floor = itertools.repeat(floor) if np.isscalar(floor) else (floor)
18
19         # offset is passed through via assemblers such as multigroup,
20         # not supposed to be directly tagged to position
21         position = position + offset
22

```

```

23     def makeBars(xval, xoff, yval, yoff):
24         return [[(x, y), (x, y+yo), (x+xo, y+yo), (x+xo, y), (x, y)]
25                 for (x, xo, y, yo) in zip(xval, xoff, yval, yoff)]
26         #build bar glyphs based on graphic parameter
27         if self.orientation in {'vertical', 'v'}:
28             verts = makeBars(position, width, floor, length)
29         elif self.orientation in {'horizontal', 'h'}:
30             verts = makeBars(floor, length, position, width)
31
32         self._paths = [mpath.Path(xy, closed=True) for xy in verts]
33         self.set_edgecolors(edgecolors)
34         self.set_facecolors(facecolors)
35
36     def draw(self, renderer, *args, **kwargs):
37         view = self.data.view(self.axes)
38         visual = {}
39         for (p, t) in self.transforms.items():
40             if isinstance(t, dict):
41                 if t['name'] in self.data.FB.F and 'encoder' in t:
42                     visual[p] = t['encoder'](view[t['name']])
43                 elif 'encoder' in t: # constant value
44                     visual[p] = t['encoder'](t['name'])
45                 elif t['name'] in self.data.FB.F: # identity
46                     visual[p] = view[t['name']]
47             else: # no transform
48                 visual[p] = t
49         self.assemble(**visual)
50         super().draw(renderer, *args, **kwargs)

```

---



630 The **draw** method here has a more complex unpacking of visual encodings to support passing  
 631 in visual component data directly. This is vastly simplifies building composite objects as  
 632 the alternative would be higher order functions that take as input the transforms passed in  
 633 by the user. This construction supports a constant visual parameter, an identity transform  
 634 where the value is the same in  $E$  and  $V$ , and setting the visual component directly. The  
 635 **assemble** function constructs bars and sets their face and edge colors. The **make\_bars**  
 636 function converts the input position and length to the coordinates of a rectangle of the given  
 637 width. Defaults are provided for 'width' and 'floor' to make this function more reusable.  
 638 Typically the defaults are used for the type of chart shown in figure 21, but these visual  
 639 variables are often set when building composite versions of this chart type as discussed in  
 640 section 4.4.

## 641 4.2 Encoders $\nu$

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

---

```

1 cmap = color.Categorical({'true':'deeppink', 'false':'deepskyblue'})
2 transforms = {'x': {'name': 'v4', 'encoder': lambda x: x},
3               'y': {'name': 'v2', 'encoder': lambda x: x},
4               'facecolors': {'name': 'v3', 'encoder': cmap},
5               's': {'name': None, 'encoder': lambda _: itertools.repeat(.02)}}

```

---

647 where the position  $(x, y)$   $\nu$  transformers are identity functions. The size  $s$  transformer is not  
 648 acting on a component of  $F$ , instead it is a  $\nu$  that returns a constant value. While size could  
 649 be embedded inside the **assemble** function, it is added to the transformers to illustrate user  
 650 configured visual parameters that could either be constant or mapped to a component in  $F$ .

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

---

```
1 class Categorical:
2     def __init__(self, mapping):
3         # check that the conversion is to valid colors
4         assert(mcolors.is_color_like(color) for color in mapping.values())
5         self._mapping = mapping
6
7     def __call__(self, value):
8         # convert value to a color
9         return [mcolors.to_rgba(self._mapping[v]) for v in values]
```

---

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

---

```
1 def test_nominal(values, encoder):
2     m1 = list(zip(values, encoder(values)))
3     random.shuffle(values)
4     m2 = list(zip(values, encoder(values)))
5     assert sorted(m1) == sorted(m2)
```

---

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

### 661 4.3 Data $E$

662 The data input into the **Artist** will often be a wrapper class around an existing data  
 663 structure. This wrapper object must specify the fiber components  $F$  and connectivity  $K$   
 664 and have a **view** method that returns an atomic object that encapsulates  $\tau$ . The object  
 665 returned by the view must be key valued pairs of {**component name** : **component section**}  
 666 where each section is a component as defined in equation 15. To support specifying the fiber  
 667 bundle, we define a **FiberBundle** data class[82]

---

```

1  @dataclass
2  class FiberBundle:
3      """
4      Attributes
5      -----
6      K: {'tables': []}
7      F: {variable name: type}
8      """
9      K: dict
10     F: dict

```

---

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

674 To generate the scatter plot in figure 20, we fully specify a dataset with random keys  
 675 and values in a section chosen at random from the corresponding fiber component. The  
 676 fiberbundle **FB** is a class level attribute since all instances of **VertexSimplex** come from the  
 677 same fiberbundle.

---

```

1 class VertexSimplex: #maybe change name to something else
2     """Fiberbundle is consistent across all sections
3     """
4     FB = FiberBundle({'tables': ['vertex']},
5                       {'v1': float, 'v2': str, 'v3': float})
6
7     def __init__(self, sid = 45, size=1000, max_key=10**10):
8         # create random list of keys
9     def tau(self, k):
10        # e1 is sampled from F1, e2 from F2, etc...
11        return (k, (e1, e2, e3, e4))
12
13    def view(self, axes):
14        table = defaultdict(list)
15        for k in self.keys:
16            table['index'] = k
17            # on each iteration, add one (name, value) pair per component
18            for (name, value) in zip(self.FB.fiber.keys(), self.tau(k)[1]):
19                table[name].append(value)
20        return table

```

---

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

---

```

1 class EdgeSimplex:
2
3     FB = FiberBundle({'tables': ['vertex','edge']],
4                       {'x' : float, 'y': float,
5                        'color':mtypes.Color()})
6
7     def __init__(self, num_edges=4, num_samples=1000):
8         self.keys = range(num_edge) #edge id
9         # distance along edge
10        self.distances = np.linspace(0,1, num_samples)
11        # half generlized representation of arcs on a circle
12        self.angle_samples = np.linspace(0, 2*np.pi, len(self.keys)+1)
13
14    @staticmethod
15    def _color(edge):
16        colors = ['red','orange', 'green','blue']
17        return colors[edge%len(colors)]
18
19    @staticmethod
20    def _xy(edge, distances, start=0, end=2*np.pi):
21        # start and end are parameterizations b/c really there is
22        angles = (distances *(end-start)) + start
23        return np.cos(angles), np.sin(angles)
24
25    def tau(self, k): #will fix location on page on revision
26        x, y = self._xy(k, self.distances,
27                        self.angle_samples[k], self.angle_samples[k+1])
28        color = self._color(k)
29        return (k, (x, y, color))

```

```

29
30     def view(self, axes):
31         table = defaultdict(list)
32         for k in self.keys:
33             table['index'].append(k)
34             # (name, value) pair, value is [x0, ..., xn] for x, y
35             for (name, value) in zip(self.FB.fiber.keys(), self.tau(k, simplex)[1]):
36                 table[name].append(value)

```

---

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

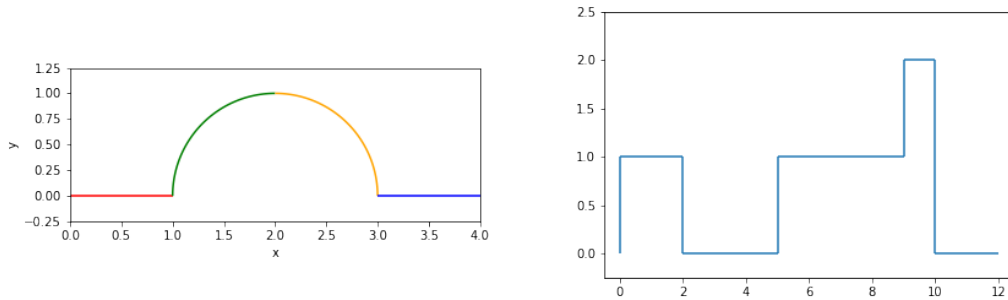


Figure 22: Continuous and discontinuous lines as defined via the same data model, and generated with the same `A'Line`

689 The graphics in figure 22 are made using the `Line` artist and the `Graphline` data source

---

```

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

```

```

3         #s set args as attributes and generate distance
4         if connect: # test connectivity if edges are continuous
5             assert edge_table.keys() == self.FB.F.keys()
6             assert is_continuous(vertex_table)
7
8     def tau(self, k):
9         # evaluates functions defined in edge table
10        return(k, (self.edges[c][k](self.distances) for c in self.FB.F.keys()))
11
12    def view(self, axes):
13        """walk the edge_vertex table to return the edge function
14        """
15        table = defaultdict(list)
16        #sort since intervals lie along number line and are ordered pair neighbors
17        for (i, (start, end)) in sorted(zip(self.ids, self.vertices), key=lambda v:v[1][0]):
18            table['index'].append(i)
19            # same as view for line, returns nested list
20            for (name, value) in zip(self.FB.F.keys(), self.tau(i, simplex)[1]):
21                table[name].append(value)
22        return table

```

---

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

---

```

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

```

---

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

---

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

---

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

## 698 4.4 Case Study: Penguins

699 For this case study, we use the Palmer Penguins dataset[83, 84] since it is multivariate and  
700 has a varying number of penguins. We use a version of the data packaged as a pandas  
701 dataframe[85] since that is a very commonly used Python labeled data structure. The  
702 wrapper is very thin because there is explicitly only one section.

---

```
1 class DataFrame:
2     def __init__(self, dataframe):
3         self.FB = FiberBundle(K = {'tables':['vertex']},
4                                   F = dict(dataframe.dtypes))
5         self._tau = dataframe.iloc
6         self._view = dataframe
7
8     def view(self, axes=None):
9         return self._view
```

---

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



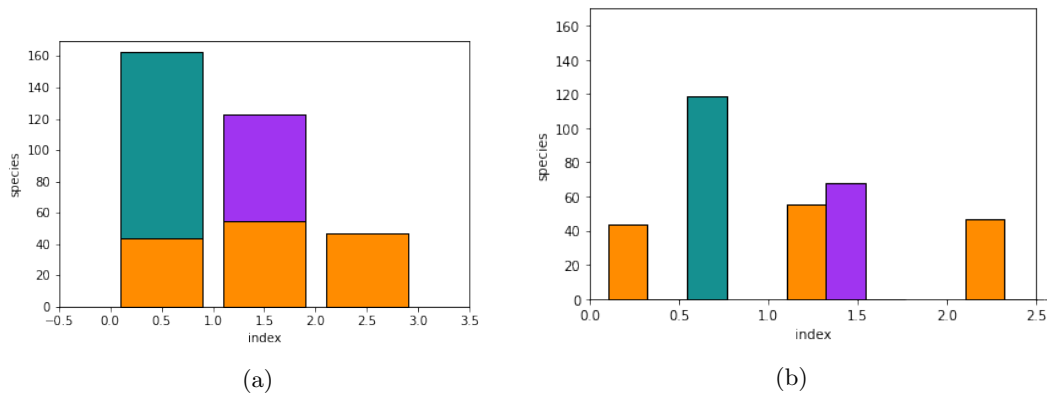


Figure 23: Penguin count disaggregated by island and species

708 The stacked and grouped bar charts in figure 23 are both out of `Bar` artists such that  
 709 the difference between `StackedBar` and `GroupedBar` is specific to the ways in which the  
 710 `Bar` are stitched together. These two artists have identical constructors and `draw` methods.  
 711 As with `Bar`, the orientation is set in the constructor. In both these artists, we separate  
 712 the transforms applied to only one component and the case `mtransforms` where the same  
 713 transform is applied to multiple components such that `V` has multiple components that map  
 714 to the same retinal variable.

---

```

1 class StackedBar(martist.Artist):
2     def __init__(self, data, transforms, mtransforms, 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        super().__init__(*args, **kwargs)
```

```

12     self.data = data
13     self.orientation = orientation
14     self.transforms = copy.deepcopy(transforms)
15     self.mtransforms = copy.deepcopy(mtransforms)
16
17     def assemble(self):
18         view = self.data.view(self.axes)
19         self.children = [] # list of bars to be rendered
20         floor = 0
21         for group in self.mtransforms:
22             # pull out the specific group transforms
23             group['floor'] = floor
24             group.update(self.transforms)
25             bar = Bar(self.data, group, self.orientation, transform=self.axes.transData)
26             self.children.append(bar)
27             floor += view[group['length']]['name']
28
29
30     def draw(self, renderer, *args, **kwargs):
31         # all the visual conversion gets pushed to child artists
32         self.assemble()
33         #self._transform = self.children[0].get_transform()
34         for artist in self.children:
35             artist.draw(renderer, *args, **kwargs)

```

---

715 Since all the visual transformation is passed through to `Bar`, the `draw` method does not  
716 do any visual transformations. In `StackedBar` the `view` is used to adjust the `floor` for  
717 every subsequent bar chart since a stacked bar chart is bar chart area marks concatenated  
718 together in the `length` parameter. In contrast, `GroupedBar` does not even need the `view`, but

719 instead keeps track of the relative position of each group of bars in the visual only variable  
720 `offset`.

---

```
1 class GroupedBar(martist.Artist):
2     def assemble(self):
3         self.children = [] # list of bars to be rendered
4         ngroups = len(self.mtransforms)
5
6         for gid, group in enumerate(self.mtransforms):
7             group.update(self.transforms)
8             width = group.get('width', .8)
9             group['width'] = width/ngroups
10            group['offset'] = gid/ngroups*width
11            bar = Bar(self.data, group, self.orientation, transform=self.axes.transData)
12            self.children.append(bar)
```

---

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

---

```
1 transforms = {'position': {'name': 'island',
2                             'encoder': position.Nominal({'Biscoe':0.1, 'Dream':1.1, 'Torgersen':2.1})}}
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}}]
```

---

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

---

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

---

728 but differ in assembly  $\hat{Q}$ . By decomposing the architecture into data, visual encoding,  
 729 and assembly steps, we are able to build components that are more flexible and also more self  
 730 contained than the existing code base. While very rough, this API demonstrates that the  
 731 ideas presented in the math framework are implementable. For example, the `draw` function  
 732 that maps most closely to  $A$  is functional, with state only being necessary for bookkeeping  
 733 the many inputs that the function requires. In choosing a functional approach, if not  
 734 implementation, we provide a framework for library developers to build reusable encoder  
 735  $\nu$  assembly  $\hat{Q}$  and artists  $A$ . We argue that if these functions are built such that they  
 736 are equivariant with respect to monoid actions and the graphic topology is a deformation  
 737 retraction of the data topology, then the artist by definition will be a structure and property  
 738 preserving map from data to graphic.

## 739 5 Discussion

740 This work contributes a mathematical description of the mapping  $A$  from data to visual rep-  
 741 resentation. Combining Butler’s proposal of a fiber bundle model of visualization data with  
 742 Spivak’s formalism of schema lets this model support a variety of datasets, including discrete  
 743 relational tables, multivariate high resolution spatio temporal datasets, and complex net-  
 744 works. Decomposing the artist into encoding  $\nu$ , assembly  $Q$ , and reindexing  $\xi$  provides the  
 745 specifications for producing visualization where the structure and properties match those  
 746 of the input data. These specifications are that the graphic must have continuity equiva-

747 lent to the data, and that the visual characteristics of the graphics are equivariant to their  
748 corresponding components under monoid actions. This model defines these constraints on  
749 the transformation function such that they are not specific to any one type of encoding or  
750 visual characteristic. Encoding the graphic space as a fiber bundle provides a structure rich  
751 abstraction of the target graphical design in the target display space.

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

## 763 5.1 Limitations

764 So far this model has only been worked out for a single data set tied to a primitive mark,  
765 but it should be extensible to compositing datasets and complex glyphs. The examples and  
766 prototype have so far only been implemented for the static 2D case, but nothing in the math  
767 limits to 2D and expansion to the animated case should be possible because the model is  
768 formalized in terms of the sheaf. While this model supports equivariance of figurative glyphs  
769 generated from parameters of the data[86, 87], it currently does not have a way to evaluate  
770 the semantic accuracy of the figurative representation. Effectiveness is out of scope for this  
771 model because it is not part of the structure being preserved, but potentially a developer  
772 building a domain specific library with this model could implement effectiveness criteria in  
773 the artists. Also, even though the model is designed to be backend and format independent,  
774 it has only really been tested against PNGs rendered with the AGG backend. It is especially

unknown how this framework interfaces with high performance rendering libraries such as  
openGL[66]. Because this model has been limited to the graphic design space, it does not  
address the critical task of laying out the graphics in the image

This model and the associated prototype is deeply tied to Matplotlib’s existing architecture. While the model is expected to generalize to other libraries, such as those built on Mackinlay’s APT framework, this has not been worked through. In particular, Mackinlay’s formulation of graphics as a language with semantic and syntax lends itself a declarative interface[88], which Heer and Bostock use to develop a domain specific visualization language that they argue makes it simpler for designers to construct graphics without sacrificing expressivity [29]. Similarly, the model presented in this work formulates visualization as equivariant maps from data space to visual space, and is designed such that developers can build software libraries with data and graphic topologies tuned to specific domains.

## 5.2 Future Work

While the model and prototype demonstrate that generation of simple marks from the data, there is a lot of work left to develop a model that underpins a minimally viable library. Foremost is implementing a data object that encodes data with a 2D continuous topology and an artist that can consume data with a 2D topology to visualize the image[89–91] and also encoding a separate heatmap[92, 93] artist that consumes 1D discrete data. A second important proof of concept artist is a boxplot[94] because it is a graphic that assumes computation on the data side and the glyph is built from semantically defined components and a list of outliers. The model supports simple composition of glyphs by overlaying glyphs at the same position, but more work is needed to define an operator where the fiber bundles have shared  $S_2 \hookrightarrow S_1$  such that fibers could be pulled back over the subset. While the model’s simple addition supports axes as standalone artists with overlapping visual position encoding, the complex operator would allow for binding together data that needs to be updated together. Additionally, implementing the complex addition operator and explicit graphic to data maps would allow for developing a mathematical formalism and prototype

802 of how interactivity would work in this model. In summary, the proposed scope of work for  
803 the dissertation is

- 804 • expansion of the mathematical framework to include complex addition
- 805 • formalization of definition of equivalence class  $A'$
- 806 • implementation of artist with explicit  $\xi$
- 807 • specification of interactive visualization
- 808 • mathematical formulation of a graphic with axes labeling
- 809 • implementation of new prototype artists that do not inherit from Matplotlib artists
- 810 • provisional mathematics and implementation of user level composite artists
- 811 • proof of concept domain specific user facing library

812 Other potential tasks for future work is implementing a data object for a non-trivial fiber  
813 bundle and exploiting the models section level formalism to build distributed data source  
814 models and concurrent artists. This could be pushed further to integrate with topological[95]  
815 and functional [96] data analysis methods. Since this model formalizes notions of structure  
816 preservation, it can serve as a good base for tools that assess quality metrics[97] or invariance  
817 [14] of visualizations with respect to graphical encoding choices. While this paper formulates  
818 visualization in terms of monoidal action homomorphisms between fiberbundles, the model  
819 lends itself to a categorical formulation[98, 99] that could be further explored.

## 820 6 Conclusion

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

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

## 829 References

- 830 [1] Krist Wongsuphasawat. *Navigating the Wide World of Data Visualization Libraries*  
831 *(on the Web)*. 2021.
- 832 [2] John W. Tukey. “We Need Both Exploratory and Confirmatory”. In: *The American*  
833 *Statistician* 34.1 (Feb. 1980), pp. 23–25. ISSN: 0003-1305. DOI: [10.1080/00031305.](https://doi.org/10.1080/00031305.1980.10482706)  
834 [1980.10482706](https://doi.org/10.1080/00031305.1980.10482706).
- 835 [3] Z. Pousman, J. Stasko, and M. Mateas. “Casual Information Visualization: Depictions  
836 of Data in Everyday Life”. In: *IEEE Transactions on Visualization and Computer*  
837 *Graphics* 13.6 (Nov. 2007), pp. 1145–1152. ISSN: 1941-0506. DOI: [10.1109/TVCG.](https://doi.org/10.1109/TVCG.2007.70541)  
838 [2007.70541](https://doi.org/10.1109/TVCG.2007.70541).
- 839 [4] J. Hughes. “Why Functional Programming Matters”. In: *The Computer Journal* 32.2  
840 (Jan. 1989), pp. 98–107. ISSN: 0010-4620. DOI: [10.1093/comjnl/32.2.98](https://doi.org/10.1093/comjnl/32.2.98).
- 841 [5] Zhenjiang Hu, John Hughes, and Meng Wang. “How Functional Programming Mat-  
842 tered”. In: *National Science Review* 2.3 (Sept. 2015), pp. 349–370. ISSN: 2095-5138.  
843 DOI: [10.1093/nsr/nwv042](https://doi.org/10.1093/nsr/nwv042).
- 844 [6] J. D. Hunter. “Matplotlib: A 2D Graphics Environment”. In: *Computing in Science*  
845 *Engineering* 9.3 (May 2007), pp. 90–95. ISSN: 1558-366X. DOI: [10.1109/MCSE.2007.](https://doi.org/10.1109/MCSE.2007.55)  
846 [55](https://doi.org/10.1109/MCSE.2007.55).
- 847 [7] John Hunter and Michael Droettboom. *The Architecture of Open Source Applications*  
848 *(Volume 2): Matplotlib*. <https://www.aosabook.org/en/matplotlib.html>.
- 849 [8] A. Sarikaya et al. “What Do We Talk About When We Talk About Dashboards?”  
850 In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (Jan. 2019),  
851 pp. 682–692. ISSN: 1941-0506. DOI: [10.1109/TVCG.2018.2864903](https://doi.org/10.1109/TVCG.2018.2864903).



- [9] Michael Friendly. “A Brief History of Data Visualization”. en. In: *Handbook of Data Visualization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 15–56. ISBN: 978-3-540-33036-3 978-3-540-33037-0. DOI: [10.1007/978-3-540-33037-0\\_2](https://doi.org/10.1007/978-3-540-33037-0_2).
- [10] L. Byrne, D. Angus, and J. Wiles. “Acquired Codes of Meaning in Data Visualization and Infographics: Beyond Perceptual Primitives”. In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (Jan. 2016), pp. 509–518. ISSN: 1077-2626. DOI: [10.1109/TVCG.2015.2467321](https://doi.org/10.1109/TVCG.2015.2467321).
- [11] Jacques Bertin. *Semiology of Graphics : Diagrams, Networks, Maps*. English. Redlands, Calif.: ESRI Press, 2011. ISBN: 978-1-58948-261-6 1-58948-261-1.
- [12] Jock Mackinlay. “Automating the Design of Graphical Presentations of Relational Information”. In: *ACM Transactions on Graphics* 5.2 (Apr. 1986), pp. 110–141. ISSN: 0730-0301. DOI: [10.1145/22949.22950](https://doi.org/10.1145/22949.22950).
- [13] Jock Mackinlay. “Automatic Design of Graphical Presentations”. English. PhD Thesis. Stanford, 1987.
- [14] G. Kindlmann and C. Scheidegger. “An Algebraic Process for Visualization Design”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (Dec. 2014), pp. 2181–2190. ISSN: 1941-0506. DOI: [10.1109/TVCG.2014.2346325](https://doi.org/10.1109/TVCG.2014.2346325).
- [15] Ricky Shadrach. *Introduction to Groups*. <https://www.mathsisfun.com/sets/groups-introduction.html>. 2017.
- [16] William S. Cleveland. “Research in Statistical Graphics”. In: *Journal of the American Statistical Association* 82.398 (June 1987), p. 419. ISSN: 01621459. DOI: [10.2307/2289443](https://doi.org/10.2307/2289443).
- [17] William S. Cleveland and Robert McGill. “Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods”. In: *Journal of the American Statistical Association* 79.387 (Sept. 1984), pp. 531–554. ISSN: 0162-1459. DOI: [10.1080/01621459.1984.10478080](https://doi.org/10.1080/01621459.1984.10478080).
- [18] John M Chambers et al. *Graphical Methods for Data Analysis*. Vol. 5. Wadsworth Belmont, CA, 1983.

- [19] Tamara Munzner. *Visualization Analysis and Design*. AK Peters Visualization Series. CRC press, Oct. 2014. ISBN: 978-1-4665-0891-0.
- [20] *Naturalness Principle - InfoVis:Wiki*. [https://infovis-wiki.net/wiki/Naturalness\\_Principle](https://infovis-wiki.net/wiki/Naturalness_Principle).
- [21] Edward R. Tufte. *The Visual Display of Quantitative Information*. English. Cheshire, Conn.: Graphics Press, 2001. ISBN: 0-9613921-4-2 978-0-9613921-4-7 978-1-930824-13-3 1-930824-13-0.
- [22] J. Heer and M. Agrawala. “Software Design Patterns for Information Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (2006), pp. 853–860. DOI: [10.1109/TVCG.2006.178](https://doi.org/10.1109/TVCG.2006.178).
- [23] Leland Wilkinson. *The Grammar of Graphics*. en. 2nd ed. Statistics and Computing. New York: Springer-Verlag New York, Inc., 2005. ISBN: 978-0-387-24544-7.
- [24] Hadley Wickham. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN: 978-3-319-24277-4.
- [25] M. Bostock and J. Heer. “Protovis: A Graphical Toolkit for Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1121–1128. ISSN: 1941-0506. DOI: [10.1109/TVCG.2009.174](https://doi.org/10.1109/TVCG.2009.174).
- [26] M. Bostock, V. Ogievetsky, and J. Heer. “D<sup>3</sup> Data-Driven Documents”. In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pp. 2301–2309. ISSN: 1941-0506. DOI: [10.1109/TVCG.2011.185](https://doi.org/10.1109/TVCG.2011.185).
- [27] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. “Declarative Interaction Design for Data Visualization”. en. In: *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*. Honolulu Hawaii USA: ACM, Oct. 2014, pp. 669–678. ISBN: 978-1-4503-3069-5. DOI: [10.1145/2642918.2647360](https://doi.org/10.1145/2642918.2647360).
- [28] Jacob VanderPlas et al. “Altair: Interactive Statistical Visualizations for Python”. en. In: *Journal of Open Source Software* 3.32 (Dec. 2018), p. 1057. ISSN: 2475-9066. DOI: [10.21105/joss.01057](https://doi.org/10.21105/joss.01057).

- [29] Jeffrey Heer and Michael Bostock. “Declarative Language Design for Interactive Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.6 (Nov. 2010), pp. 1149–1156. ISSN: 1077-2626. DOI: [10.1109/TVCG.2010.144](https://doi.org/10.1109/TVCG.2010.144).
- [30] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. “NIH Image to ImageJ: 25 Years of Image Analysis”. In: *Nature Methods* 9.7 (July 2012), pp. 671–675. ISSN: 1548-7105. DOI: [10.1038/nmeth.2089](https://doi.org/10.1038/nmeth.2089).
- [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).
- [32] *Writing Plugins*. en. [https://imagej.net/Writing\\_plugins](https://imagej.net/Writing_plugins).
- [33] Software Studies. *Culturevis/Imageplot*. Jan. 2021.
- [34] *Data Representation in Mayavi — Mayavi 4.7.2 Documentation*. <https://docs.enthought.com/mayavi/mayavi/data-representation>.
- [35] Marcus D. Hanwell et al. “The Visualization Toolkit (VTK): Rewriting the Rendering Code for Modern Graphics Cards”. en. In: *SoftwareX* 1-2 (Sept. 2015), pp. 9–12. ISSN: 23527110. DOI: [10.1016/j.softx.2015.04.001](https://doi.org/10.1016/j.softx.2015.04.001).
- [36] Berk Geveci et al. “VTK”. In: *The Architecture of Open Source Applications* 1 (2012), pp. 387–402.
- [37] P. Ramachandran and G. Varoquaux. “Mayavi: 3D Visualization of Scientific Data”. In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 40–51. ISSN: 1558-366X. DOI: [10.1109/MCSE.2011.35](https://doi.org/10.1109/MCSE.2011.35).
- [38] James Ahrens, Berk Geveci, and Charles Law. “Paraview: An End-User Tool for Large Data Visualization”. In: *The visualization handbook* 717.8 (2005).
- [39] Brian Wylie and Jeffrey Baumes. “A Unified Toolkit for Information and Scientific Visualization”. In: *Proc.SPIE*. Vol. 7243. Jan. 2009. DOI: [10.1117/12.805589](https://doi.org/10.1117/12.805589).
- [40] Stephen Few and Perceptual Edge. “Dashboard Confusion Revisited”. In: *Perceptual Edge* (2007), pp. 1–6.

- [41] D. M. Butler and M. H. Pendley. “A Visualization Model Based on the Mathematics of Fiber Bundles”. en. In: *Computers in Physics* 3.5 (1989), p. 45. ISSN: 08941866. DOI: [10.1063/1.168345](https://doi.org/10.1063/1.168345).
- [42] David M. Butler and Steve Bryson. “Vector-Bundle Classes Form Powerful Tool for Scientific Visualization”. en. In: *Computers in Physics* 6.6 (1992), p. 576. ISSN: 08941866. DOI: [10.1063/1.4823118](https://doi.org/10.1063/1.4823118).
- [43] David I Spivak. *Databases Are Categories*. en. Slides. June 2010.
- [44] David I Spivak. “SIMPLICIAL DATABASES”. en. In: (), p. 35.
- [45] Tamara Munzner. “Ch 2: Data Abstraction”. In: *CPSC547: Information Visualization, Fall 2015-2016* ().
- [46] A. M. Cegarra. “Cohomology of Monoids with Operators”. In: *Semigroup Forum*. Vol. 99. Springer, 2019, pp. 67–105. ISBN: 1432-2137.
- [47] E.H. Spanier. *Algebraic Topology*. McGraw-Hill Series in Higher Mathematics. Springer, 1989. ISBN: 978-0-387-94426-5.
- [48] *Locally Trivial Fibre Bundle - Encyclopedia of Mathematics*. [https://encyclopediaofmath.org/wiki/Locally\\_trivial](https://encyclopediaofmath.org/wiki/Locally_trivial)
- [49] S. S. Stevens. “On the Theory of Scales of Measurement”. In: *Science* 103.2684 (1946), pp. 677–680. ISSN: 00368075, 10959203.
- [50] Brent A Yorgey. “Monoids: Theme and Variations (Functional Pearl)”. en. In: (), p. 12.
- [51] Michele Stieven. *A Monad Is Just a Monoid...* en. <https://medium.com/@michelestieven/a-monad-is-just-a-monoid-a02bd2524f66>. Apr. 2020.
- [52] “Monoid”. en. In: *Wikipedia* (Jan. 2021).
- [53] “Semigroup Action”. en. In: *Wikipedia* (Jan. 2021).
- [54] nLab authors. “Action”. In: (Mar. 2021).
- [55] W A Lea. “A Formalization of Measurement Scale Forms”. en. In: (), p. 44.
- [56] Eric W. Weisstein. *Similarity Transformation*. en. <https://mathworld.wolfram.com/SimilarityTransformation.html>. Text.

- [57] “Quotient Space (Topology)”. en. In: *Wikipedia* (Nov. 2020).
- [58] Professor Denis Auroux. “Math 131: Introduction to Topology”. en. In: (), p. 113.
- [59] Charles R Harris et al. “Array Programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362.
- [60] Jeff Reback et al. *Pandas-Dev/Pandas: Pandas 1.0.3*. Zenodo. Mar. 2020. DOI: [10.5281/zenodo.3715232](https://doi.org/10.5281/zenodo.3715232).
- [61] Stephan Hoyer and Joe Hamman. “Xarray: ND Labeled Arrays and Datasets in Python”. In: *Journal of Open Research Software* 5.1 (2017).
- [62] Matthew Rocklin. “Dask: Parallel Computation with Blocked Algorithms and Task Scheduling”. In: *Proceedings of the 14th Python in Science Conference*. Vol. 126. Cite-seer, 2015.
- [63] “Retraction (Topology)”. en. In: *Wikipedia* (July 2020).
- [64] Tim Bienz, Richard Cohn, and Calif.) Adobe Systems (Mountain View. *Portable Document Format Reference Manual*. Citeseer, 1993.
- [65] A. Quint. “Scalable Vector Graphics”. In: *IEEE MultiMedia* 10.3 (July 2003), pp. 99–102. ISSN: 1941-0166. DOI: [10.1109/MMUL.2003.1218261](https://doi.org/10.1109/MMUL.2003.1218261).
- [66] George S. Carson. “Standards Pipeline: The OpenGL Specification”. In: *SIGGRAPH Comput. Graph.* 31.2 (May 1997), pp. 17–18. ISSN: 0097-8930. DOI: [10.1145/271283.271292](https://doi.org/10.1145/271283.271292).
- [67] *Cairographics.Org*. <https://www.cairographics.org/>.
- [68] “Jet Bundle”. en. In: *Wikipedia* (Dec. 2020).
- [69] Jana Musilová and Stanislav Hronek. “The Calculus of Variations on Jet Bundles as a Universal Approach for a Variational Formulation of Fundamental Physical Theories”. In: *Communications in Mathematics* 24.2 (Dec. 2016), pp. 173–193. ISSN: 2336-1298. DOI: [10.1515/cm-2016-0012](https://doi.org/10.1515/cm-2016-0012).
- [70] “Connected Space”. en. In: *Wikipedia* (Dec. 2020).

- [71] Caroline Ziemkiewicz and Robert Kosara. “Embedding Information Visualization within Visual Representation”. In: *Advances in Information and Intelligent Systems*. Ed. by Zbigniew W. Ras and William Ribarsky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 307–326. ISBN: 978-3-642-04141-9. DOI: [10.1007/978-3-642-04141-9\\_15](https://doi.org/10.1007/978-3-642-04141-9_15).
- [72] Sheelagh Carpendale. *Visual Representation from Semiology of Graphics by J. Bertin*. en.
- [73] Robert W. Ghrist. *Elementary Applied Topology*. Vol. 1. Createspace Seattle, 2014.
- [74] Robert Ghrist. “Homological Algebra and Data”. In: *Math. Data* 25 (2018), p. 273.
- [75] Michael S. Crouch, Andrew McGregor, and Daniel Stubbs. “Dynamic Graphs in the Sliding-Window Model”. In: *European Symposium on Algorithms*. Springer, 2013, pp. 337–348.
- [76] Chia-Shang James Chu. “Time Series Segmentation: A Sliding Window Approach”. In: *Information Sciences* 85.1 (July 1995), pp. 147–173. ISSN: 0020-0255. DOI: [10.1016/0020-0255\(95\)00021-G](https://doi.org/10.1016/0020-0255(95)00021-G).
- [77] Dmitry Nekrasovski et al. “An Evaluation of Pan & Zoom and Rubber Sheet Navigation with and without an Overview”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 11–20. ISBN: 1-59593-372-7. DOI: [10.1145/1124772.1124775](https://doi.org/10.1145/1124772.1124775).
- [78] Yael Albo et al. “Off the Radar: Comparative Evaluation of Radial Visualization Solutions for Composite Indicators”. In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (Jan. 2016), pp. 569–578. ISSN: 1077-2626. DOI: [10.1109/TVCG.2015.2467322](https://doi.org/10.1109/TVCG.2015.2467322).
- [79] Z. Qu and J. Hullman. “Keeping Multiple Views Consistent: Constraints, Validations, and Exceptions in Visualization Authoring”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (Jan. 2018), pp. 468–477. ISSN: 1941-0506. DOI: [10.1109/TVCG.2017.2744198](https://doi.org/10.1109/TVCG.2017.2744198).

- [80] Richard A. Becker and William S. Cleveland. “Brushing Scatterplots”. In: *Technometrics* 29.2 (May 1987), pp. 127–142. ISSN: 0040-1706. DOI: [10.1080/00401706.1987.10488204](https://doi.org/10.1080/00401706.1987.10488204).
- [81] Andreas Buja et al. “Interactive Data Visualization Using Focusing and Linking”. In: *Proceedings of the 2nd Conference on Visualization '91*. VIS '91. Washington, DC, USA: IEEE Computer Society Press, 1991, pp. 156–163. ISBN: 0-8186-2245-8.
- [82] *Dataclasses — Data Classes — Python 3.9.2rc1 Documentation*. [https://docs.python.org/3/library/dataclasses.h](https://docs.python.org/3/library/dataclasses.html)
- [83] Kristen B. Gorman, Tony D. Williams, and William R. Fraser. “Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus *Pygoscelis*)”. In: *PLOS ONE* 9.3 (Mar. 2014), e90081. DOI: [10.1371/journal.pone.0090081](https://doi.org/10.1371/journal.pone.0090081).
- [84] Allison Marie Horst, Alison Presmanes Hill, and Kristen B Gorman. *Palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. Manual. 2020. DOI: [10.5281/zenodo.3960218](https://doi.org/10.5281/zenodo.3960218).
- [85] Muhammad Chenariyan Nakhaee. *Mcnakhaee/Palmerpenguins*. Jan. 2021.
- [86] F. Beck. “Software Feathers Figurative Visualization of Software Metrics”. In: *2014 International Conference on Information Visualization Theory and Applications (IVAPP)*. Jan. 2014, pp. 5–16.
- [87] Lydia Byrne, Daniel Angus, and Janet Wiles. “Figurative Frames: A Critical Vocabulary for Images in Information Visualization”. In: *Information Visualization* 18.1 (Aug. 2017), pp. 45–67. ISSN: 1473-8716. DOI: [10.1177/1473871617724212](https://doi.org/10.1177/1473871617724212).
- [88] Kenneth C. Loudon. *Programming Languages : Principles and Practice*. English. Pacific Grove, Calif: Brooks/Cole, 2010. ISBN: 978-0-534-95341-6 0-534-95341-7.
- [89] M. Tory and T. Moller. “Rethinking Visualization: A High-Level Taxonomy”. In: *IEEE Symposium on Information Visualization*. 2004, pp. 151–158. DOI: [10.1109/INFVIS.2004.59](https://doi.org/10.1109/INFVIS.2004.59).

- [90] Robert B Haber and David A McNabb. “Visualization Idioms: A Conceptual Model for Scientific Visualization Systems”. In: *Visualization in scientific computing* 74 (1990), p. 93.
- [91] Charles D Hansen and Chris R Johnson. *Visualization Handbook*. Elsevier, 2011.
- [92] Leland Wilkinson and Michael Friendly. “The History of the Cluster Heat Map”. In: *The American Statistician* 63.2 (May 2009), pp. 179–184. ISSN: 0003-1305. DOI: [10.1198/tas.2009.0033](https://doi.org/10.1198/tas.2009.0033).
- [93] Toussaint Loua. *Atlas Statistique de La Population de Paris*. J. Dejeu & cie, 1873.
- [94] Hadley Wickham and Lisa Stryjewski. “40 Years of Boxplots”. In: *The American Statistician* (2011).
- [95] C. Heine et al. “A Survey of Topology-Based Methods in Visualization”. In: *Computer Graphics Forum* 35.3 (June 2016), pp. 643–667. ISSN: 0167-7055. DOI: [10.1111/cgf.12933](https://doi.org/10.1111/cgf.12933).
- [96] James O Ramsay. *Functional Data Analysis*. Wiley Online Library, 2006.
- [97] Enrico Bertini, Andrada Tatu, and Daniel Keim. “Quality Metrics in High-Dimensional Data Visualization: An Overview and Systematization”. In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (2011), pp. 2203–2212.
- [98] Brendan Fong and David I. Spivak. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. en. First. Cambridge University Press, July 2019. ISBN: 978-1-108-66880-4 978-1-108-48229-5 978-1-108-71182-1. DOI: [10.1017/9781108668804](https://doi.org/10.1017/9781108668804).
- [99] Bartosz Milewski. “Category Theory for Programmers”. en. In: (), p. 498.