

1 1 Introduction

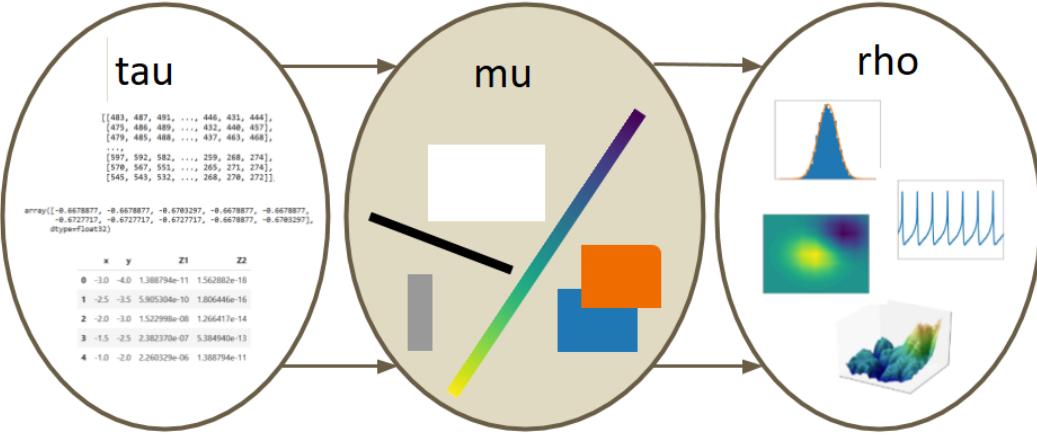


Figure 1: Visualization is equivariant maps between data and visual encoding of the variables and assembly of those encodings into a graphic. *will replace w/ overarching figure w/ same structure*

2 The work presented in this paper is motivated by a need for a library of visualization
 3 components that developers could use to build complex, domain specific tools tuned to
 4 the semantics and structure carried in domain specific data. While many researchers have
 5 identified and described important aspects of visualization, they have specialized in such
 6 different ways as to not provide a model general enough to natively support the full range
 7 of data and visualization types many general purpose modern visualization tools may need
 8 to support. The core architecture also needs to be robust to the big data needs of many
 9 visualization practitioners, and therefore support distributed and streaming data needs. To
 10 support both exploratory and confirmatory visualization[94], this tool needs to support 2D
 11 and 3D, static, dynamic and interactive visualizations.

12 Specifically, this work was driven by a rearchitecture of the Python visualization li-
 13 brary Matplotlib[53] to meet modern data visualization needs. We aim to take advantage
 14 of developments in software design, data structures, and visualization to improve the con-
 15 sistency, compositability, and discoverability of the API. To do so, this work first presents a
 16 mathematical description of how data is transformed into graphic representations, as shown
 17 in figure 1. As with other mathematical formalisms of visualization [56, 62, 91, 96], a
 18 mathematical framework provides a way to formalize the properties and structure of the
 19 visualization. In contrast to the other formalisms, the model presented here is focused on
 20 the components that build a visualization rather than the visualization itself.

21 In other words this model is not intended to be evaluative, it is intended to be a reference
 22 specification for visualization library API. To make this model as implementation indepen-
 23 dent as possible, we propose fairly general mathematical abstractions of the data container
 24 such that we do not need to assume the data has any specific structure, such as a relational
 25 database. We reuse this structure for the graphic as that allows us to specifically discuss
 26 how structure is preserved. We take a functional approach because functional paradigms
 27 encourage writing APIs that are flexible, concise and predictable due to the lack of side
 28 effects [61]. Furthermore, by structuring the API in terms of composition of the smallest

29 units of transformation for which we can define correctness, a functional paradigm naturally
30 leads to a library of highly modular components that are composable in such a way that
31 by definition the composition is also correct. This allows us to ensure that domain specific
32 visualizations built on top of these components are also correct without needing knowl-
33 edge of the domain. As with the other mathematical formalisms of visualization, we factor
34 out the rendering into a separate stage; but, our framework describes how these rendering
35 instructions are generated.

36 In this work, we present a framework for understanding visualization as equivariant maps
37 between topological spaces. Using this mathematical formalism, we can interpret and extend
38 prior work and also develop new tools. We validate our model by using it to re-design artist
39 and data access layer of Matplotlib, a general purpose visualization tool.

40 2 Background

41 One of the reasons we developed a new formalism rather than adopting the architecture
42 of an existing library is that most information visualization software design patterns,
43 as categorized by Heer and Agrawala[47], are tuned to very specific data structures.
44 This in turn restricts the design space of visual algorithms that display information
45 (the visualization types the library supports) since the algorithms are designed such
46 that the structure of data is assumed, as described in Tory and Möller's taxonomy
47 [**toryRethinkingVisualizationHighLevel2004**]. In proposing a new architecture, we
48 contrast the trade offs libraries make, describe different types of data continuity, and discuss
49 metrics by which a visualization library is traditionally evaluated.

50 2.1 Tools

51 One extensive family of relational table based libraries are those based on Wilkenson's
52 Grammar of Graphics (GoG) [101], including ggplot[99], protovis[14] and D3 [15], vega[81]
53 and altair[95]. The restriction to tables in turn restricts the native design space to visu-
54 alizations suited to tables. Since the data space and graphic space is very well defined in
55 this grammar, it lends itself to a declarative interface [48]. This grammar oriented approach
56 allows users to describe how to compose visual elements into a graphical design [103], while
57 we are proposing a framework for building those elements. An example of this distinction
58 is that the GoG grammar includes computation and aggregation of the table as part of the
59 grammar, while we propose that most computations are specific to domains and only try to
60 describe them when they are specifically part of the visual encoding - for example mapping
61 data to a color. Disentangling the computation from the visual transforms allows us to
62 determine whether the visualization library needs to handle them or if they can be more
63 efficiently computed by the data container.

64 A different class of user facing tools are those that support images, such as ImageJ[82]
65 or Napari[84]. These tools often have some support for visualizing non image components
66 of a complex data set, but mostly in service to the image being visualized. These tools
67 are ill suited for general purpose libraries that need to support data other than images
68 because the architecture is oriented towards building plugins into the existing system [104]
69 where the image is the core data structure. Even the digital humanities oriented ImageJ
70 macro ImagePlot[90], which supports some non-image aggregate reporting charts, is still
71 built around image data as the primary input.

72 There are also visualization tools where there is no single core structure, and instead in-
73 ternally carry around many different representations of data. Matplotlib, has this structure,
74 as does VTK [39, 45] and its derivatives such as MayaVi[**ramachandranMayaVi2011**]
75 and extensions such as ParaView[5] and the infoviz themed Titan[16]. Where GoG and
76 ImageJ type libraries have very consistent APIs for their visualization tools because the
77 data structure is the same, the APIs for visualizations in VTK and Matplotlib are signif-
78 icantly dependent on the structure of the data it expects. This in turn means that every
79 new type of visualization must carry implicit assumptions about data structure in how it
80 interfaces with the input data. This has lead to poor API consistency and brittle code as
81 every visualization type has a very different point of view on how the data is structured.
82 This API choice particularly breaks down when the same dataset is fed into visualizations
83 with different assumptions about structure or into a dashboard consisting of different types
84 of visualization[1, 36] because there is no consistent way to update the data and therefore
85 no consistent way of guaranteeing that the views stay in sync. Our model is a structure
86 dependent formalism, but then also provides a core representation of that structure that is
87 abstract enough to provide a common interface for many different types of visualization.

88 **2.2 Data**

89 Discrete and continuous data and their attributes form a discipline independent design
90 space [73], so one of the drivers of this work was to facilitate building libraries that could
91 natively support domain specific data containers that do not make assumptions about data
92 continuity. As shown in figure 2, there are many types of connectivity. A database typically
93 consists of unconnected records, while an image is an implicit 2D grid and a network is
94 some sort of explicitly connected graph. These data structures typically contain not only
95 the measurements or values of the data, but also domain specific semantic information such
96 as that the data is a map or an image that a modern visualization library could exploit if
97 this information was exposed to the API.

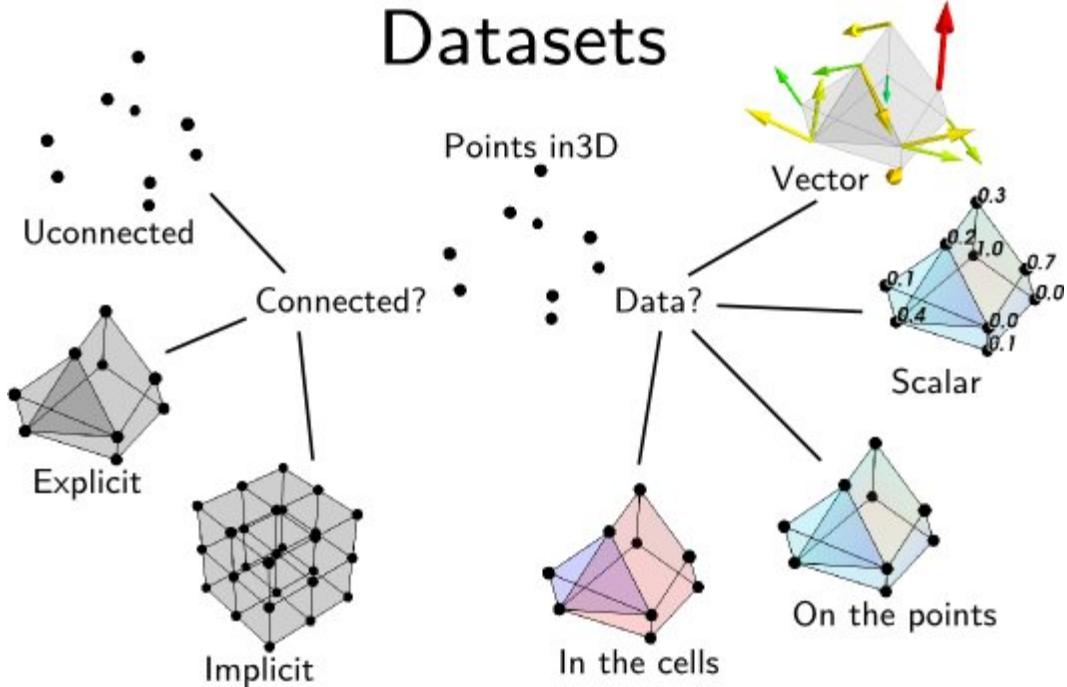


Figure 2: One way to describe data is by the connectivity of the points in the dataset. A database for example is often discrete unconnected points, while an image is an implicitly connected 2D grid. This image is from the Data Representation chapter of the MayaVi 4.7.2 documentation.[32]

As shown in figure 2, there are many distinct ways of encoding each specific type of structure, while as mentioned in section 2.1 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 [18, 19]. Since Butler's model lacks a robust way of describing variables, we fold in Spivak's Simplicial formulation of databases [86, 87] so that we can encode a schema like description of the data in the fiber bundle. In this work we will refer to the points of the dataset as *records* to indicate that a point can be a vector of heterogenous elements. Each *component* of the record is a single object, such as a temperature measurement, a color value, or an image. We also generalize *component* to mean all objects in the dataset of a given type, such as all temperatures or colors or images. The way in which these records are connected is the *connectivity*, *continuity*, or more generally *topology*.

definitions

records points, observations, entries

components variables, attributes, fields

connectivity how the records are connected to each other

Often this topology has metadata associated with it, describing for example when or where the measurement was taken. Building on the idea of metadata as *keys* and their associated *value* proposed by Munzner [67], we propose that information rich metadata are part of the components and instead the values are keyed on coordinate free structural ids. In contrast to Munzner’s model where the semantic meaning of the key is tightly coupled to the position of the value in the dataset, our model allows for renaming all the metadata, for example changing the coordinate systems or time resolution, without imposing new semantics on the underlying structure.

2.3 Visualization

	<i>Points</i>	<i>Lines</i>	<i>Areas</i>	<i>Best to show</i>
<i>Shape</i>		<i>possible, but too weird to show</i>	<i>cartogram</i>	<i>qualitative differences</i>
<i>Size</i>			<i>cartogram</i>	<i>quantitative differences</i>
<i>Color Hue</i>				<i>qualitative differences</i>
<i>Color Value</i>				<i>quantitative differences</i>
<i>Color Intensity</i>				<i>qualitative differences</i>
<i>Texture</i>				<i>qualitative & quantitative differences</i>

Figure 3: Retinal variables are a codification of how position, size, shape, color and texture are used to illustrate variations in the components of a visualization. The best to show column describes which types of information can be expressed in the corresponding visual encoding. This tabular form of Bertin’s retinal variables is from Understanding Graphics [64] who reproduced it from Krygier and Wood’s *Making Maps: A Visual Guide to Map Design for GIS* [57]

120 Visual representations of data, by definition, reflect something of the underlying structure
121 and semantics[38], whether through direct mappings from data into visual elements or via
122 figurative representations that have meaning due to their similarity in shape to external
123 concepts [20]. The components of a visual representation were first codified by Bertin[11].
124 As illustrated in figure 3, Bertin proposes that there are classes of visual encodings such as
125 shape, color, and texture that when mapped to from specific types of measurement, quanti-
126 tative or qualitative, will preserve the properties of that measurement type. For example,
127 that nominal data mapped to hue preserves the selectivity of the nominal measurements.
128 Furthermore he proposes that the visual encodings be composited into graphical marks that
129 match the connectivity of the data - for example discrete data is a point, 1D continuous
130 is the line, and 2D data is the area mark. A general form of marks are glyphs, which are
131 graphical objects that convey one or more attributes of the data entity mapped to it[68, 97]
132 and minimally need to be differentiable from other visual elements [106]. The set of encod-
133 ing relations from data to visual representation is termed the graphical design by Mackinlay
134 [62, 63] and the design rendered in an idealized abstract space is what throughout this paper
135 we will refer to as a graphic.

136 The measure of how much of the structure of the data the graphic encodes is a con-
137 cept Mackinlay termed expressiveness, while the graphic's effectiveness describes how much
138 design choices are made in deference to perceptual saliency [26, 28, 29, 68]. When the
139 properties of the representation match the properties of the data, then the visualization
140 is easier to understand according to Norman's Naturalness Principal[71]. These ideas are
141 combined into Tufte's notion of graphical integrity, which is that a visual representation
142 of quantitative data must be directly proportional to the numerical quantities it represents
143 (Lie Principal), must have the same number of visual dimensions as the data, and should
144 be well labeled and contextualized, and not have any extraneous visual elements [93]. This
145 notion of matching is explicitly formalized by Mackinlay as a structure preserving mapping
146 of a binary operator from one domain to another [63]. A functional dependency framework
147 for evaluating visualizations was proposed by Sugibuchi et al [91], and an algebraic basis
148 for visualization design and evaluation was proposed by Kindlmann and Scheidegger[56].
149 Vickers et al. propose a category theory framework[96] that extends structural preservation
150 to layout, but is focused strictly on the design layer like the other mathematical frameworks.

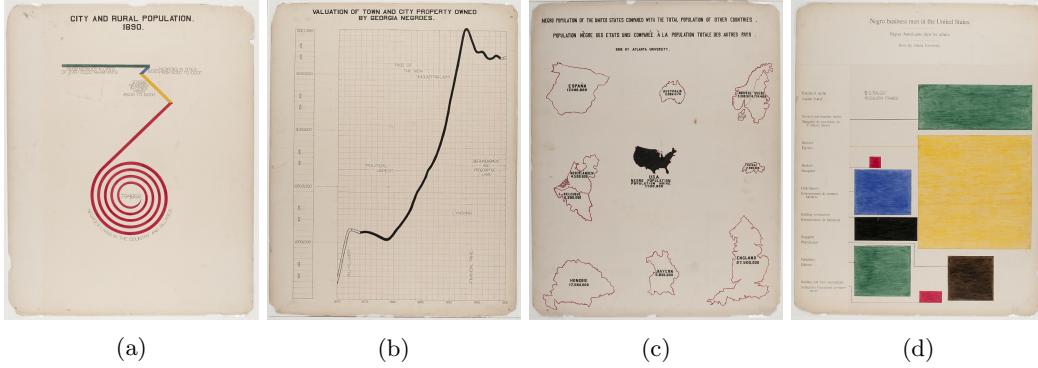


Figure 4: Du Bois' data portraits[35] of post reconstruction Black American life exemplify that the fundamental characteristics of data visualization is that the visual elements vary in proportion to the source data. In figure 4a, the length of each segment maps to population; in figure 4b, the line changes color to indicate a shift in the political environment; in figure 4c the countries are scaled to population size; and figure 4d is a treemap where the area of the rectangle is representative of the number of businesses in each field. The images here are from the Prints and Photographs collection of the Library of Congress [2, 3, 34, 92]

151 One example of highly expressive visualizations are the data portraits by Du Bois shown
 152 in figure 4. While the Du Bois charts are different from the usual scatter, line, and plot
 153 charts, they conform to the constraint that a graphic is a structure preserving map from
 154 data to visual representation. Figure 4a is semantically similar to a bar chart in that the
 155 lengths of the segments are mapped to the values, but in this chart the segments are stacked
 156 together. Figure 4b is a multicolored line chart where the color shifts are at periods of
 157 political significance. In figure 4c, Du Bois combines a graphical representation where glyph
 158 size varies by population with a figurative representation of those glyphs as the countries
 159 the data is from, which means that the semantic and numerical properties of the data are
 160 preserved in the graph. Figure 4b is simply a treemap[49] with space between the marks.
 161 Since the Du Bois data portraits meet the criteria of a faithful visual representation, we
 162 propose a mathematical framework and implementation that allows us to express the Du
 163 Bois charts and common chart types with equal fidelity.

164 2.4 Contribution

165 This work presents a mathematical model of the transformation from data to graphic rep-
 166 resentation and a proof of concept implementation. Specifically, this work contributes

- 167 1. a functional oriented visualization tool architecture
- 168 2. topology-preserving maps from data to graphic
- 169 3. monoidal action equivariant maps from component to visual variable
- 170 4. algebraic sum such that more complex visualizations can be built from simple ones
- 171 5. prototype built on Matplotlib's infrastructure

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

176 3 Topological Artist Model

As discussed in the introduction, visualization is generally defined as structure preserving maps from data to graphic representation. 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 $\varphi : M \rightarrow M'$ [25]. Monoid actions are a set of operations and 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 symmetric

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

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

179 We model the data \mathcal{E} , graphic \mathcal{H} , and intermediate visual encoding \mathcal{V} stages of visual-
180 ization as topological structures that encapsulate types of variables and continuity; by doing
181 so we can develop implementations that keep track of both in ways that let us distribute
182 computation while still allowing assembly and dynamic update of the graphic. To explain
183 which structure the artist is preserving, we first describe how we model data (3.1), graphics
184 (3.2), and intermediate visual characteristics (3.3) as fiber bundles. We then discuss the
185 equivariant maps between data and visual characteristics (3.3.2) and visual characteristics
186 and graphics (3.3.3) that make up the artist.

187 3.1 Data Space E

Building on Butler’s proposal of using fiber bundles as a common data representation format for visualization data[18, 19], a fiber bundle is a tuple (E, K, π, F) defined by the projection map π

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

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

192 By definition fiber bundles are locally trivial[59, 85], meaning that over a localized neigh-
193 borhood we can dispense with extra structure on E and focus on the components and conti-
194 nuity. We use fiber bundles as the data model because they are inclusive enough to express
195 all the types of data described in section 2.2.

¹⁹⁶ **3.1.1 Variables: Fiber Space F**

To formalize the structure of the data components, we use notation introduced by Spivak [87] that binds the components of the fiber to variable names and types. Spivak constructs a set \mathbb{U} that is the disjoint union of all possible objects of types $\{T_0, \dots, T_m\} \in \mathbf{DT}$, where \mathbf{DT} are the data types of the variables in the dataset. He then defines the single variable set \mathbb{U}_σ

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

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

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

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

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

which is equivalent to

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

¹⁹⁷ which allows us to decouple F into components F_i .

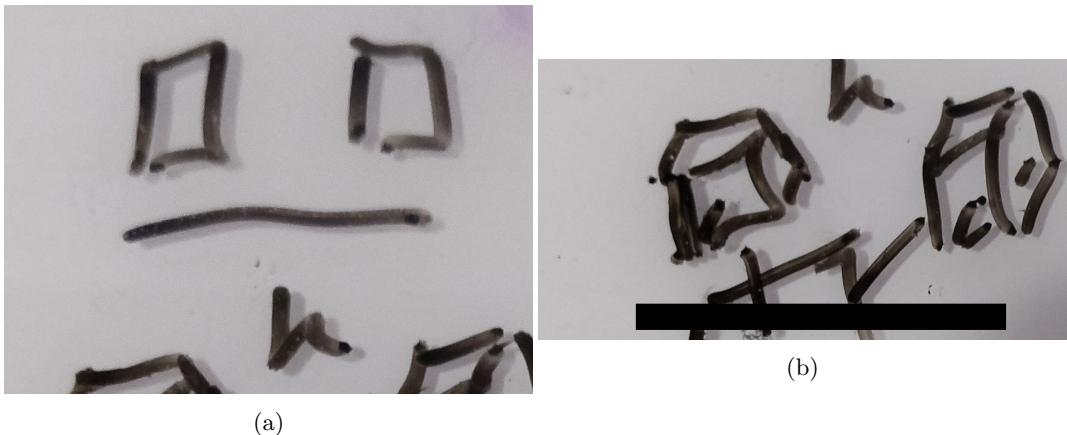


Figure 5: These two datasets have the same base space K but figure 5a has fiber $F = \mathbb{R} \times \mathbb{R}$ which is (time, temperature) while figure 5b has fiber $\mathbb{R}^+ \times \mathbb{R}^2$ which is (time, wind=(speed, direction))

For example, the data in figure 5a 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 5b, 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

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

such that F_1 is specified by ($c = \text{wind}$, $T = \text{wind}$, $\mathbb{U}_\sigma = \mathbb{R}^2$). As illustrated in figure 5, Spivak's framework provides a consistent way to describe potentially complex components of the input data.

3.1.2 Measurement Scales: Monoid Actions

Implementing expressive visual encodings requires formally describing the structure on the components of the fiber, which we define by the action of a monoid on the component. While structure on a set of values is often described algebraically as operations or through the actions of a group, for example Steven's scales [88], we generalize to monoids to support more component types. Monoids are also commonly found in functional programming because they specify compositions of transformations [89, 105].

A monoid [66] 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 [4, 83] 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

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

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

Steven's described the measurement scales[58, 88] in terms of the monoid actions on the measurements: nominal data is permutable, ordinal data is monotonic, interval data is translatable, and ratio data is scalable [98]. For example, given an arbitrary interval scale fiber component ($c = \text{temperature}$, $T = \text{float}$, $\mathbb{U}_\sigma = \mathbb{R}$) with arbitrarily chosen monoid translation actions actions

- monoid operator addition $* = +$
- monoid operations: $f : x \mapsto x + 1$, $g : x \mapsto x + 2$
- 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

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

218 where 1° and 2° are valid distances between two temperatures x . What this diagram means
 219 is that either the fiber could be shifted by 1 (vertical line) then by 2 (horizontal), or the
 220 two shifts could be combined such that in this case the fiber is shifted by 3 (diagonal) and
 221 these two paths yield the same temperature.

222 While many component types will be one of the measurement scale types, we gen-
 223 eralize to monoids specifically for the case of partially ordered set. Given a set $W =$
 224 $\{mist, drizzle, rain\}$, then the map $f : W \rightarrow W$ defined by

- 225 1. $f(rain) = drizzle,$
- 226 2. $f(drizzle) = mist$
- 227 3. $f(mist) = mist$

228 is order preserving such that $mist \leq drizzle \leq rain$ but has no inverse since $drizzle$ and
 229 $mist$ go to the same value $mist$. Therefore order preserving maps do not form a group, and
 230 instead we generalize to monoids to support partial order component types. Defining the
 231 monoid actions on the components serves as the basis for identifying the invariance[56] that
 232 must be preserved in the visual representation of the component. We propose equivariance
 233 of monoid actions individually on the fiber to visual component maps and on the graphic
 234 as a whole.

235 3.1.3 Continuity: Base Space K

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

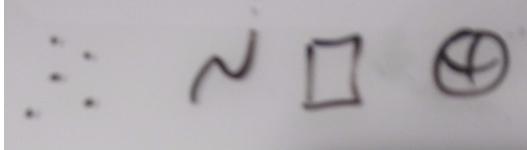


Figure 6: The topological base space K encodes the connectivity of the data space, for example if the data is independent points or a map or on a sphere

241 As illustrated in figure 6, K is akin to an indexing space into E that describes the
 242 structure of E . K can have any number of dimensions and can be continuous or discrete.

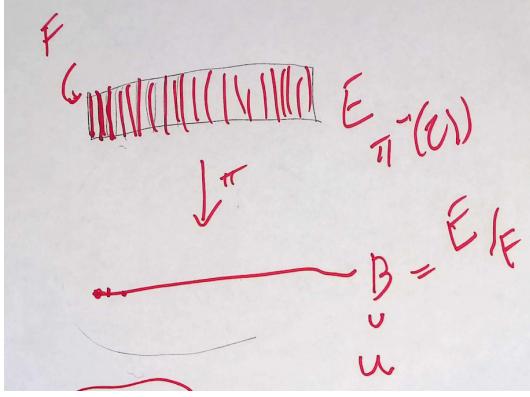


Figure 7: The base space E is divided into fiber segments F . The base space K acts as an index into the records in the fibers. **this figure might be good all the way up top to lay out the components of fb**

Formally K is the quotient space [76] of E meaning it is the finest space[8] such that every $k \in K$ has a corresponding fiber F_k [76]. In figure 7, 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

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

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

The datasets in figure 8 have the same fiber of (temperature, time). In figure 8a the fibers lie over discrete K such that the records in the datasets in the fiber bundles are discrete. The same fiber in figure 8b lies over a continuous interval K such that the records are samples from a continuous function defined on K . By encoding this continuity in the model as K the data model now explicitly carries information about its structure such that the implicit assumptions of the visualization algorithms are now explicit. The explicit topology is a concise way of distinguishing visualizations that appear identical, for example heatmaps and images.

3.1.4 Data: Sections τ

The section $\tau : K \rightarrow E$ is what ties together the base space K with the fiber F . A section is a function that takes as input location $k \in K$ and returns a record $r \in E$. For example, in the special case of a table [87], K is a set of row ids, F is the columns, and the section τ returns the record r at a given key in K . For any fiber bundle, there exists a map

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

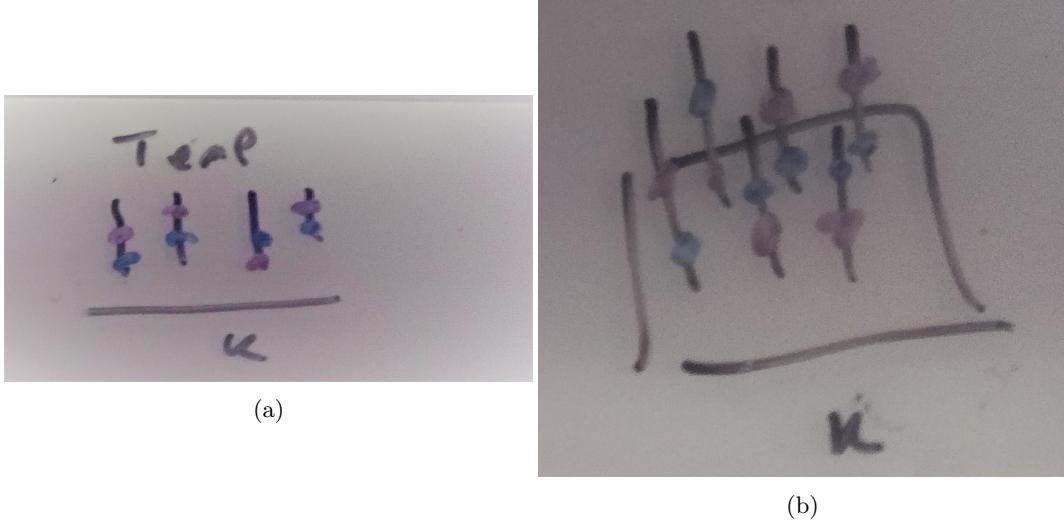


Figure 8: These two datasets have the same (time, temperature) fiber. In figure ?? the total space E is discrete over points $k \in K$, meaning the records in the fiber are also discrete. In figure ?? E lies over the continuous interval K , meaning the records in the fiber are sampled from a continuous space. *revamp figure: F=Plane, k1 = dots, k2=line*

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 τ as

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

254 where each section τ_i is a variable or set of variables. This allows for accessing the data
255 component wise rather than just as sections on K .

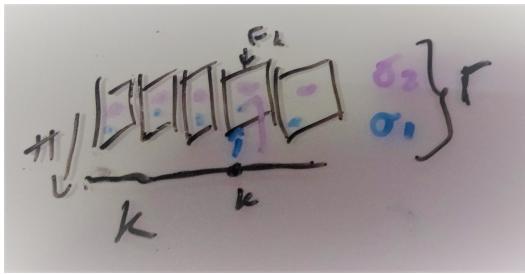


Figure 9: Fiber (time, temperature) with an interval K basespace. The sections τ_i and τ_j are constrained such that the time variable must be monotonic, which means each section is a timeseries of temperature values. They are included in the global set of sections $\tau_1, \tau_2 \in \Gamma(E)$

256 In the example in figure 9, the fiber is $(time, temperature)$ as described in figure 5 and
 257 the base space is the interval K . The section $\tau^{(1)}$ resolves to a series of monotonically
 258 increasing in time records of $(time, temperature)$ values. Section $\tau^{(2)}$ returns a different
 259 timeseries of $(time, temperature)$ values. Both sections are included in the global set of
 260 sections $\tau^{(1)}, \tau^{(2)} \in \Gamma(E)$.

261 This model provides a common interface to widely used data containers without sacri-
 262 ficing the semantic structure embedded in each container. For example, the section can be
 263 any instance of a univariate numpy array[46] that stores an image. This could be a section
 264 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
 265 color, and the other two components are the x and y positions of the sampled data in the
 266 image. Instead of an image, the numpy array could also store a 2D discrete table. The
 267 fiber would not change, but the K would now be 0D discrete points. These different choices
 268 in topology indicate, for example, what sorts of interpolation would be appropriate when
 269 visualizing the data.

270 There are also many types of labeled containers that can richly be described in this
 271 framework because of the fiber. For example, a pandas series which stores a labeled list,
 272 or a dataframe[78] which stores a relational table. A series could store the values of $\tau^{(1)}$
 273 and a second series could be $\tau^{(2)}$. We could also flatten the fiber to hold two temperature
 274 series, such that a section would be an instance of a dataframe with a time column and two
 275 temperature columns. While the series and dataframe explicitly have a time index column,
 276 they are components in our model and the index is always assumed to be random keys.

277 Where this model particularly shines are N dimensional labeled data structures. For
 278 example, an xarray[52] data that stores temperature field could have a K that is a continuous
 279 volume and the components would be the temperature and the time, latitude, and longitude
 280 the measurements were sampled at. A section can also be an instance of a distributed data
 281 container, such as a dask array [80]. As with the other containers, K and F are defined in
 282 terms of the index and dtypes of the components of the array. Because our framework is
 283 defined in terms of the fiber, continuity, and sections, rather than the exact values of the
 284 data, our model does not need to know what the exact values are until the renderer needs
 285 to fill in the image.

286 3.2 Graphic: H

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

292 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

$$(x, y, r, g, b) \in D \quad (16)$$

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



Figure 10: 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 . [add \$\alpha, \beta\$ coordinates to figures](#)

295 target display. By abstracting the target display space as D , the model can support different
296 targets, such as a 2D screen or 3D printer.

297 3.2.2 Continuity of the Graphic S

298 Just as the K encodes the connectivity of the records in the data, we propose an equivalent
299 S that encodes the connectivity of the rendered elements of the graphic. For some visualiza-
300 tions, K may be lower dimension than S . For example, in a typical 2D display (ignoring
301 depth), a point that is 0D in K cannot be represented on screen unless it is thickened to 2D
302 to encode the connectivity of the pixels that visually represent the point. This thickening is
303 often not necessary when the dimensionality of K matches the dimensionality of the target
304 space, for example if K is 2D and the display is a 2D screen. We introduce S to thicken K
305 in a way which preserves the structure of K .

Formally, we require that K be a deformation retract[79] of S so that K and S have the same homotopy. The surjective map $\xi : S \rightarrow K$

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

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

308 When K is discrete points and the graphic is a scatter plot, each point $k \in K$ corresponds
309 to a 2D disk S_k as shown in figure 10. In the case of 1D continuous data and a line plot,
310 the region β over a point α_i specifies the thickness of the line in S for the corresponding
311 τ on k . The image has the same dimensions in data space and graphic space such that no
312 extra dimensions are needed in S .

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

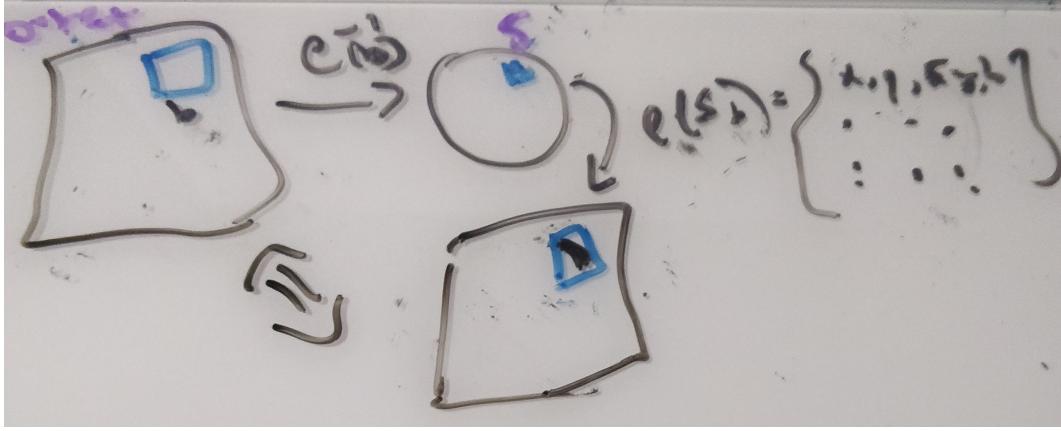


Figure 11: To render a graphic, a pixel p is selected in the display space, which is defined in the same coordinates as the x and y components in D . The inverse mapping $\rho_{xy}^{-1}(p)$ returns a region $S_p \subset S$. $\rho(S_p)$ returns the list of elements $(x, y, r, g, b) \in D$ that lie over S_p . The integral over the (r, g, b) elements is the color of the pixel.

3.2.3 Rendering ρ

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

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

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

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

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

As shown in figure 11, a pixel p in the output space is selected and inverse mapped into the corresponding region $S_p \subset S$. This triggers a lookup of the ρ over the region S_p , which yields the set of elements in D that specify the (r, g, b) values corresponding to the region p . The color of the pixel is then obtained by taking the integral of $\rho_{rgb}(S_p)$.

In general, ρ is an abstraction of rendering. In very broad strokes ρ can be a specification such as PDF[13], SVG[75], or an openGL scene graph[24]. Alternatively, ρ can be a rendering engine such as cairo[22] or AGG[7]. Implementation of ρ is out of scope for this work,

337 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[54] architecture that builds visual elements to pass off to the renderer. 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 , $\mathcal{O}(H)$.

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

338 Sheafs are a mathematical object with restriction maps that define how to glue τ over local
 339 neighborhoods $U \subseteq K$, discussed in section ??, such that the A maps are consistent over
 340 continuous regions of K . While Acan usually construct graphical elements solely with the
 341 data in τ , some visualizations, such as line, may also need some finite number n of derivatives,
 342 which is captured by the jet bundle \mathcal{J}^n [55, 69] with $\mathcal{J}^0(E) = E$. In this work, we at most
 343 need $\mathcal{J}^2(E)$ which is the value at τ and its first and second derivatives; therefore the artist
 344 takes as input the data bundle padded with the jet bundle $E' = E + \mathcal{J}^2(E)$.

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

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

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

352 3.3.1 Visual Fiber Bundle V

353 We introduce a visual bundle V to store the visual representations the artist needs to
 354 assemble into a graphic. The visual bundle (V, K, π, P) has section $\mu : V \rightarrow K$ that
 355 resolves to a visual variable in the fiber P . The visual bundle V is the latent space of
 356 possible parameters of a visualization type, such as a scatter or line plot. We define P
 357 in terms of the parameters of a visualization libraries compositing functions; for example
 358 table 1 is a sample of the fiber space for Matplotlib [53].

ν_i	μ_i	$\text{codomain}(\nu_i)$
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

359 A section μ is a tuple of visual values that specifies the visual characteristics of a part of
 360 the graphic. For example, given a fiber of $\{xpos, ypos, color\}$ one possible section could be
 361 $\{.5, .5, (255, 20, 147)\}$. The $\text{codomain}(\nu_i)$ determines the monoid actions on P_i . These fiber
 362 components are implicit in the library, by making them explicit as components of the fiber
 363 we can build consistent definitions and expectations of how these parameters behave.

364 3.3.2 Visual Encoders ν

As introduced in section 2.3, there are many ways to visually represent data components.
 We define the visual transformers ν

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

365 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
 366 there is a monoid M'_i on V_i , then there is a monoid homomorphism from $M_i \rightarrow M'_i$ that
 367 ν must preserve. As mentioned in section 3.1.2, we choose monoid actions as the basis for
 368 equivariance because they define the structure on the fiber components.

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

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

commutes such that

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

This equivariance constraint yields guidance on what makes for an invalid transform.
 For example, given a visual fiber of the same type as the data fiber $F = P$, the transform
 $\nu_i(x) = .5$ does not commute under translation monoid action $t(x) = x + 2$

$$\nu(t(r + 2)) \stackrel{?}{=} \nu(r) + \nu(2) \quad (26)$$

$$.5 \neq .5 + .5 \quad (27)$$

```

[2]: nu = {'confused': ':(', 'woozy': '=(', 'shruggy': '=@')
[3]: nu.keys()
[3]: dict_keys(['confused', 'woozy', 'shruggy'])
[4]: nu.values()
[4]: dict_values([(':(', '=(', '@=')])
[14]: values
[14]: ['woozy', 'shruggy', 'confused']
[15]: [nu[v] for v in values]
[15]: ['=((', '@=)', ':(']

```

Figure 12: In this artis, ν maps the strings to the emojis. For ν to be equivariant, a shuffle in the words should have an equivalent shuffle in the emojis, and a shuffle in the emojis should have an equivalent shuffle in the words.

On the other hand figure 12 illustrates a valid ν mapping from **Strings** to symbols. The group action on these sets is permutation, so shuffling the words must have an equivalent shuffle of the symbols they are mapped to. Continuing with the assumption that $F = P$, we can describe the constraints on the other Steven's measurement group types. To preserve ordinal and partial order monoid actions, ν must be a monotonic function such that given $r_1, r_2 \in E_i$

$$\text{if } delement_1 \leq r_2 \text{ then } \nu(r_1) \leq \nu(r_2) \quad (28)$$

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

$$\nu(x + c) = \nu(x) + \nu(c) \quad (29)$$

while for ratio data, there must be equivalent scaling

$$\nu(xc) = \nu(x)\nu(c) \quad (30)$$

369 The assumption $F = P$ is made here so that we could omit $\varphi : M \rightarrow M'$ maps that
370 convert the monoid action on F to the equivalent action on P . As shown in table 1 though, it
371 is not uncommon for data and visual variables to resolve to the same types. The constraints
372 on ν can be embedded into our artist such that the ν functions are equivariant; they also
373 provide guidance on constructing new equivariant ν functions.

374

3.3.3 Graphic Assembler Q

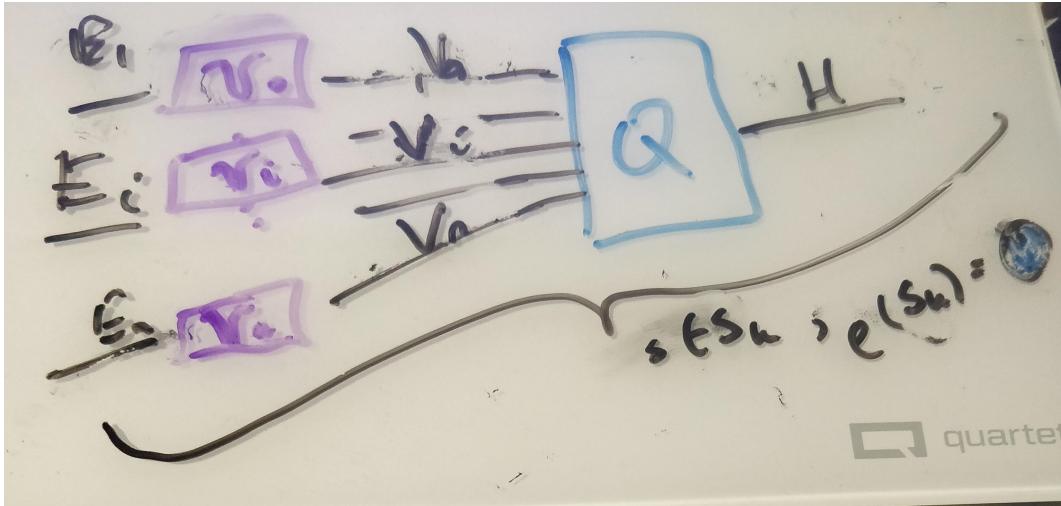


Figure 13: ν functions convert data τ_i to visual characteristics μ_i , then Q assembles μ_i into a graphic ρ such that there is a map ξ preserving the continuity of the data. ρ applied to a region of connected components S_j generates a part of a graphic, for example the point graphical mark.

375 As shown in figure 13, the assembly function Q combines the fiber F_i wise ν transforms into
376 a graphic in H . Together, ν and Q are a map-reduce operation: map the data into their

377 visual encodings, reduce the encodings into a graphic. As with ν the constraint on Q is
 378 that for every monoid action on the input μ there is corresponding monoid action on the
 379 output ρ .

380 Since we define the equivariant map as $Q : \mu \mapsto \rho$, we define an action on the subset
 381 of graphics $Q(\Gamma(V)) \in \Gamma(H)$ that Q can generate. We then define the constraint on Q such
 382 that if Q is applied to μ, μ' that generate the same ρ then the output of both sections acted
 383 on by the same monoid m must be the same.

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

$$J = \{j \in K \text{ s. t. } \exists \gamma \text{ s.t. } \gamma(0) = k \text{ and } \gamma(1) = j\} \quad (31)$$

where the path[30] γ from k to j is a continuous function from the interval $[0,1]$. We define the glyph as the graphic generated by $Q(S_j)$

$$H \xrightleftharpoons[\rho(S_j)]{\xi(s)} S_j \xrightleftharpoons[\xi^{-1}(J)]{} J_k \quad (32)$$

384 such that for every glyph there is at least one corresponding section on K . This is in
 385 keeping with the definition of glyph as any differentiable element put forth by Ziemkiewicz
 386 and Kosara[106]. The primitive point, line, and area marks[11, 23] are specially cased glyphs

$\Gamma(V) = X$ and the graphic $Q(\Gamma(V)) = Y$. If for all monoids $m \in M$ and for all $\mu, \mu' \in X$, the output is equivalent

$$Q(\mu) = Q(\mu') \implies Q(m \circ \mu) = Q(m \circ \mu') \quad (33)$$

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

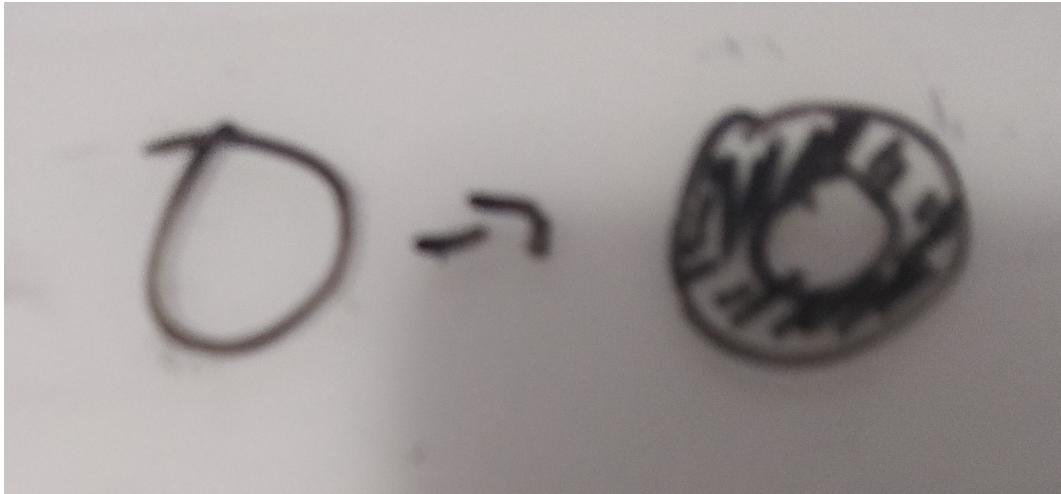


Figure 14: These two glyphs are generated by the same Q function, but differ in the value of the edge thickness parameter μ_i . A valid Q is one where a shift in μ_i is reflected in the glyph generated by ρ .

390 The glyph in figure 14 has the following characteristics P specified by $(xpos, ypos, color, thickness)$
 391 such that one section is $\mu = (0, 0, 0, 1)$ and $Q(\mu) = \rho$ generates a piece of the thin hollow
 392 circle. The equivariance constraint on Q is that the action $m = (e, e, e, x + 2)$, where e is
 393 identity, translates μ to $\mu' = (e, e, e, 3)$. The corresponding action on ρ causes $Q(\mu')$ to be
 394 the thicker circle in figure 14.
 395

396 3.3.4 Assembly Q

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

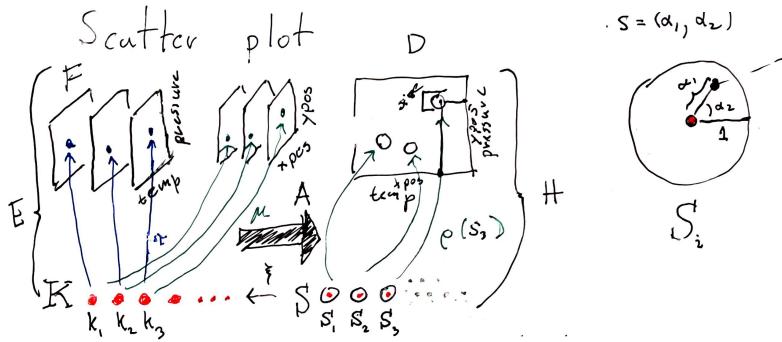


Figure 15: The data is discrete points (temperature, time). Via ν these are converted to $(xpos, ypos)$ and pulled over discrete S . These values are then used to parameterize ρ which returns a color based on the parameters $(xpos, ypos)$ and position α, β on S_k that ρ is evaluated on.

The scatter plot in figure ?? 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 15:

$$x = size \bullet \alpha \bullet \cos(\beta) + xpos \quad (34)$$

$$y = size \bullet \alpha \bullet \sin(\beta) + ypos \quad (35)$$

399 such that $\rho(s) = (x, y, 0, 0, 0)$ colors the point (x, y) black.

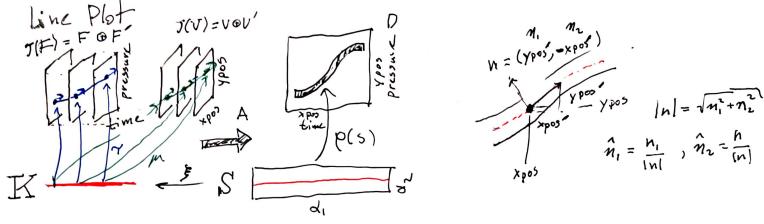


Figure 16: The line fiber (*time, temp*) is thickened with the derivative (*time', temperature'*) because that information will be necessary to figure out the tangent to the point to draw a thick line. This is because the line needs to be pushed perpendicular to the tangent of (*xpos, ypos*). *this is gonna move once this gets regenerated w/ labels* The data is converted to visual characteristics (*xpos, ypos*). The α coordinates on S specifies the position of the line, the β coordinate specifies thickness.

The line plot $Q(xpos, \hat{n}_1, ypos, \hat{n}_2)(\alpha, \beta)$ shown in fig 15 exemplifies the need for the jet discussed in section ???. 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 thickness is

$$|n| = \sqrt{n_1^2 + n_2^2} \quad (36)$$

such that the normal is

$$\hat{n}_1 = \frac{n_1}{|n|}, \quad \hat{n}_2 = \frac{n_2}{|n|} \quad (37)$$

which yields components of ρ

$$x = xpos(\xi(\alpha)) + \hat{\beta}(n_1)(\xi(\alpha)) \quad (38)$$

$$y = ypos(\xi(\alpha)) + \hat{\beta}(n_2)(\xi(\alpha)) \quad (39)$$

400 where (x,y) look up the position $\xi(\alpha)$ on the data and then apply thickness β at that
401 location.

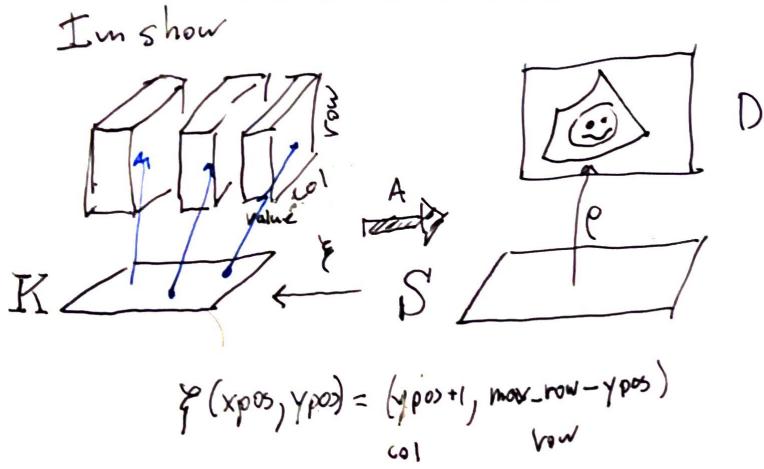


Figure 17: The only visual parameter a image requires is color since ξ encodes the mapping between position in data and position in graphic.

402 The image $Q(\text{color})$ in figure 17 is a direct lookup $\xi : S \rightarrow K$ such that

$$R = R(\xi(\alpha, \beta)) \quad (40)$$

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

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

403 where ξ may do some translating to a convention expected by Q for example reorientng the
404 array such that the first row in the data is at the bottom of the graphic.

405 **3.3.5 Assembly factory \hat{Q}**

406 The graphic base space S is not accessible in many architectures, including Matplotlib,
407 because the rendering is tightly interlaced with the graphical design; instead we can construct
408 a factory function \hat{Q} over K that can build a Q . As shown in eq 22, Q is a bundle map
409 $Q : \xi^*V \rightarrow H$ where ξ^*V and H are both bundles over S .

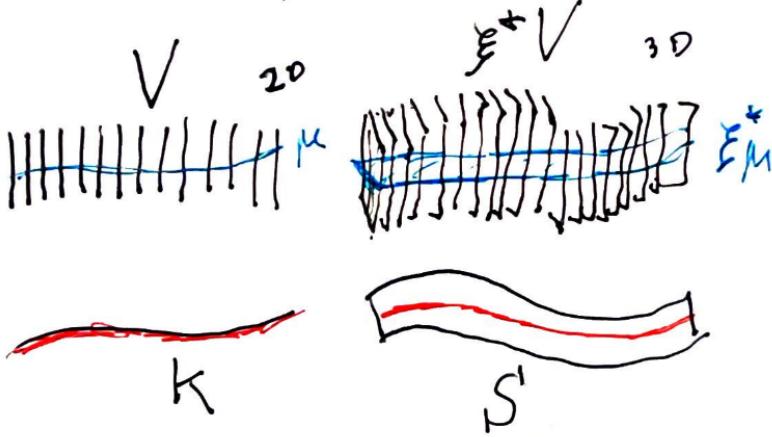


Figure 18: The pullback of the visual bundle ξ^*V is the replication of a μ over all points s that map back to a single k . Because the μ is the same, we can construct a \hat{Q} on μ over k that will fabricate the Q for the equivalent set of s associated to that k

410 The preimage of the continuity map $\xi^{-1}(k) \subset S$ is such that many graphic continuity
 411 points $s \in S_K$ go to one data continuity point k ; therefore, by definition the pull back
 412 $\xi^*V|_{\xi^{-1}(k)} = \xi^{-1}(k) \times P$ copies the visual fiber V over the preimage in $\xi^{-1}(k)$. This is
 413 illustrated in figure 18, where the 1D fiber over K is copied repeatedly to become the 2D fiber
 414 with identical components over S . Given a section $\xi^*\mu$ pulled back from μ on $\pi : V \rightarrow K$
 415 and a point $s \in \xi^{-1}(k)$ in the preimage of k the pulled back section $\xi^*\mu(s) = \xi^*(\mu(k))$ is
 416 the image $(k, \mu(k)) \mapsto (s, \xi^*\mu(s))$. This means that $\xi^*\mu$ is identical for all s where $\xi(s) = k$,
 417 which is illustrated in figure 18 as each dot on P is equivalent to the line intersection $P^*\mu$.

418 Given the equivalence between μ and $\xi^*\mu$ defined above, the reliance on S can be factored
 419 out. When Q maps visual sections $Q : \Gamma(\xi^*V) \rightarrow \Gamma(H)$, if we restrict Q input to $\xi^*\mu$ then
 420 $\rho(s) := Q(\xi^*\mu)(s)$. Since $\xi^*\mu(s) = \xi^*(\mu(k))$ and $\xi^*\mu$ is constant on $\xi^{-1}(k)$, we can define a
 421 Q factory function $\hat{Q}(\mu(k))(s) := Q((\xi^*\mu)(s))$ where $\xi^{-1}(k) = k$.

422 Factoring out s $\hat{Q}(\mu(k)) = Q(\xi^*\mu)$ generates a curried Q . In fact, \hat{Q} is a map from visual
 423 space to graphic space $\hat{Q} : \Gamma(V) \rightarrow \Gamma(H)$ locally over k such that $\hat{Q} : \Gamma(V_k) \rightarrow \Gamma(H|_{\xi^{-1}(k)})$.
 424 This allows us to construct a \hat{Q} that only depends on K , such that for each $\mu(k)$ there
 425 is part of $\rho|_{\xi^{-1}(k)}$. The construction of \hat{Q} allows us to retain the functional map reduce
 426 benefits of Q without having to majorly restructure the existing rendering pipeline.

427 3.3.6 Sheaf

The restriction maps of a sheaf describe how local τ can be glued into larger sections [40, 41]. 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

$$\begin{array}{ccc} \iota^* E & \xhookrightarrow{\iota^*} & E \\ \pi \downarrow \lrcorner \iota^* \tau & & \pi \downarrow \lrcorner \tau \\ U & \xhookrightarrow{\iota} & K \end{array} \quad (43)$$

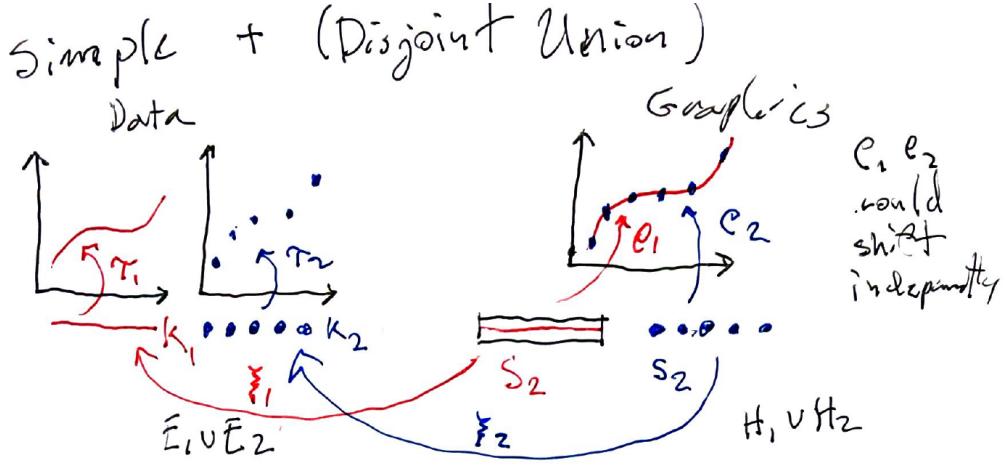


Figure 19: τ_1 and τ_2 are distinct datasets passed through artists A_1 and A_2 to generate graphics ρ_1 and ρ_2 . These graphics happen to be rendered to the same image, but otherwise have no intrinsic link.

such that the pulled back $\iota^*\tau$ only contains records over $U \subset K$. By gluing $\iota^*\tau$ together, the sheaf is putting a continuous structure on local sections which allows for defining a section over a subset in K . That section over subset K maps to the graphic generated by A for visualizations such as sliding windows[27, 31] streaming data, or navigation techniques such as pan and zoom[72].

3.3.7 Composition of Artists: +

To build graphics that are the composites of multiple artists, we define a simple addition operator that is the disjoint union of fiber bundles E . For example, in figure 19 the scatter plot E_1 and the line plot E_2 have different K that are mapped to separate S . To fully display both graphics, the composite graphic $A_1 + A_2$ needs to include all records on both K_1 and K_2 , which are the sections on the disjoint union $K_1 \sqcup K_2$. This in turn yields disjoint graphics $S_1 \sqcup S_2$ rendered to the same image. Constraints can be placed on the disjoint union such as that the fiber components need to have the same ν position encodings or that the position μ need to be in a specified range. There is a second type of composition where E_1 and E_2 share a base space $K_2 \hookrightarrow K_1$ such that the artists can be considered to be acting on different components of the same section. This type of composition is important for creating visualizations where elements need to update together in a consistent way, such as multiple views [6, 74] and brush-linked views[10, 17].

3.3.8 Equivalence class of artists A'

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

452 that defines what visual characteristics of the graphic must originate in the data such that
 453 the graphic is identifiable as a given chart type.

Figure 20: Each of these graphics is generated by a different artist *A* which is the equivalence class of scatter plots *A'*
 this is gonna be a whole bunch of scatter plots

454 For example, a scatter plot of red circles is the output of one artist, a scatter plot of
 455 green squares the output of another, as shown in figure ???. These two artists are equivalent
 456 since their only difference is in the literal visual encodings (color, shape). Shape and color
 457 could also be defined in Q but the position must come from the fiber $P = (xpos, ypos)$ since
 458 fundamentally a scatter plot is the plotting of one position against another[38]. We also use
 459 this criteria to identify derivative types, for example the bubble chart[93] is a type of scatter
 460 where by definition the glyph size is mapped from the data. The criteria for equivalence
 461 class membership serves as the basis for evaluating invariance[56].

4 Prototype Implementation: Matplottoy

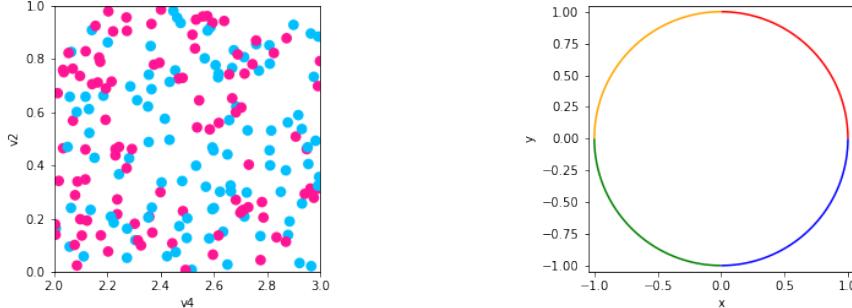


Figure 21: Scatter plot and line plot implemented using prototype artists and data models,
 building on Matplotlib rendering. update with bar

463 To prototype our model, we implemented the artist classes for the scatter and line plots
 464 shown in figure 21 because they differ in every attribute: different visual channels ν that
 465 composite to different marks Q with different continuities ξ . We make use of the Matplotlib
 466 figure and axes artists [53, 54] so that we can initially focus on the data to graphic trans-
 467 formations.

468 To generate the images in figure 21, we instantiate `fig`, `ax` artists that will contain the
 469 new `Point`, `Line` primitive objects we implemented based on our topology model.

<pre> 1 fig, ax = plt.subplots() 2 artist = Point(data, transforms) 3 ax.add_artist(artist) </pre>	<pre> 1 fig, ax = plt.subplots() 2 artist = Line(data, transforms) 3 ax.add_artist(artist) </pre>
--	---

We then add the `Point` and `Line` artist that construct the scatter and line graphics. These artists are implemented as the equivalence class A' with the aesthetic configurations factored out into a `transforms` dictionary that specifies the visual bundle V . The equivalence classes A' map well to Python classes since the functional aspects ν , \hat{Q} , and ξ - are completely reusable in a consistent composition, while the visual values in V are what change between different artists belonging to the same class A' . The `data` object is an abstraction of a data bundle E with a specified section τ . *Implementing H and ρ are out of scope for this prototype because they are part of the rendering process. We also did not implement ν .*

4.1 Artist Class A'

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

```

1  class ArtistClass(matplotlib.artist.Artist):
2      |\label{code:artist}|
3      def __init__(self, data, transforms, *args, **kwargs):
4          # properties that are specific to the graphic but not the channels
5          self.data = data
6          self.transforms = transforms
7          super().__init__(*args, **kwargs)
8
9      def assemble(self, visual):
10         # set the properties of the graphic
11
12     def draw(self, renderer):
13         # returns K, indexed on fiber then key
14         # is passed the
15         view = self.data.view(self.axes)
16         # visual channel encoding applied fiberwise
17         visual = {p: encoder(view[f] if f is not None else None)
18                   for p, (f, encoder) in self.transforms.items()}
19         self.assemble(visual)
20         # pass configurations off to the renderer
21         super().draw(renderer)

```

The data is fetched in section τ via a `view` method on the data because the input to the artist is a section on E . The `view` method takes the `axes` attribute because it provides the region in graphic coordinates S that we can use to query back into data to select a subset

480 as discussed in section ???. The ν functions are then applied to the data to generate the
 481 visual section μ that here is the object `visual`. We allow for fixed visual parameter, such
 482 as setting a constant color for all sections, via setting in `None` as the data fiber F name in
 483 the `transform` dictionary.

484 The visual object is then passed into the `assembly` function that is \hat{Q} . This assembly
 485 function is responsible for generating a representation of the glyph such that it could be
 486 serialized to recreate a static version of the graphic. Although `assemble` could be imple-
 487 mented outside the class such that it returns an object the artist could then parse to set
 488 attributes, the attributes are directly set here to reduce indirection. This artist is not opti-
 489 mized because we prioritized demonstrating the separability of ν and \hat{Q} . The last step in the
 490 artist function is handing itself off to the renderer. The extra `*args`, `**kwargs` arguments in
 491 `__init__, draw` are artifacts of how these objects are currently implemented in Matplotlib.

492 The `Point` artist builds on `collection` artists because collections are optimized to ef-
 493 ficiently draw a sequence of primitive point and area marks. In this prototype, the scatter
 494 marker shape is fixed as a circle, and the only visual fiber components are x and y position,
 495 size, and the facecolor of the marker. We only show the `assembly` function here because
 496 the `__init__, draw` are identical the prototype artist shown in ??.

```

1  class Point(mcollections.Collection):=
2      def assemble(self, visual):
3          # construct geometries of the circle marks in visual coordinates
4          self._paths = [mpath.Path.circle(center=(x,y), radius=s)
5                         for (x, y, s) in zip(visual['x'], visual['y'], visual['s'])]
6          # set attributes of marks, these are vectorized
7          # circles and facecolors are lists of the same size
8          self.set_facecolors(visual['facecolors'])

```

497 The `view` method repackages the data as a fiber component indexed table of vertices, as
 498 described in section ???. Even though the `view` is fiber indexed, each vertex at an index k has
 499 corresponding values in section $\tau(k_i)$. This means that all the data on one vertex maps to
 500 one glyph. To ensure the integrity of the section, `view` must be atomic. This means that the
 501 values cannot change after the method is called in `draw` until a new call in `draw`. We put this
 502 constraint on the return of the `pythonview` method so that we do not risk race conditions.
 503 Using triangulation provides a common interface to the data, one we will reuse for all the
 504 artists presented here.

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

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

```

1  class Line(mcollections.LineCollection):
2      def assemble(self, visual):
3          #assemble line marks as set of segments
4          segments = [np.vstack((vx, vy)).T for vx, vy
5                      in zip(visual['x'], visual['y'])]
6          self.set_segments(segments)
7          self.set_color(visual['color'])

```

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

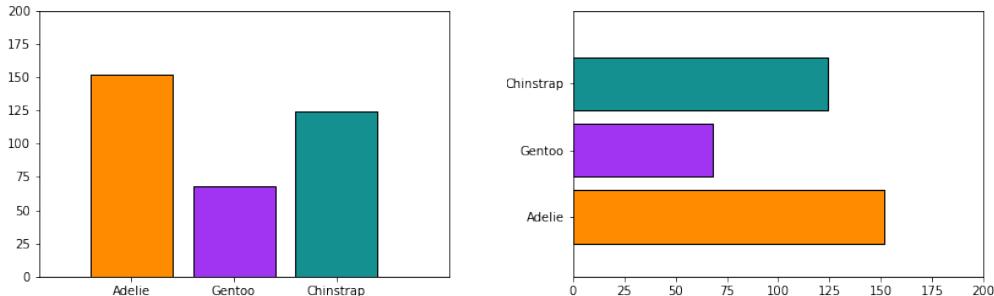


Figure 22: Frequency of Penguin types visualized as discrete bars.

520 The bar charts in figure 22 are generated with a `Bar` artist. The have the same required
 521 P components of (position, length). In of `Bar` an additional parameter is set, `orientation`
 522 which controls whether the bars are arranged vertically or horizontally and only applies
 523 holistically to the graphic and never to individual data parameters. This highlights another
 524 advantage of this model, that it encourages explicit differentiation between parameters in
 525 V and graphic parameters that specify \hat{Q}

```

1  class Bar(mcollections.Collection):
2      def __init__(self, data, transforms, orientation = 'v', *args, **kwargs):
3          # parameter of the graphic
4          self.orientation = orientation
5          super().__init__(*args, **kwargs)
6          self.data = data

```

```

7         self.transforms = transforms
8
9     def assemble(self, visual):
10        #set some defaults
11        visual.setdefault('width', itertools.repeat(0.8))
12        visual.setdefault('floor', itertools.repeat(0))
13        visual.setdefault('facecolors', 'C0')
14        #build bar glyphs based on graphic parameter
15        if self.orientation in {'vertical', 'v'}:
16            order = ['position', 'width', 'floor', 'length']
17        elif self.orientation in {'horizontal', 'h'}:
18            order = ['floor', 'length', 'position', 'width']
19
20        verts = [[(x, y), (x, y+yo), (x+xo, y+yo), (x+xo, y), (x, y)]
21                  for (x, xo, y, yo) in zip(*[visual[k] for k in order])]
22
23        self._paths = [mpath.Path(xy, closed=True) for xy in verts]
24        self.set_edgecolors('k')
25        self.set_facecolors(visual['facecolors'])

```

526 The `draw` method is the same as the one in the artist example ?? so is omitted here.
527 This bar underpins a rudimentary version of the complex operator such that we can specify
528 multiple components to map to the same visual components without requiring the compo-
529 nents to be individually specified in the `transforms` dictionary. The implementation of
530 this support is slightly more complicated than the simple case, but not in a way that is
531 useful for understanding the framework. The `assemble` function constructs bars and sets
532 their edge color to black. Defaults are provided for 'width' and 'floor' to make this function
533 more reusable. Typically the defaults are used for the type of chart shown in figure 22, but
534 these visual variables are often set when building composite versions of this chart type as
535 discussed in section ??.

536 4.2 Encoders ν

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

```

1 cmap = color.Categorical({'true':'deeppink', 'false':'deepskyblue'})
2 transforms = {'y': ('v1', lambda x: x,
3                 'x': ('v3', lambda x: x),

```

```

4     'facecolors': ('v2', cmap),
5     's':(None ,lambda _: itertools.repeat(.02))}
```

542 where the position (x,y) ν transformers are identity functions. The size s transformer is not
 543 acting on a component of F , instead it is a ν that returns a constant value. While size could
 544 be embedded inside the `assembly` function, it is added to the transformers to illustrate user
 545 configured visual parameters that could either be constant or mapped to a component in F .
 546 The identity and constant ν are explicitly implemented here to demonstrate their implicit
 547 role in the visual pipeline, but they could be optimized away. More complex encoders can
 548 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]
```

549 where `__init__` can validate that the output of the ν is a valid element of the P component
 550 the ν function is targeting. Creating a callable class also provides a simple way to
 551 swap out the specific (data, value) mapping without having to reimplement the validation
 552 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)
```

553 but is currently factored out of the artist for clarity. In this example, `is_nominal` checks
 554 for equivariance of permutation group actions by applying the encoder to a set of values,
 555 shuffling values, and checking that (value, encoding) pairs remain the same. This equivariance
 556 test can be implemented as part of the artist or encoder, but for minimal overhead,
 557 the equivariant it is implemented as part of the library tests.

558 4.3 Data E

559 The data input into the will often be a wrapper class around an existing data structure.
 560 This wrapper object must specify the fiber components F and connectivity K and have a

561 that returns an atomic object that encapsulates τ . The object returned by the view must
 562 be key valued pairs of {component name : component section} where each section is a
 563 component as defined in equation 15. To support specifying the fiber bundle, we define a
 564 `FiberBundle` data class[33]

```

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

```

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

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

```

1  class VertexSimplex: #maybe change name to something else
2      """Fiberbundle is consistent across all sections
3      """
4
5      FB = FiberBundle({'tables': ['vertex']},
6                         {'v1': {'type': float, 'monoid': 'interval', 'range': [0,1]},
7                          'v2': {'type': str, 'monoid': 'nominal', 'range': ['true', 'false']},
8                          'v3': {'type': float, 'monoid': 'interval', 'range': [2,3]}})
9
10     def __init__(self, sid = 45, size=1000, max_key=10**10):
11         # create random list of keys
12     def tau(self, k):
13         # e1 is sampled from F1, e2 from F2, etc...
14         return (k, (e1, e2, e3, e4))

```

```

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

```

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

```

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

```

```

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

```

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

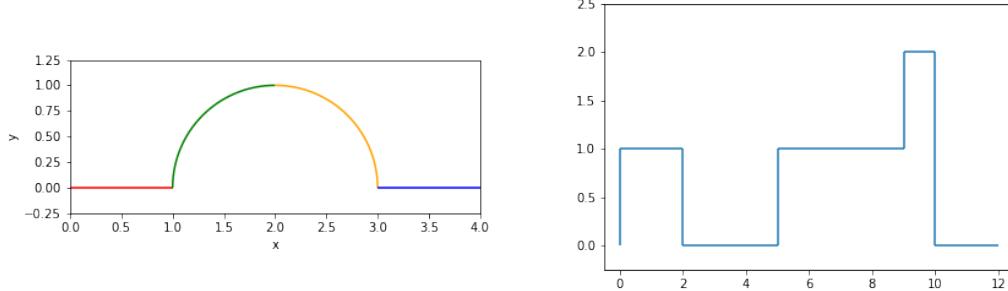


Figure 23: Continuous and discontinuous lines as defined by different data models, but generated with the same $A'=\text{artist}$

586 The graphics in figure 23 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          # 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

```

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

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

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

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

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

595 4.4 Case Study: Penguins

596 For this case study, we use the Palmer Penguins dataset[42, 51] since it is multivariate and
 597 has a varying number of penguins. We use a version of the data packaged as a pandas
 598 dataframe[70, 78] since that is a very commonly used Python labeled data structure. The
 599 wrapper is very thin since here there is explicitly only one section.

```
1 class DataFrameSection:
2     def __init__(self, dataframe):
3         self._tau = dataframe.iloc
4         self._view = dataframe
5     def view(self, axes):
6         return self._view
```

600 The pandas indexer is a key valued set of discrete vertices, so there is no need to repackage
 601 for triangulation.

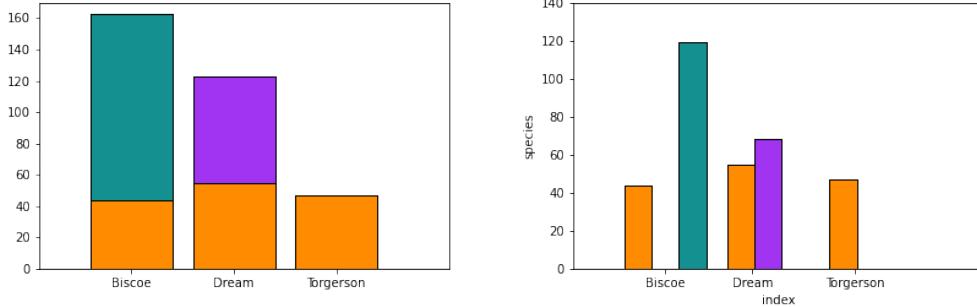


Figure 24: Penguin count disaggregated by island and species

602 For example, the `MultiBar` artist that makes figure 24 reuses `_make_bars` but does
 603 not reuse the `assemble` function because the composition of elements forces fundamental
 604 differences in glyph construction. As demonstrated in the `init`, the composite bar chart
 605 has orientation and whether it is stacked or not. While the stacked bar chart and the grouped
 606 bar chart could be separate artists, as demonstrated they share so much overlapping code
 607 that it is far less redundant to implement them together. *looking at the mess that is this
 608 code, I'm a) not convinced these should be combined b) no longer convinced this provides
 609 anything over just bar if it isn't rewritten to use bar more*

```

1  class MultiBar(mcollections.Collection):
2      def __init__(self, data, transforms, *args, **kwargs):
3          #set the orientation of the graphic
4          self.orientation = kwargs.pop('orientation', 'v')
5          # set how the bar glyphs are put together to create the graphic
6          self.stacked = kwargs.pop('stacked', False)
7          # rest is same as other artist __init__s
8
9          #this needs to be factored out but just want to finish now
10         self.width = kwargs.pop('width', .8)
11
12     def assemble(self, visual, view):
13         (groups, gencoder) = self.transforms['length']
14         ngroups = len(np.atleast_1d(groups))
15         visual['floor'] = visual.get('floor', np.empty(len(view[groups[0]])))
16         visual['facecolors'] = visual.get('facecolors', 'C0')
17         # make equal width stacked columns
18         if 'width' not in visual and self.stacked:

```

```

19         visual['width'] = itertools.repeat(self.width)
20
21     # make equal with groups
22     if not self.stacked:
23         visual['width'] = itertools.repeat(self.width/ngroups)
24         offset = (np.arange/ngroups) /ngroups) * self.width
25     else:
26         offset = itertools.repeat(0)
27
28     # make the bars and arrange them
29     verts = []
30     for group, off in zip(groups, offset):
31         verts.extend(Bar._make_bars(self.orientation, visual['position'] + off,
32                                     visual['width'], visual['floor'], view[group]))
33     if self.stacked: #add stacked bar to previous bar
34         visual['floor'] += view[group]
35
36     # convert lengths after all calculations are made and reorient if needed
37     # here or in transform machinery?
38     if self.orientation in {'v', 'vertical'}:
39         tverts = [[(x, gencoder(y)) for (x, y) in vert]
40                   for vert in verts]
41     elif self.orientation in {'h', 'horizontal'}:
42         tverts = [[(gencoder(x), y) for (x, y) in vert]
43                   for vert in verts]
44     self._paths = [mpath.Path(xy, closed=True) for xy in tverts]
45     #flatted columns of colors to match list of bars
46     self.set_facecolor(list(itertools.chain.from_iterable(visual['facecolors'])))
47     self.set_edgecolors('k')
48
49     def draw(self, renderer, *args, **kwargs):
50         view = self.data.view()
51         #exclude converting the group visual length, special cased in assemble
52         visual = utils.convert_transforms(view, self.transforms, exclude=['length'])
53         # pass in view because nu is not distributable so may need to apply it
54         # after visual assembly
55         self.assemble(visual, view)
56         super().draw(renderer, *args, **kwargs)
57         return

```

610 In the `__draw__`, a utility function is used for conversions, but the length transforms
 611 are held until after assembly because the length is computed by adding the current length
 612 to the previous and many transforms are not distributable such that $\nu(x_0 + x_1 + x_2) =$
 613 $\nu(x_0) + \nu(x_1) + \nu(x_2)$. Inside `assemble`, the glyphs are either shifted vertically (`stacked`)
 614 or horizontally (`grouped`) such that the positions are recorded and added to with the next
 615 group. This function allows multiple columns to be mapped to a visual parameter, but it
 616 must be equal numbers of columns

```

1  {'position': ('island', lambda x: {'Biscoe':0, 'Dream':1, 'Torgersen':2}[x]),
2   'length':(['Adelie', 'Chinstrap', 'Gentoo'], lambda x: x),
3   'facecolors': ([['Adelie_s', 'Chinstrap_s', 'Gentoo_s'],
4                  color.Categorical({'Adelie':'#FF8C00',
5                                     'Gentoo':'#159090',
6                                     'Chinstrap':'#A034F0'})])

```

617 such as in this example where for each column contributing to a segment of the bar there is
 618 a corresponding column of colors for this segment. The reason the multibar can work with
 619 such a transformer is because it is relying on the data model to do most of the bookkeeping
 620 of which values get mapped to which bars. This also yields a much simpler function call to
 621 the artist

```

1  fig, ax = plt.subplots()
2  artist = bar.MultiBar(table, trans, orientation='h', stacked=True)
3  ax.add_artist(artist)

```

622 where `trans` is the same dictionary for both stacked and grouped version, as is the
 623 `DataFrameSection` object `table`. The only difference between the two versions is the
 624 `stacked` flag, and the only difference between figures 22 is the `orientation` argument. By
 625 decomposing the architecture into data, visual encoding, and assembly steps, we are able
 626 to build components that are more flexible and also more self contained than the existing
 627 code base.

628 4.5 Summary

629 In general, the way in which we implemented the artist is as follows:

630 ν encoder function converting data to library normalized form, can also be `Encoder.__call__`
 631 method

632 μ dictionary of the form `{parameter:(variable, encoder)}` curried until `Artist.draw`

633 A' objects that take in as input E and V for example `Point`, `Line`, `bar`

634 A `Artist.draw` method which is A' parameterized by V

635 \hat{Q} `Artist.assemble(visual)` method that arranges components of V into glyphs

636 with the data representation implemented as

637 E object with `view` method

638 F named components of the data and their types

639 K determines how the data is bound together when returned by view

640 τ `Data.view()` method that returns component sections

641 While very rough, this API demonstrates that the ideas presented in the math framework
642 are implementable. In choosing a functional approach, if not implementation, we provide a
643 framework for library developers to build reusable encoder ν and assembly Q . We argue that
644 if these functions are built such that they are equivariant with respect to monoid actions
645 and the graphic topology is a deformation retraction of the data topology, then the artist
646 by definition will be a structure and property preserving map from data to graphic.

647 5 Discussion

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

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

671 5.1 Limitations

672 So far this model has only been worked out for a single data set tied to a primitive mark,
673 but it should be extensible to compositing datasets and complex glyphs. The examples and
674 prototype have so far only been implemented for the static 2D case, but nothing in the math
675 limits to 2D and expansion to the animated case should be possible because the model is

676 formalized in terms of the sheaf. While this model supports equivariance of figurative glyphs
677 generated from parameters of the data[9, 21], it currently does not have a way to evaluate
678 the semantic accuracy of the figurative representation. Effectiveness is out of scope for this
679 model because it is not part of the structure being preserved, but potentially a developer
680 building a domain specific library with this model could implement effectiveness criteria in
681 the artists. Also, even though the model is designed to be backend independent, it has only
682 really been tested against the AGG backend. It is especially unknown how this framework
683 interfaces with high performance rendering libraries such as openGL[24]. Because this model
684 has been limited to the graphic design space, it does not address the critical task of laying
685 out the graphics in the image

686 This model and the associated prototype is deeply tied to Matplotlib’s existing archi-
687 tecture. While the model is expected to generalize to other libraries, such as those built on
688 Mackinlay’s APT framework, this has not been worked through. In particular, Mackinlay’s
689 formulation of graphics as a language with semantic and syntax lends itself a declarative in-
690 terface[61], which Heer and Bostock use to develop a domain specific visualization language
691 that they argue makes it simpler for designers to construct graphics without sacrificing
692 expressivity [48]. Similarly, the model presented in this work formulates visualization as
693 equivariant maps from data space to visual space, and is designed such that developers can
694 build software libraries with data and graphic topologies tuned to specific domains.

695 5.2 Future Work

696 While the model and prototype demonstrate that generation of simple marks from the
697 data, there is a lot of work left to develop a model that underpins a minimally viable
698 library. Foremost is implementing a data object that encodes data with a 2D conti-
699 nuous topology and an artist that can consume data with a 2D topology to visualize the
700 image[toryRethinkingVisualizationHighLevel2004, 43, 44] and also encoding a sepa-
701 rate heatmap[60, 102] artist that consumes 1D discrete data. A second important proof of
702 concept artist is a boxplot[100] because it is a graphic that assumes computation on the
703 data side and the glyph is built from semantically defined components and a list of outliers.
704 The model supports simple composition of glyphs by overlaying glyphs at the same position,
705 but more work is needed to define an operator where the fiber bundles have shared $S_2 \hookrightarrow S_1$
706 such that fibers could be pulled back over the subset. While the model’s simple addition
707 supports axes as standalone artists with overlapping visual position encoding, the complex
708 operator would allow for binding together data that needs to be updated together. Ad-
709 ditionally, implementing the complex addition operator and explicit graphic to data maps
710 would allow for developing a mathematical formalism and prototype of how interactivity
711 would work in this model. In summary, the proposed scope of work for the dissertation is

- 712 • expansion of the mathematical framework to include complex addition
- 713 • implementation of artist with explicit ξ
- 714 • specification of interactive visualization
- 715 • mathematical formulation of a graphic with axes labeling
- 716 • implementation of new prototype artists that do not inherit from Matplotlib artists
- 717 • provisional mathematics and implementation of user level composite artists

- 718 • proof of concept domain specific user facing library

719 Other potential tasks for future work is implementing a data object for a non-trivial fiber
720 bundle and exploiting the models section level formalism to build distributed data source
721 models and concurrent artists. This could be pushed further to integrate with topological[50] and functional [77] data analysis methods. Since this model formalizes notions of
722 structure preservation, it can serve as a good base for tools that assess quality metrics[12]
723 or invariance [56] of visualizations with respect to graphical encoding choices. While this
724 paper formulates visualization in terms of monoidal action homomorphisms between fiber-
725 bundles, the model lends itself to a categorical formulation[37, 65] that could be further
726 explored.
727

728 6 Conclusion

729 An unofficial philosophy of Matplotlib is to support making whatever kinds of plots a user
730 may want, even if they seem nonsensical to the development team. The topological frame-
731 work described in this work provides a way to facilitate this graph creation in a rigorous
732 manner; any artist that meets the equivariance criteria described in this work by definition
733 generates a graphic representation that matches the structure of the data being represented.
734 We leave it to domain specialists to define the structure they need to preserve and the maps
735 they want to make, and hopefully make the process easier by untangling these components
736 into separate constrained maps and providing a fairly general data and display model.

737 Yes, need to figure out how to wrap urls cleanly in bib

738 References

- 739 [1] A. Sarikaya et al. “What Do We Talk About When We Talk About Dashboards?”
740 In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (Jan. 2019),
741 pp. 682–692. ISSN: 1941-0506. DOI: 10.1109/TVCG.2018.2864903.
- 742 [2] *[A Series of Statistical Charts Illustrating the Condition of the Descendants of Former
743 African Slaves Now in Residence in the United States of America] Negro Business
744 Men in the United States*. eng. <https://www.loc.gov/item/2014645363/>. Image.
- 745 [3] *[A Series of Statistical Charts Illustrating the Condition of the Descendants of Former
746 African Slaves Now in Residence in the United States of America] Negro Population
747 of the United States Compared with the Total Population of Other Countries /*. eng.
748 <https://www.loc.gov/item/2013650368/>. Image.
- 749 [4] *Action in nLab*. https://ncatlab.org/nlab/show/action#actions_of_a_monoid.
- 750 [5] James Ahrens, Berk Geveci, and Charles Law. “Paraview: An End-User Tool for
751 Large Data Visualization”. In: *The visualization handbook* 717.8 (2005).
- 752 [6] Yael Albo et al. “Off the Radar: Comparative Evaluation of Radial Visualization
753 Solutions for Composite Indicators”. In: *IEEE Transactions on Visualization and
754 Computer Graphics* 22.1 (Jan. 2016), pp. 569–578. ISSN: 1077-2626. DOI: 10.1109/
755 TVCG.2015.2467322.
- 756 [7] *Anti-Grain Geometry* -. <http://agg.sourceforge.net/antigrain.com/index.html>.
- 757 [8] Professor Denis Auroux. “Math 131: Introduction to Topology”. en. In: (), p. 113.

- 758 [9] F. Beck. “Software Feathers Figurative Visualization of Software Metrics”. In: *2014*
 759 *International Conference on Information Visualization Theory and Applications (IVAPP)*. Jan. 2014, pp. 5–16.
- 760
 761 [10] Richard A. Becker and William S. Cleveland. “Brushing Scatterplots”. In: *Techno-*
 762 *metrics* 29.2 (May 1987), pp. 127–142. ISSN: 0040-1706. DOI: 10.1080/00401706.
 763 1987.10488204.
- 764 [11] Jacques Bertin. *Semiology of Graphics : Diagrams, Networks, Maps*. English. Red-
 765 lands, Calif.: ESRI Press, 2011. ISBN: 978-1-58948-261-6 1-58948-261-1.
- 766 [12] Enrico Bertini, Andrada Tatu, and Daniel Keim. “Quality Metrics in High-
 767 Dimensional Data Visualization: An Overview and Systematization”. In: *IEEE*
 768 *Transactions on Visualization and Computer Graphics* 17.12 (2011), pp. 2203–2212.
- 769 [13] Tim Bienz, Richard Cohn, and Calif.) Adobe Systems (Mountain View. *Portable*
 770 *Document Format Reference Manual*. Citeseer, 1993.
- 771 [14] M. Bostock and J. Heer. “Protovis: A Graphical Toolkit for Visualization”. In: *IEEE*
 772 *Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1121–
 773 1128. ISSN: 1941-0506. DOI: 10.1109/TVCG.2009.174.
- 774 [15] M. Bostock, V. Ogievetsky, and J. Heer. “D³ Data-Driven Documents”. In: *IEEE*
 775 *Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pp. 2301–
 776 2309. ISSN: 1941-0506. DOI: 10.1109/TVCG.2011.185.
- 777 [16] Brian Wylie and Jeffrey Baumes. “A Unified Toolkit for Information and Scientific
 778 Visualization”. In: *Proc.SPIE*. Vol. 7243. Jan. 2009. DOI: 10.1117/12.805589.
- 779 [17] Andreas Buja et al. “Interactive Data Visualization Using Focusing and Linking”. In:
 780 *Proceedings of the 2nd Conference on Visualization '91. VIS '91*. Washington, DC,
 781 USA: IEEE Computer Society Press, 1991, pp. 156–163. ISBN: 0-8186-2245-8.
- 782 [18] D. M. Butler and M. H. Pendley. “A Visualization Model Based on the Mathematics
 783 of Fiber Bundles”. en. In: *Computers in Physics* 3.5 (1989), p. 45. ISSN: 08941866.
 784 DOI: 10.1063/1.168345.
- 785 [19] David M. Butler and Steve Bryson. “Vector-Bundle Classes Form Powerful Tool
 786 for Scientific Visualization”. en. In: *Computers in Physics* 6.6 (1992), p. 576. ISSN:
 787 08941866. DOI: 10.1063/1.4823118.
- 788 [20] L. Byrne, D. Angus, and J. Wiles. “Acquired Codes of Meaning in Data Visualization
 789 and Infographics: Beyond Perceptual Primitives”. In: *IEEE Transactions on Visual-*
 790 *ization and Computer Graphics* 22.1 (Jan. 2016), pp. 509–518. ISSN: 1077-2626. DOI:
 791 10.1109/TVCG.2015.2467321.
- 792 [21] Lydia Byrne, Daniel Angus, and Janet Wiles. “Figurative Frames: A Critical Vocab-
 793 uary for Images in Information Visualization”. In: *Information Visualization* 18.1
 794 (Aug. 2017), pp. 45–67. ISSN: 1473-8716. DOI: 10.1177/1473871617724212.
- 795 [22] *Cairogaphics.Org*. <https://www.cairogaphics.org/>.
- 796 [23] Sheelagh Carpendale. *Visual Representation from Semiology of Graphics by J. Bertin*.
 797 en.
- 798 [24] George S. Carson. “Standards Pipeline: The OpenGL Specification”. In: *SIGGRAPH*
 799 *Comput. Graph.* 31.2 (May 1997), pp. 17–18. ISSN: 0097-8930. DOI: 10.1145/271283.
 800 271292.

- 801 [25] A. M. Cegarra. “Cohomology of Monoids with Operators”. In: *Semigroup Forum*.
 802 Vol. 99. Springer, 2019, pp. 67–105. ISBN: 1432-2137.
- 803 [26] John M Chambers et al. *Graphical Methods for Data Analysis*. Vol. 5. Wadsworth
 804 Belmont, CA, 1983.
- 805 [27] Chia-Shang James Chu. “Time Series Segmentation: A Sliding Window Approach”.
 806 In: *Information Sciences* 85.1 (July 1995), pp. 147–173. ISSN: 0020-0255. DOI: 10.
 807 1016/0020-0255(95)00021-G.
- 808 [28] William S. Cleveland. “Research in Statistical Graphics”. In: *Journal of the American*
 809 *Statistical Association* 82.398 (June 1987), p. 419. ISSN: 01621459. DOI: 10.2307/
 810 2289443.
- 811 [29] William S. Cleveland and Robert McGill. “Graphical Perception: Theory, Experi-
 812 mentation, and Application to the Development of Graphical Methods”. In: *Journal*
 813 *of the American Statistical Association* 79.387 (Sept. 1984), pp. 531–554. ISSN: 0162-
 814 1459. DOI: 10.1080/01621459.1984.10478080.
- 815 [30] “Connected Space”. en. In: *Wikipedia* (Dec. 2020).
- 816 [31] Michael S. Crouch, Andrew McGregor, and Daniel Stubbs. “Dynamic Graphs in the
 817 Sliding-Window Model”. In: *European Symposium on Algorithms*. Springer, 2013,
 818 pp. 337–348.
- 819 [32] *Data Representation in Mayavi — Mayavi 4.7.2 Documentation*. <https://docs.enthought.com/mayavi/mayavi/d>
- 820 [33] *Dataclasses — Data Classes — Python 3.9.2rc1 Documentation*. <https://docs.python.org/3/library/dataclasses>.
- 821 [34] W. E. B. (William Edward Burghardt) Du Bois. *[The Georgia Negro] Valuation of*
 822 *Town and City Property Owned by Georgia Negroes*. en. <https://www.loc.gov/pictures/item/2013650441/>.
 823 Still Image. 1900.
- 824 [35] T. W. E. B. Du Bois Center at the University of Massachusetts, W. Battle-Baptiste,
 825 and B. Rusert. *W. E. B. Du Bois’s Data Portraits: Visualizing Black America*.
 826 Princeton Architectural Press, 2018. ISBN: 978-1-61689-706-2.
- 827 [36] Stephen Few and Perceptual Edge. “Dashboard Confusion Revisited”. In: *Perceptual*
 828 *Edge* (2007), pp. 1–6.
- 829 [37] Brendan Fong and David I. Spivak. *An Invitation to Applied Category Theory:*
 830 *Seven Sketches in Compositionality*. en. First. Cambridge University Press, July
 831 2019. ISBN: 978-1-108-66880-4 978-1-108-48229-5 978-1-108-71182-1. DOI: 10.1017/
 832 9781108668804.
- 833 [38] Michael Friendly. “A Brief History of Data Visualization”. en. In: *Handbook of Data*
 834 *Visualization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 15–56. ISBN:
 835 978-3-540-33036-3 978-3-540-33037-0. DOI: 10.1007/978-3-540-33037-0_2.
- 836 [39] Berk Geveci et al. “VTK”. In: *The Architecture of Open Source Applications* 1 (2012),
 837 pp. 387–402.
- 838 [40] Robert Ghrist. “Homological Algebra and Data”. In: *Math. Data* 25 (2018), p. 273.
- 839 [41] Robert W. Ghrist. *Elementary Applied Topology*. Vol. 1. Createspace Seattle, 2014.
- 840 [42] Kristen B. Gorman, Tony D. Williams, and William R. Fraser. “Ecological Sexual
 841 Dimorphism and Environmental Variability within a Community of Antarctic Pen-
 842 guins (Genus Pygoscelis)”. In: *PLOS ONE* 9.3 (Mar. 2014), e90081. DOI: 10.1371/
 843 journal.pone.0090081.

- 844 [43] Robert B Haber and David A McNabb. “Visualization Idioms: A Conceptual Model
 845 for Scientific Visualization Systems”. In: *Visualization in scientific computing* 74
 846 (1990), p. 93.
- 847 [44] Charles D Hansen and Chris R Johnson. *Visualization Handbook*. Elsevier, 2011.
- 848 [45] Marcus D. Hanwell et al. “The Visualization Toolkit (VTK): Rewriting the Rendering
 849 Code for Modern Graphics Cards”. en. In: *SoftwareX* 1-2 (Sept. 2015), pp. 9–12. ISSN:
 850 23527110. DOI: 10.1016/j.softx.2015.04.001.
- 851 [46] Charles R Harris et al. “Array Programming with NumPy”. In: *Nature* 585.7825
 852 (2020), pp. 357–362.
- 853 [47] J. Heer and M. Agrawala. “Software Design Patterns for Information Visualization”.
 854 In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (2006), pp. 853–
 855 860. DOI: 10.1109/TVCG.2006.178.
- 856 [48] Jeffrey Heer and Michael Bostock. “Declarative Language Design for Interactive Vi-
 857 sualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.6
 858 (Nov. 2010), pp. 1149–1156. ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.144.
- 859 [49] Jeffrey Heer, Michael Bostock, and Vadim Ogievetsky. “A Tour Through the Visu-
 860 alization Zoo”. In: *Communications of the ACM* 53.6 (June 2010), pp. 59–67. ISSN:
 861 0001-0782. DOI: 10.1145/1743546.1743567.
- 862 [50] C. Heine et al. “A Survey of Topology-Based Methods in Visualization”. In: *Computer
 863 Graphics Forum* 35.3 (June 2016), pp. 643–667. ISSN: 0167-7055. DOI: 10.1111/cgf.
 864 12933.
- 865 [51] Allison Marie Horst, Alison Presmanes Hill, and Kristen B Gorman. *Palmerpen-
 866 guins: Palmer Archipelago (Antarctica) Penguin Data*. Manual. 2020. DOI: 10.5281/
 867 zenodo.3960218.
- 868 [52] Stephan Hoyer and Joe Hamman. “Xarray: ND Labeled Arrays and Datasets in
 869 Python”. In: *Journal of Open Research Software* 5.1 (2017).
- 870 [53] J. D. Hunter. “Matplotlib: A 2D Graphics Environment”. In: *Computing in Science
 871 Engineering* 9.3 (May 2007), pp. 90–95. ISSN: 1558-366X. DOI: 10.1109/MCSE.2007.
 872 55.
- 873 [54] John Hunter and Michael Droettboom. *The Architecture of Open Source Applications
 874 (Volume 2): Matplotlib*. <https://www.aosabook.org/en/matplotlib.html>.
- 875 [55] “Jet Bundle”. en. In: *Wikipedia* (Dec. 2020).
- 876 [56] Gordon Kindlmann and Carlos Scheidegger. “An Algebraic Process for Visualization
 877 Design”. In: *IEEE transactions on visualization and computer graphics* 20.12 (2014),
 878 pp. 2181–2190.
- 879 [57] John Krygier and Denis Wood. *Making Maps: A Visual Guide to Map Design for
 880 GIS*. English. 1 edition. New York: The Guilford Press, Aug. 2005. ISBN: 978-1-59385-
 881 200-9.
- 882 [58] W A Lea. “A Formalization of Measurement Scale Forms”. en. In: (), p. 44.
- 883 [59] *Locally Trivial Fibre Bundle - Encyclopedia of Mathematics*. https://encyclopediaofmath.org/wiki/Locally_trivial_fibre_bundle.
- 884 [60] Toussaint Loua. *Atlas Statistique de La Population de Paris*. J. Dejey & cie, 1873.
- 885 [61] Kenneth C. Louden. *Programming Languages : Principles and Practice*. English. Pa-
 886 cific Grove, Calif: Brooks/Cole, 2010. ISBN: 978-0-534-95341-6 0-534-95341-7.

- 887 [62] Jock Mackinlay. "Automating the Design of Graphical Presentations of Relational
 888 Information". In: *ACM Transactions on Graphics* 5.2 (Apr. 1986), pp. 110–141. ISSN:
 889 0730-0301. DOI: 10.1145/22949.22950.
- 890 [63] JOCK D. MACKINLAY. "AUTOMATIC DESIGN OF GRAPHICAL PRESEN-
 891 TATIONS (DATABASE, USER INTERFACE, ARTIFICIAL INTELLIGENCE, IN-
 892 FORMATION TECHNOLOGY)". English. PhD Thesis. 1987.
- 893 [64] Connie Malamed. *Information Display Tips*. [https://understandinggraphics.com/visualizations/information-](https://understandinggraphics.com/visualizations/information-display-tips/)
 894 [display-tips/](#). Blog. Jan. 2010.
- 895 [65] Bartosz Milewski. "Category Theory for Programmers". en. In: (), p. 498.
- 896 [66] "Monoid". en. In: *Wikipedia* (Jan. 2021).
- 897 [67] Tamara Munzner. "Ch 2: Data Abstraction". In: *CPSC547: Information Visualiza-*
 898 *tion, Fall 2015-2016* ().
- 899 [68] Tamara Munzner. *Visualization Analysis and Design*. AK Peters Visualization Series.
 900 CRC press, Oct. 2014. ISBN: 978-1-4665-0891-0.
- 901 [69] Jana Musilová and Stanislav Hronek. "The Calculus of Variations on Jet Bundles
 902 as a Universal Approach for a Variational Formulation of Fundamental Physical
 903 Theories". In: *Communications in Mathematics* 24.2 (Dec. 2016), pp. 173–193. ISSN:
 904 2336-1298. DOI: 10.1515/cm-2016-0012.
- 905 [70] Muhammad Chenariyan Nakhaee. *Mcnakhaee/Palmerpenguins*. Jan. 2021.
- 906 [71] *Naturalness Principle - Info Vis: Wiki*. https://infovis-wiki.net/wiki/Naturalness_Principle.
- 907 [72] Dmitry Nekrasovski et al. "An Evaluation of Pan & Zoom and Rubber Sheet
 908 Navigation with and without an Overview". In: *Proceedings of the SIGCHI Con-*
 909 *ference on Human Factors in Computing Systems*. CHI '06. New York, NY, USA:
 910 Association for Computing Machinery, 2006, pp. 11–20. ISBN: 1-59593-372-7. DOI:
 911 10.1145/1124772.1124775.
- 912 [73] Z. Poussman, J. Stasko, and M. Mateas. "Casual Information Visualization: Depic-
 913 tions of Data in Everyday Life". In: *IEEE Transactions on Visualization and Com-*
 914 *puter Graphics* 13.6 (Nov. 2007), pp. 1145–1152. ISSN: 1941-0506. DOI: 10.1109/
- 915 *TVCG*.2007.70541.
- 916 [74] Z. Qu and J. Hullman. "Keeping Multiple Views Consistent: Constraints, Validations,
 917 and Exceptions in Visualization Authoring". In: *IEEE Transactions on Visualization*
 918 *and Computer Graphics* 24.1 (Jan. 2018), pp. 468–477. ISSN: 1941-0506. DOI: 10.
- 919 1109/TVCG.2017.2744198.
- 920 [75] A. Quint. "Scalable Vector Graphics". In: *IEEE MultiMedia* 10.3 (July 2003), pp. 99–
 921 102. ISSN: 1941-0166. DOI: 10.1109/MMUL.2003.1218261.
- 922 [76] "Quotient Space (Topology)". en. In: *Wikipedia* (Nov. 2020).
- 923 [77] James O Ramsay. *Functional Data Analysis*. Wiley Online Library, 2006.
- 924 [78] Jeff Reback et al. *Pandas-Dev/Pandas: Pandas 1.0.3*. Zenodo. Mar. 2020. DOI: 10.
- 925 5281/zenodo.3715232.
- 926 [79] "Retraction (Topology)". en. In: *Wikipedia* (July 2020).
- 927 [80] Matthew Rocklin. "Dask: Parallel Computation with Blocked Algorithms and Task
 928 Scheduling". In: *Proceedings of the 14th Python in Science Conference*. Vol. 126.
 929 Citeseer, 2015.

- 930 [81] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. “Declarative Inter-
 931 action Design for Data Visualization”. en. In: *Proceedings of the 27th Annual ACM*
 932 *Symposium on User Interface Software and Technology*. Honolulu Hawaii USA: ACM,
 933 Oct. 2014, pp. 669–678. ISBN: 978-1-4503-3069-5. DOI: 10.1145/2642918.2647360.
- 934 [82] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. “NIH Image to
 935 ImageJ: 25 Years of Image Analysis”. In: *Nature Methods* 9.7 (July 2012), pp. 671–
 936 675. ISSN: 1548-7105. DOI: 10.1038/nmeth.2089.
- 937 [83] “Semigroup Action”. en. In: *Wikipedia* (Jan. 2021).
- 938 [84] Nicholas Sofroniew et al. *Napari/Napari: 0.4.5rc1*. Zenodo. Feb. 2021. DOI: 10.5281/
 939 zenodo.4533308.
- 940 [85] E.H. Spanier. *Algebraic Topology*. McGraw-Hill Series in Higher Mathematics.
 941 Springer, 1989. ISBN: 978-0-387-94426-5.
- 942 [86] David I Spivak. *Databases Are Categories*. en. Slides. June 2010.
- 943 [87] David I Spivak. “SIMPLICIAL DATABASES”. en. In: (), p. 35.
- 944 [88] S. S. Stevens. “On the Theory of Scales of Measurement”. In: *Science* 103.2684 (1946),
 945 pp. 677–680. ISSN: 00368075, 10959203.
- 946 [89] Michele Stieven. *A Monad Is Just a Monoid...* en. <https://medium.com/@michelestieven/a-monad-is-just-a-monoid-a02bd2524f66>. Apr. 2020.
- 947 [90] Software Studies. *Culturevis/Imageplot*. Jan. 2021.
- 949 [91] T. Sugibuchi, N. Spyros, and E. Siminenko. “A Framework to Analyze Information
 950 Visualization Based on the Functional Data Model”. In: *2009 13th International*
 951 *Conference Information Visualisation*. 2009, pp. 18–24. DOI: 10.1109/IV.2009.56.
- 952 [92] *[The Georgia Negro] City and Rural Population. 1890*. eng. <https://www.loc.gov/item/2013650430/>.
 953 Image. 1900.
- 954 [93] Edward R. Tufte. *The Visual Display of Quantitative Information*. English. Cheshire,
 955 Conn.: Graphics Press, 2001. ISBN: 0-9613921-4-2 978-0-9613921-4-7 978-1-930824-13-
 956 3 1-930824-13-0.
- 957 [94] John W. Tukey. “We Need Both Exploratory and Confirmatory”. In: *The American*
 958 *Statistician* 34.1 (Feb. 1980), pp. 23–25. ISSN: 0003-1305. DOI: 10.1080/00031305.
 959 1980.10482706.
- 960 [95] Jacob VanderPlas et al. “Altair: Interactive Statistical Visualizations for Python”.
 961 en. In: *Journal of Open Source Software* 3.32 (Dec. 2018), p. 1057. ISSN: 2475-9066.
 962 DOI: 10.21105/joss.01057.
- 963 [96] P. Vickers, J. Faith, and N. Rossiter. “Understanding Visualization: A Formal Ap-
 964 proach Using Category Theory and Semiotics”. In: *IEEE Transactions on Visualiza-*
 965 *tion and Computer Graphics* 19.6 (June 2013), pp. 1048–1061. ISSN: 1941-0506. DOI:
 966 10.1109/TVCG.2012.294.
- 967 [97] C. Ware. *Information Visualization: Perception for Design*. Interactive Technologies.
 968 Elsevier Science, 2019. ISBN: 978-0-12-812876-3.
- 969 [98] Eric W. Weisstein. *Similarity Transformation*. en. <https://mathworld.wolfram.com/SimilarityTransformation.html>. Text.
- 971 [99] Hadley Wickham. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New
 972 York, 2016. ISBN: 978-3-319-24277-4.

- 973 [100] Hadley Wickham and Lisa Stryjewski. “40 Years of Boxplots”. In: *The American*
974 *Statistician* (2011).
- 975 [101] Leland Wilkinson. *The Grammar of Graphics*. en. 2nd ed. Statistics and Computing.
976 New York: Springer-Verlag New York, Inc., 2005. ISBN: 978-0-387-24544-7.
- 977 [102] Leland Wilkinson and Michael Friendly. “The History of the Cluster Heat Map”.
978 In: *The American Statistician* 63.2 (May 2009), pp. 179–184. ISSN: 0003-1305. DOI:
979 10.1198/tas.2009.0033.
- 980 [103] Krist Wongsuphasawat. *Navigating the Wide World of Data Visualization Libraries*
981 (*on the Web*). 2021.
- 982 [104] *Writing Plugins*. en. <https://imagej.net/Writing-plugins>.
- 983 [105] Brent A Yorgey. “Monoids: Theme and Variations (Functional Pearl)”. en. In: (),
984 p. 12.
- 985 [106] Caroline Ziemkiewicz and Robert Kosara. “Embedding Information Visualization
986 within Visual Representation”. In: *Advances in Information and Intelligent Systems*.
987 Ed. by Zbigniew W. Ras and William Ribarsky. Berlin, Heidelberg: Springer Berlin
988 Heidelberg, 2009, pp. 307–326. ISBN: 978-3-642-04141-9. DOI: 10.1007/978-3-642-
989 04141-9_15.