

1 **1 Introduction**

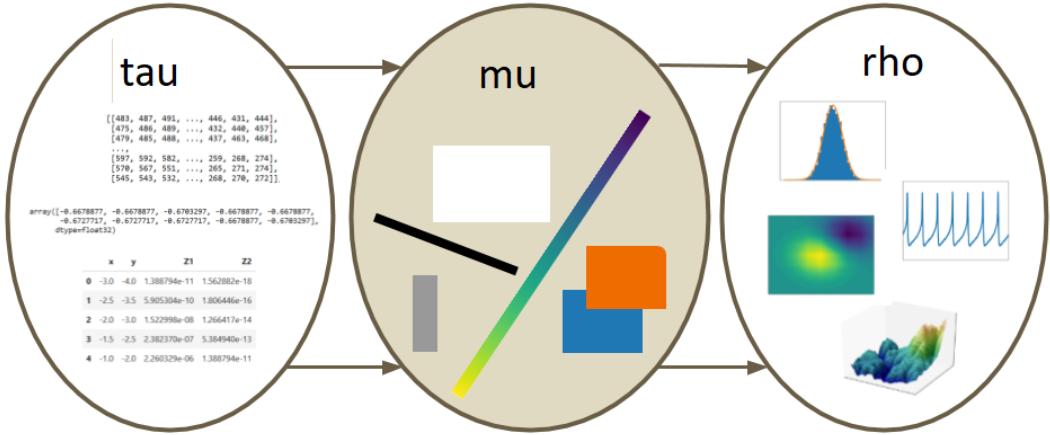


Figure 1: Visualization is equivariant maps between data and visual encoding of the variables and assembly of those encodings into a graphic. will replace w/ overarching figure w/ same structure

2 This work is motivated by a need for a visualization library that developers could build  
3 complex, domain specific tools tuned to the semantics and structure carried in domain spe-  
4 cific data. The core architecture also needs to be robust to the big data needs of many  
5 visualization practitioners, and therefore support distributed and streaming data needs. To  
6 support both exploratory and confirmatory visualization[85], this tool needs to support 2D  
7 and 3D, static, dynamic and interactive visualizations. Specifically, this work was driven by  
8 a rearchiture the Python visualization library Matplotlib[42] to meet modern data visualiza-  
9 tion needs. We aim to take advantage of developments in software design, data structures,  
10 and visualization to improve the consistency, compositability, and discoverability of the API.

11 This work presents a mathematical description of how data is transformed into graphic  
12 representations, as shown in figure 1. While many researchers have identified and described  
13 important aspects of visualization, they have specialized in such different ways as to not  
14 provide a model general enough to natively support the full range of data and visualization  
15 types many general purpose modern visualization tools may need to support. In this work,  
16 we present a framework for understanding visualization as equivariant maps between topo-  
17 logical spaces. Using this mathematical formalism, we can interpret and extend prior work  
18 and also develop new tools. We validate our model by using it to re-design artist and data  
19 access layer of Matplotlib, a general purpose visualization tool.

20 **2 Background**

21 Visual algorithms that display information are designed such that the structure of data  
22 is assumed, as described in Tory and Möller’s taxonomy [83]. Specifically they note that  
23 discrete and continuous data and their attributes form a discipline independent design space  
24 [61]. This assumption is evident in the information visualization software design patterns  
25 categorized by Heer and Agrawala[37] and can often be found in modern general purpose

26 visualization libraries. In proposing a new architecture, we contrast the trade offs libraries  
27 make, describe different types of data continuity, and discuss metrics by which a visualization  
28 library is traditionally evaluated.

29 **2.1 Tools**

30 The library driving this rearchitecture, Matplotlib, aims to be flexible enough that develop-  
31 ers can build domain specific libraries on top of it. To preserve this flexibility, Matplotlib  
32 enforces minimal constraints on what sorts of data the user is allowed to input and supports  
33 very many visual algorithms, from primitive marks to computationally complex visualiza-  
34 tions. Instead of enforcing constraints at the API level, Matplotlib carries implicit assump-  
35 tions about data continuity in how each function interfaces with the input data. This has  
36 lead to poor API consistency and brittle code as every visual algorithm has a very different  
37 point of view on how the data is structured.

38 A commonly cited alternative is the family of tools built on top of Wilkenson’s Grammar  
39 of Graphics (GoG) [92], including ggplot[90], protovis[12] and D3 [13], vega[70] and altair[86].  
40 This framework is very popular in the data visualization community because it lends itself  
41 to a declarative interface [38] which allows end users to describe the visualization they are  
42 trying to create. Grammar oriented approaches describe how to compose visual elements  
43 into a graphical design [94], while we are proposing a framework for building the elements.

44 A different class of user facing tools are those that support images, such as ImageJ[71].  
45 These tools mostly have some support for visualizing non image components of a complex  
46 data set, but mostly in service to the image being visualized. These tools are ill suited for  
47 general purpose library developers as the architecture is to build plugins into the existing  
48 system [95] rather than domain specific tools that are built on top. Even the digital hu-  
49 manities oriented macro ImagePlot[79], which supports some non-image aggregate reporting  
50 charts, is still built around image data as the primary input.

51 There are also visualization tools designed around scientifically complex data that  
52 support explicit description of the data, such as vtk[31, 35] and its derivatives such as  
53 MayaVi[65] and extensions such as ParaView[4] and the infoviz themed Titan[14]. These  
54 libraries, generally speaking, are architected more as graphics libraries than visualization  
55 libraries, meaning they lack a clear distinction between the graphic construction and the  
56 rendering of that graphic. This is very close to the existing Matplotlib architecture, where  
57 the data, visual encoding, and rendering are jumbled in ways that make it hard to figure  
58 out what the code is doing. We are proposing a functional framework instead in large part  
59 to clearly encapsulate these separate responsibilities of a visualization tool. In turn, this  
60 should make it easier to reuse components in the building of new tools.

61 **2.2 Data**

62 As mentioned, one of the drivers of this work was to facilitate building libraries that could  
63 natively support domain specific data containers. Fiber bundles are one such way model  
64 to model containers, as Butler proposes because they encode the continuity of the data  
65 separately from the types of variables and are flexible enough to support discrete and ND  
66 continuous datasets [16, 17]. Since Butler’s model lacks a robust way of describing the  
67 variables, we fold in Spivak’s Simplicial formulation of databases [74, 75] so that we can  
68 encode a schema like description of the data in the fiber bundle.

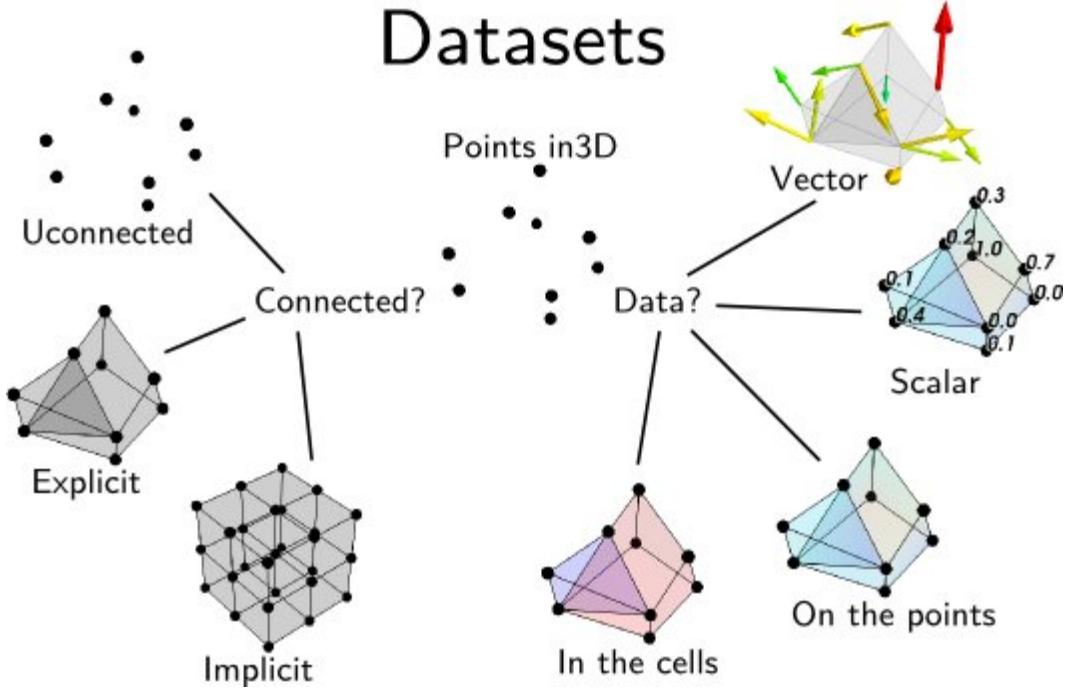


Figure 2: One way to describe data is by the connectivity of the points in the dataset. A database for example is often discrete unconnected points, while an image is an implicitly connected 2D grid. This image is from the Data Representation chapter of the MayaVi 4.7.2 documentation.[27]

As shown in figure 2, there are many types of connectivity. A database typically consists of unconnected records, while an image is an implicit 2D grid and a network is some sort of explicitly connected graph. 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 [56], 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

82 to the position of the value in the dataset, our model allows for renaming all the metadata,  
 83 for example changing the coordinate systems or time resolution, without imposing new  
 84 semantics on the underlying structure.

### 85 2.3 Visualization

86 A visualization tool produces a graphical design and an image rendered based on that design,  
 87 as described by Mackinlay [52]. He defines the graphical design as the set of encoding  
 88 relations from data to visual representation[51], and the design rendered in an idealized  
 89 abstract space is what throughout this paper we will refer to as a graphic. In addition  
 90 to the graphic representations, Byrne et al. describe how visualizations have figurative  
 91 representations that have meaning due to their similarity in shape to external concepts [18].  
 92 Mackinlay proposes that a visualization tool’s expressiveness is a measure of how much of  
 93 the structure of the data the tool encodes, while the tool’s effectiveness describes how much  
 94 design choices are made in deference to perceptual saliency [23–25, 57].

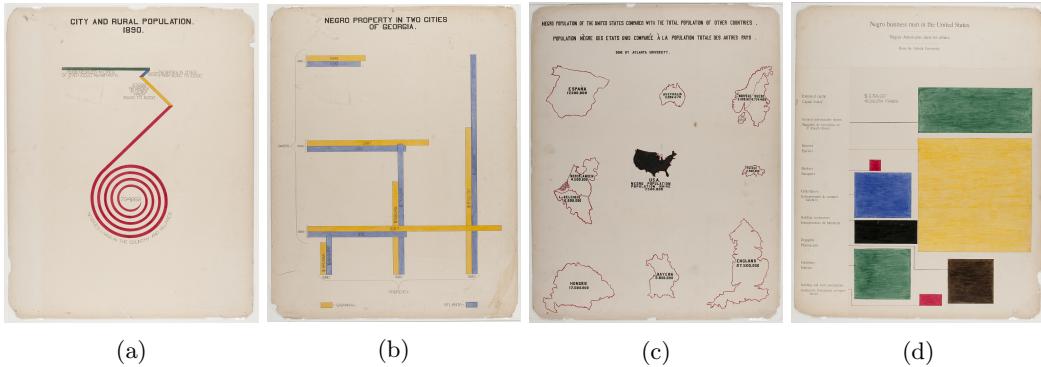


Figure 3: Du Bois’ data portraits[28] of post reconstruction Black American life exemplify that the fundamental characteristics of data visualization is that the visual elements vary in proportion to the source data. In figure 3a, the length of each segment maps to population; in figure 3b, the bar charts are intersected to show the number of property owners and how much they own in a given year; in figure 3c the countries are scaled to population size; and figure 3d 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 [1, 2, 81, 82]

95 We propose that the Du Bois visualizations in figure 3 are representative of the expressivity  
 96 a core architecture should allow a downstream library to express. This is because  
 97 while the Du Bois figures are not the common scatter, bar, or line plot [30], they conform  
 98 to the constraint that a visualization is a mapping from data to visuals. In figure 3c, Du  
 99 Bois combines a graphical representation where glyph size varies by population with a fig-  
 100 urative representation of those glyphs as the countries the data is from, which means that  
 101 the semantic and numerical properties of the data are preserved in the graph. The visual  
 102 representations in the Du Bois figures are in proportion to the quantitative data being rep-  
 103 resented, so the a chart is faithful according to Tufte’s Lie Principal[84]. The properties  
 104 of the representation match the properties of the information being represented, so they  
 105 are fairly understandable according to Norman’s Naturalness Principal[60]. As with the Du

<sup>106</sup> Bois data portraits, it is fundamental that any architecture tool we build ensures that the  
<sup>107</sup> data properties match the visual properties.

	<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 &amp; quantitative differences</i>

Figure 4: 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 [53] who reproduced it from *Making Maps: A Visual Guide to Map Design for GIS* [46]

<sup>108</sup> The visual properties were first codified by Bertin[9], who also described the types of  
<sup>109</sup> data that paired with the visual properties shown in figure 4. It is at this encoding level  
<sup>110</sup> that Mackinlay formalized expressiveness as a homomorphic mapping which preserves some  
<sup>111</sup> binary operator from one domain to another [52]. A functional dependency framework for  
<sup>112</sup> evaluating these equivariances was proposed by Sugibuchi et al [80], and an algebraic basis  
<sup>113</sup> for visualization design and evaluation was proposed by Kindlmann and Scheidegger[45].  
<sup>114</sup> Vickers et al. propose a category theory framework[87] that extends the equivariance checks  
<sup>115</sup> to layout, but is focused strictly on the design layer like the other mathematical frameworks.  
<sup>116</sup>

<sup>117</sup> The visual encodings are composed into the point, line, and area graphical marks, as shown in figure 4. Marks can be generalized to glyphs, which are graphical objects that

118 convey one or more attributes of the data entity mapped to it[57, 88] and minimally need  
119 to be differentiable from other visual elements [97]. As retinal variables and marks are the  
120 building blocks of most visualizations, they must be fully expressible in a framework for  
121 building visualization tools. By building visualization concepts into the core architecture,  
122 developers can incorporate assessments of the visualization, such as as quality metrics[10]  
123 or invariance [45] of visualizations with respect to graphical encoding choices.

## 124 2.4 Contribution

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

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

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

## 136 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 a structure preserving map from data  $\mathcal{E}$  to  $\mathcal{H}$

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

137 We model the data  $\mathcal{E}$ , graphic  $\mathcal{H}$ , and intermediate visual encoding  $\mathcal{V}$  stages of visual-  
138 ization as topological structures that encapsulate types of variables and continuity; by doing  
139 so we can develop implementations that keep track of both in ways that let us distribute  
140 computation while still allowing assembly and dynamic update of the graphic. To explain  
141 which structure the artist is preserving, we first describe how we model data(3.1), graph-  
142 ics(3.2), and intermediate visual characteristics (3.3) as fiber bundles. We then discuss the  
143 equivariant maps between data and visual characteristics (3.3.2) and visual characteristics  
144 and graphics (3.3.3) that make up the artist. *xi should maybe be moved down to artist but*  
145 *is more readable in graphics*

<sub>146</sub> **3.1 Data Space  $E$**

Building on Butler's proposal of using fiber bundles as a common data representation format for visualization data[16, 17], a fiber bundle is a tuple  $(E, K, \pi, F)$  defined by the projection map  $\pi$

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

<sub>147</sub> that binds the components of the data in  $F$  to the continuity represented in  $K$ . The fiber  
<sub>148</sub> bundle models the properties of data component types  $F$  (3.1.1), the continuity of records  
<sub>149</sub>  $K$  (3.1.3), the collections of records  $\tau$  (3.1.4), and the space  $E$  of all possible datasets with  
<sub>150</sub> these components and continuity.

<sub>151</sub> By definition fiber bundles are locally trivial[48, 73], meaning that over a localized neighbor-  
<sub>152</sub> hood we can dispense with extra structure on  $E$  and focus on the components and conti-  
<sub>153</sub> nuity. We use fiber bundles as the data model because they are inclusive enough to express  
<sub>154</sub> all the types of data described in section 2.2.

<sub>155</sub> **3.1.1 Variables: Fiber Space  $F$**

To formalize the structure of the data components, we use notation introduced by Spivak [75] 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  $\{T_0, \dots, T_n\} \in \mathbf{DT}$ , where  $\mathbf{DT}$  are the data types of the variables in the dataset. He then defines the single variable set  $\mathbb{U}_\sigma$

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

which is  $\mathbb{U}$  restricted to objects of type  $T$  bound to variable name  $c$ . Given  $\sigma$ , the fiber for a one variable dataset is

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

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

$$F = \mathbb{U}_{T_1} \times \dots \mathbb{U}_{T_i} \dots \times \mathbb{U}_{T_n} \tag{5}$$

which is equivalent to

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

<sub>156</sub> which allows us to decouple  $F$  into components  $F_i$ .

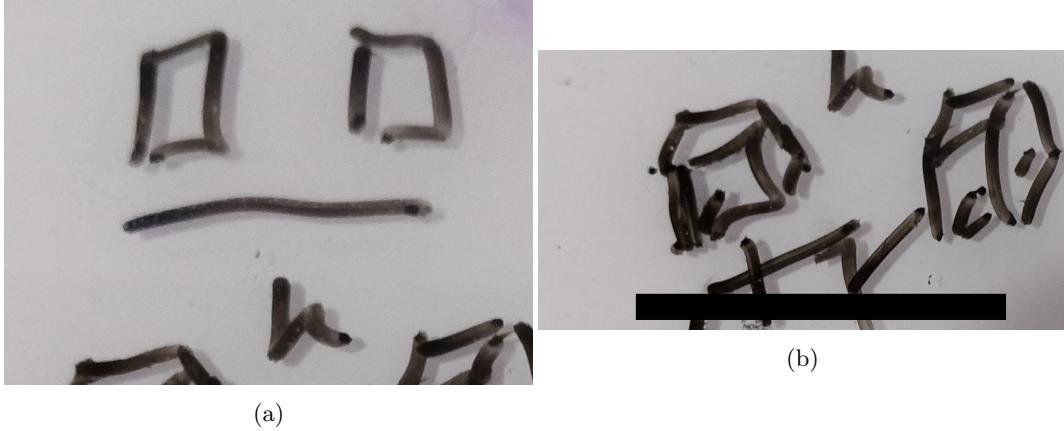


Figure 5: These two datasets have the same base space  $K$  but figure 5a has fiber  $F = \mathbb{R} \times \mathbb{R}$  which is (time, temperature) while figure 5b has fiber  $\mathbb{R}^+ \times \mathbb{R}^2$  which is (time, wind=(speed, direction))

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

$$\mathbb{U} = \mathbb{R} \times \mathbb{R}^+ \quad (7)$$

where the first component  $F_0$  is the set of values specified by ( $c_0 = \text{time}, T_0 = \text{datetime}, \mathbb{U}_\sigma = \mathbb{R}$ ) and  $F_1$  is specified by ( $c_1 = \text{temperature}, T_1 = \text{float}, \mathbb{U}_\sigma = \mathbb{R}^+$ ) 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

$$F = \mathbb{R}^+ \times \mathbb{R}^2 \quad (8)$$

such that  $F_1$  is specified by ( $c_1 = \text{wind}, T_1 = \text{wind}, \mathbb{U}_\sigma = \mathbb{R}^2$ ). As illustrated in figure ??, Spivak’s framework provides a consistent way to describe potentially complex components of the input data.

### 160 3.1.2 Measurement Scales: Monoid Actions

161 Implementing expressive visual encodings requires formally describing the structure on the  
 162 components of the fiber, which we define as the action of a monoid on the component. While  
 163 structure on a set of values is often described algebraically as operations or through the  
 164 actions of a group, for example Steven’s scales [77], we generalize to monoids to support more  
 165 component types. Monoids are also commonly found in functional programming because  
 166 they specify compositions of transformations [78, 96].

A monoid [55]  $M$  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 [3, 72] of  $M_i$  is a set  $F_i$  with an action

- :  $M \times F_i \rightarrow F_i$  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

$$M = M_0 \times \dots \times M_i \times \dots \times M_n \quad (9)$$

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

169 Steven's described the measurement scales[47, 77] in terms of the monoid actions on  
170 the measurements: nominal data is permutable, ordinal data is monotonic, interval data  
171 is translatable, and ratio data is scalable [89]. For example, given the interval scale fiber  
172 component ( $c = \text{temperature}$ ,  $T = \text{float}$ ,  $\mathbb{U}_\sigma = \mathbb{R}$ ):

- 173 • monoid operator addition  $* = +$
- 174 • monoid operations:  $f : x \mapsto x + 1$ ,  $g : x \mapsto x + 2$
- 175 • monoid action operator composition  $\bullet = \circ$

then the translation monoid actions on temperature satisfy the condition

$$\begin{array}{ccc} \mathbb{R} & & \\ \downarrow_{x+1^\circ} & \searrow^{(x+1^\circ)\circ(x+2^\circ)} & \\ \mathbb{R} & \xrightarrow{x+2^\circ} & \mathbb{R} \end{array} \quad (10)$$

176 where  $1^\circ$  and  $2^\circ$  are valid distances between two temperatures  $x$ .

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

- 180 1.  $f(\text{rain}) = \text{drizzle}$ ,
- 181 2.  $f(\text{drizzle}) = \text{mist}$
- 182 3.  $f(\text{mist}) = \text{mist}$

183 is order preserving such that  $\text{mist} \leq \text{drizzle} \leq \text{rain}$  but has no inverse since  $\text{drizzle}$  and  
184  $\text{mist}$  go to the same value  $\text{mist}$ . Therefore order preserving maps do not form a group, and  
185 instead we generalize to monoids to support partial order component types. Defining the  
186 monoid actions on the components serves as the basis for identifying the invariance[45] that  
187 must be preserved in the visual representation of the component.

### 188 3.1.3 Continuity: Base Space $K$

189 The base space  $K$  is way to express the connectivity of the records. This is assumed in the  
190 choice of visualization, for example a line plot implies 1D continuous data and a scatter plot  
191 implies discrete records, but an explicit representation allows for verifying that the topology  
192 of the graphic representation is equivalent to the topology of the data.

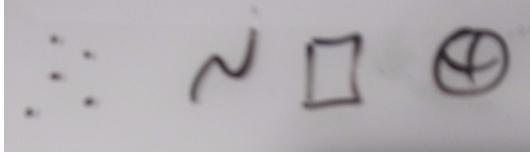


Figure 6: The topological base space  $K$  encodes the connectivity of the data space, for example if the data is independent points or a map or on a sphere

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

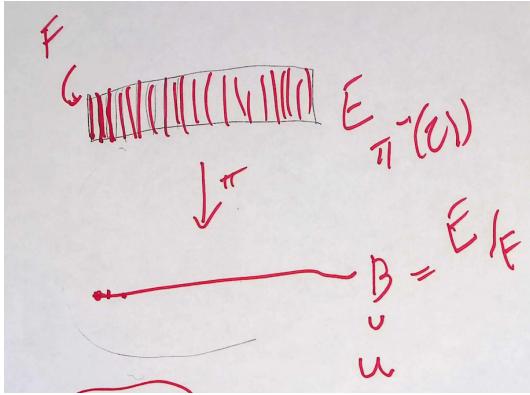


Figure 7: The base space  $E$  is divided into fiber segments  $F$ . The base space  $K$  acts as an index into the records in the fibers. **this figure might be good all the way up top to lay out the components of fb**

195     Formally  $K$  is the quotient space [64] of  $E$  meaning it is the finest space[6] such that  
 196     every  $k \in K$  has a corresponding fiber  $F_k$ [64]. In figure 7,  $E$  is a rectangle divided by  
 197     vertical fibers  $F$ , so the minimal  $K$  for which there is always a mapping  $\pi : E \rightarrow K$  is the  
 198     line.

As with fibers and monoids, we can decompose the total space into components  $\pi : E_i \rightarrow K$  where

$$\pi : E_1 \oplus \dots \oplus E_n \oplus \dots \oplus E_n \rightarrow K \quad (11)$$

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

201     The datasets in figure 8 have the same fiber of (temperature, time). In figure 8a the  
 202     fibers lie over discrete  $K$  such that the records in the datasets in the fiber bundles are  
 203     discrete. The same fiber in figure 8b lies over a continuous interval  $K$  such that the records  
 204     are samples from a continuous function defined on  $K$ . By encoding this continuity in the  
 205     model as  $K$  the data model now explicitly carries information about its structure such  
 206     that the implicit assumptions of the visualization algorithms are now explicit. This in turn  
 207     allows for building algorithms that can work with distributed or streaming data, since it  
 208     has a common data access interface with a promise that the data exists.

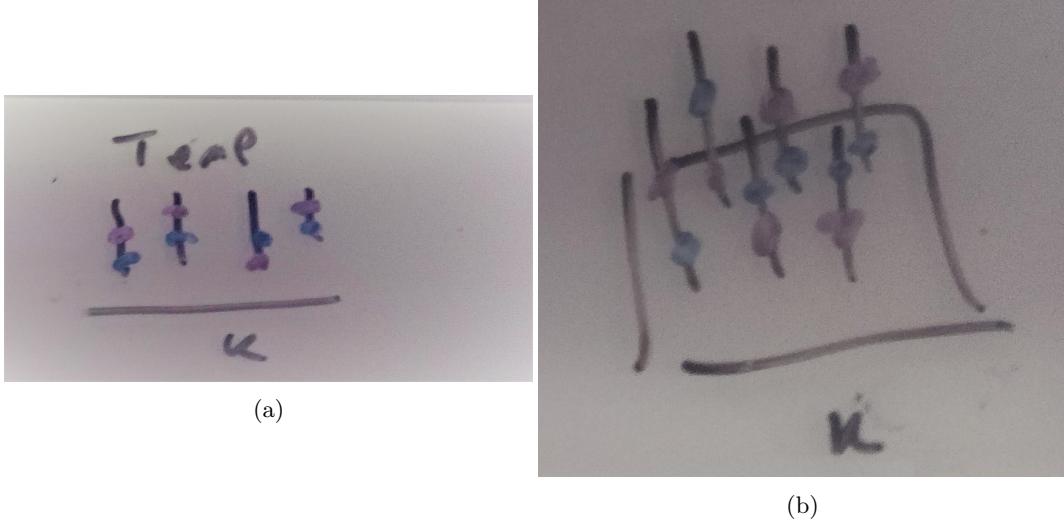


Figure 8: These two datasets have the same (time, temperature) fiber. In figure ?? the total space  $E$  is discrete over points  $k \in K$ , meaning the records in the fiber are also discrete. In figure ??  $E$  lies over the continuous interval  $K$ , meaning the records in the fiber are sampled from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

209 **3.1.4 Data: Sections  $\tau$**

A set of records is a section  $\tau : K \rightarrow E$  of the fiber bundle. For example, in the special case of a table [75],  $K$  is a set of row ids,  $F$  is the columns, and the section  $\tau$  returns the record at a given key in  $K$ . In general, for any fiber bundle, there is always a map

$$\begin{array}{ccc} F & \xhookrightarrow{\quad} & E \\ \pi \downarrow & \nearrow \tau & \\ K & & \end{array} \tag{12}$$

such that  $\pi(\tau(k)) = k$ . There can be many sections  $\tau$  and the space of all global sections is  $\Gamma(E)$ . Assuming a trivial fiber bundle  $E = K \times F$ , the section is

$$\tau(k) = (k, (g_{F_0}(k), \dots, g_{F_n}(k))) \tag{13}$$

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 formulate  $\tau$  as

$$\tau = (\tau_0, \dots, \tau_i, \dots, \tau_n) \tag{14}$$

210 where each section  $\tau_i$  is a variable or set of variables.

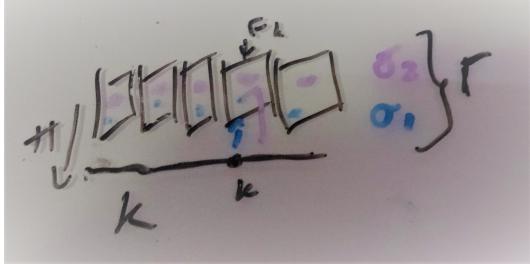


Figure 9: Fiber (time, temperature) with an interval  $K$  basespace. The sections  $\tau_i$  and  $\tau_j$  are constrained such that the time variable must be monotonic, which means each section is a timeseries of temperature values. They are included in the global set of sections  $\tau_1, \tau_2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

In the example in figure 9, the fiber is *(time, temperature)* as described in figure 5 and the base space is the interval  $K$ . The section  $\tau_i$  resolves to a series of monotonically increasing in time records of (time, temperature) values. Section  $\tau_j$  returns a different timeseries of (time, temperature) values. Both sections are included in the global set of sections  $\tau_i, \tau_j \in \Gamma(E)$ .

The section can be any instance of a data container, for example a numpy array[36], a pandas series or dataframe[67], or an xarray[41]. A univariate numpy array that stores an image is a section of a fiber bundle where  $K$  is a 2D continuous plane and the  $F$  is  $(\mathbb{R}^3, \mathbb{R}, \mathbb{R})$  where  $\mathbb{R}^3$  is color, and the other two components are the x and y positions of the sampled data in the image. A series could store the values of  $\tau_i$  and a second series could be  $dsection_j$ . We could also flatten the fiber to hold two temperature series, such that a section would be an instance of a dataframe with a time column and two temperature columns. While the series and dataframe explicitly have a time index column, they are components in our model and the index is always assumed to be random keys. An instance of an xarray would also be a  $\tau$  for example if the data is a temperature field than the  $K$  could be a continuous volume and the components would be the temperature and the time, latitude, and longitude the measurements were sampled at. As with the dataframe, the semantic index labels are considered components and the indices are instead assumed to be random. A section can also be an instance of a distributed data container, such as a dask array [69]. As with the other containers,  $K$  and  $F$  are defined in terms of the index and dtypes of the components of the array. The section can remain curried until render since the model is functional, in keeping with dasks compute graph architecture. Our model provides a common interface to these widely used data containers without sacrificing the semantic structure embedded in each container.

### 3.1.5 Sheaf and Stalk

To support working with a subset of data, we can take the sheaf  $\mathcal{O}(E)$  as input into the artist.  $E$  restricted over a small enough neighborhood  $U \subset K$  is a locally trivial bundle over  $U$ [48]. The sheaf  $\mathcal{O}(E)$  is the localized section of fibers  $\iota^*\tau : U \rightarrow \iota^*E$

$$\begin{array}{ccc}
 \iota^*E & \xleftarrow{\iota^*} & E \\
 \pi \downarrow \lrcorner \iota^*\tau & & \pi \downarrow \lrcorner \tau \\
 U & \xleftarrow{\iota} & K
 \end{array} \tag{15}$$

239 pulled back over the neighborhood  $U$  via the inclusion map  $\iota : U \rightarrow K$ . The localized section  
240 is the germ  $\xi^*\tau$ . The neighborhood of points  $k_i$  surrounding the point  $k$  the sheaf lies over  
241 is the stalk  $\mathcal{F}_b$  [73, 76].

242 While  $E$  is only the fiber  $F_k$  over a specific  $k$ , the stalk includes the local behavior of the  
243 section at  $k$  which can include information about nearby records. In practice, often the only  
244 information needed from the stalk is some finite number  $n$  of derivatives, which is captured  
245 by the jet bundle  $\mathcal{J}^n$  [44, 58] with  $\mathcal{J}^0(E) = E$ . In this work, we at most need  $\mathcal{J}^2(E)$  which  
246 is the value at  $\tau$  and its first and second derivatives. The sheaf facilitates interactions such  
247 as zooming or updating the graphic based on zooming data; the jet ensures that the artist  
248 only needs a single record on  $k$  to render a piece of a graphic. This allows for the creation  
249 of highly concurrent and possibly distributed artists.

## 250 3.2 Graphic: $H$

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

### 256 3.2.1 Idealized Display $D$

257 To fully specify the visual characteristics of the image, we construct a fiber  $D$  that is an  
258 infinite resolution version of the target space. Typically  $H$  is trivial and therefore sections  
259 can be thought of as mappings into  $D$ .

In this work, we assume a 2D opaque image  $D = \mathbb{R}^5$  with elements

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

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

### 264 3.2.2 Continuity of the Graphic $S$

265 Since we propose that data topology  $K$  is by some measure preserved in the graphic, we  
266 introduce a graphic topology  $S$  to encode the carried over structure. These topologies are not  
267 always identical because the underlying topology  $S$  of a graphic may need more dimensions  
268 than the data topology  $K$ . In a typical 2D display (ignoring depth), visible elements need to  
269 have 2D extent so that for example a scatter marker has an area greater than 0 or a line is  
270 not infinitely thin.

Since  $K$  with fewer than two dimensions need to be thickened, we define the mapping  
from graphic  $S$  to data  $K$

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



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 heatmap has the same continuity in the graphic  $S$  as in the data  $K$ . **add  $\alpha, \beta$  coordinates to figures**

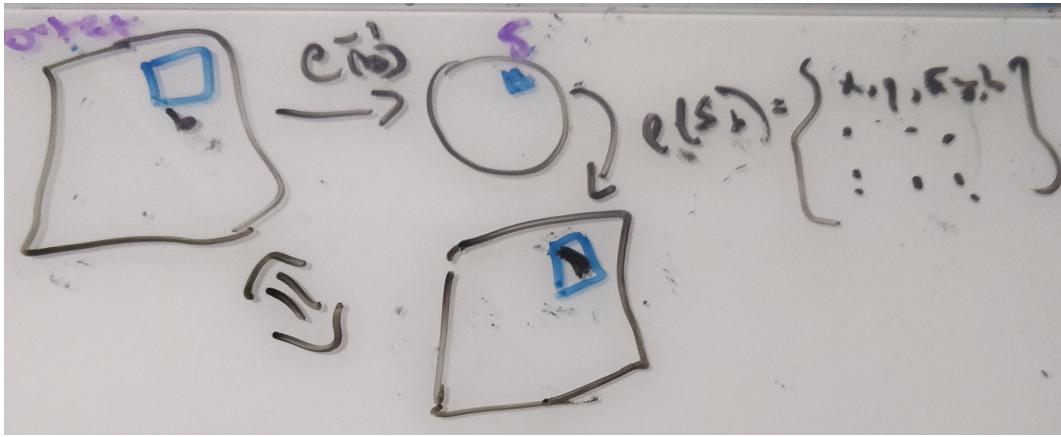


Figure 11: To render a graphic, a pixel  $p$  is selected in the display space, which is defined in the same coordinates as the  $x$  and  $y$  components in  $D$ . The inverse mapping  $\rho_{xy}(p)$  returns a region  $S_p \subset S$ .  $\rho(S_p)$  returns the list of elements  $(x, y, r, g, b) \in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

as the deformation retraction [68]  $\xi : S \rightarrow K$  that goes from a region  $s \in S_k$  to its associated point  $s$ , such that when  $\xi(s) = k$ ,  $\xi^*\tau(s) = \tau(s)$ . While dimensions can be added to  $S$ , it retains the same continuity as  $K$ .

When  $K$  is discrete points and the graphic is a scatter plot, each point  $k \in K$  corresponds to a 2D disk  $S_k$  as shown in figure 10. In the case of 1D continuous data and a line plot, the region  $\beta$  over a point  $\alpha_i$  specifies the thickness of the line in  $S$  for the corresponding  $\tau$  on  $k$ . The heatmap has the same continuity in data space and graphic space such that no extra dimensions are needed in  $S$ . The mapping function  $\xi$  provides a way to identify the part of the visual transformation that is specific to the the connectivity of the data rather than the values; for example it is common to flip a matrix when displaying an image.

281    **3.2.3    Renderering  $\rho$**

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

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

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

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

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

291    As shown in figure 11, a pixel  $p$  in the output space is selected and inverse mapped into  
 292    the corresponding region  $S_p \subset S$ . This triggers a lookup of the  $\rho$  over the region  $S_p$ , which  
 293    yields the set of elements in  $D$  that specify the  $(r, g, b)$  values corresponding to the region  
 294     $p$ . The color of the pixel is then obtained by taking the integral of  $\rho_{rgb}(S_p)$ . In general,  $\rho$   
 295    is an abstraction of rendering specifications, for example PDF[11], SVG[63], or an openGL  
 296    scene graph[22] or rendering engines such as cairo[20] and AGG[5]. Implementation of  $\rho$  is  
 297    out of scope for this work.

298    **3.3    Artist**

299    The artist is the function that converts data into graphics; its name is taken from the  
 300    analogous part of Matplotlib[43] that builds visual elements to pass off to the renderer. The  
 301    artist  $A$  is a mapping from  $E$  padded with data from  $E' = \mathcal{J}(E)$  to a graphic that is a  
 302    section  $\rho \in \Gamma(H)$

$$\begin{array}{ccccc} E' & \xrightarrow{\nu} & V & \xleftarrow{\xi^*} & \xi^*V \xrightarrow{Q} H \\ & \searrow \pi & \downarrow \pi & \downarrow \xi^*\pi & \swarrow \pi \\ & & K & \xleftarrow{\xi} & S \end{array} \quad (21)$$

303    which due to using the jet is point wise such that the input can be  $\tau(k)$ . The encoders  
 304     $\nu : E' \rightarrow V$  converts the data components to visual components(3.2.2). The continuity  
 305    map  $\xi : SK$  then pulls back the visual bundle  $V$  over  $S$ (3.3.2). Then the assembly function  
 306     $Q : \xi^*V \rightarrow H$  composites the fiber components of  $\xi^*V$  into a graphic in  $H$ (3.3.3). This  
 307    functional decomposition of the visualization artists allows us to specify what are the re-  
 308    sponsibilities of each function in a fully constrained way. In turn, this allows for building  
 309    reusable components at each stage of the transformation.

310    **3.3.1 Visual Fiber Bundle  $V$**

311    We introduce a visual bundle  $V$  to store the visual representations the artist needs to  
 312    composite into a graphic. The visual bundle  $(V, K, \pi, P)$  has section  $\mu : V \rightarrow K$  that  
 313    resolves to a visual variable in the fiber  $P$ . The visual bundle  $V$  is the latent space of  
 314    possible parameters of a visualization type, such as a scatter or line plot. We define  $P$   
 315    in terms of the parameters of a visualization libraries compositing functions; for example  
 316    table 1 is a sample of the fiber space for Matplotlib [42].

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

317    A section  $\mu$  is a tuple of visual values that specifies the visual characteristics of a part of  
 318    the graphic. For example, given a fiber of  $\{xpos, ypos, color\}$  one possible section could be  
 319     $\{.5, .5, (255, 20, 147)\}$ . The  $\text{codomain}(\nu_i)$  determines the monoid actions on  $\mu_i$ . These fiber  
 320    components are implicit in the library, by making them explicit as components of the fiber  
 321    we can build consistent definitions and expectations of how these parameters behave.

322    **3.3.2 Visual Encoders  $\nu$**

As introduced in section ??, there are many ways to encode data visually. We define the visual transformers  $\nu$  as the set of independent conversion functions

$$\{\nu_0, \dots, \nu_n\} : \{\tau_0, \dots, \tau_n\} \mapsto \{\mu_0, \dots, \mu_n\} \quad (22)$$

where  $\nu_i : \tau_i \mapsto \mu_i$  is an equivariant map such that there is a monoid homomorphism from  $F_i$  to  $v_{fiber_i}$ . As mentioned in section 3.1.2, we choose monoid actions as the basis for equivariance because they define the structure on the fiber components. A validly constructed  $\nu$  is one where the diagram of the monoid transform  $m$

$$\begin{array}{ccc} E_i & \xrightarrow{\nu_i} & V_i \\ m_x \downarrow & & \downarrow m_v \\ E_i & \xrightarrow{\nu_i} & V_i \end{array} \quad (23)$$

```

[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,  $\nu$  maps the strings to the emojis. For  $\nu$  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.

commutes such that  $\nu_i(m_x(E_i)) = m_v(\nu_i(E_i))$ . This equivariance constraint yields guidance on what makes for an invalid transform. For example, the conversion  $\nu_i(x) = .5$  does not commute under translation monoid action  $t(x) = x + 2$

$$\nu(t(x + 2)) \stackrel{?}{=} \nu(x) + \nu(2) \quad (24)$$

$$.5 \neq .5 + .5 \quad (25)$$

On the other hand figure 12 illustrates a valid  $\nu$  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. To preserve ordinal and partial order monoid actions,  $\nu$  must be a monotonic function such that given  $x_1, x_2 \in E_i$ , if  $delement_1 \leq x_2$  then  $\nu(x_1) \leq \nu(x_2)$ . For interval scale data,  $\nu$  is equivariant under translation monoid actions if  $\nu(x + c) = \nu(x) + \nu(c)$ . For ratio data, there must be equivalent scaling  $\nu(xc) = \nu(x)\nu(c)$ . These constraints can be embedded into our artist such that the  $\nu$  functions are equivariant; they also provide guidance on constructing new equivariant  $\nu$  functions.

### 3.3.3 Graphic Assembler $Q$

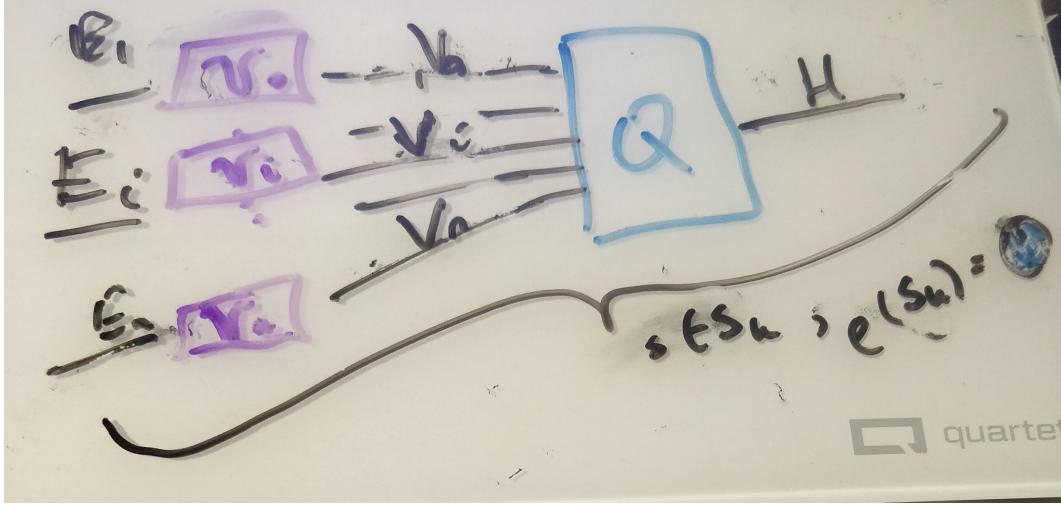


Figure 13:  $\nu$  functions convert data  $\tau_i$  to visual characteristics  $\mu_i$ , then  $Q$  assembles  $\mu_i$  into a graphic  $\rho$  such that there is a map  $\xi$  preserving the continuity of the data.  $\rho$  applied to a region of connected components  $S_j$  generates a part of a graphic, for example the point graphical mark.

As shown in figure 13, the assembly function  $Q$  combines the fiber  $F_i$  wise  $\nu$  transforms into a graphic in  $H$ . Together,  $\nu$  and  $Q$  are a map-reduce operation: map the data into their visual encodings, reduce the encodings into a graphic. As with  $\nu$  the constraint on  $Q$  is that for every monoid action on the input  $\mu$  there is a corresponding monoid action on the output  $\rho$ .

Since we define the equivariant map as  $Q : \mu \mapsto \rho$ , we define an action on the subset of graphics  $Q(\Gamma(V)) \in \Gamma(H)$  that  $Q$  can generate. We then define the constraint on  $Q$  such

339 that if  $Q$  is applied to  $\mu, \mu'$  that generate the same  $\rho$  then the output of both sections acted  
 340 on by the same monoid  $m$  must be the same.

Lets call the visual encodings  $\Gamma(V) = X$  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

$$Q(\mu) = Q(\mu') \implies Q(m \circ \mu) = Q(m \circ \mu') \quad (26)$$

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

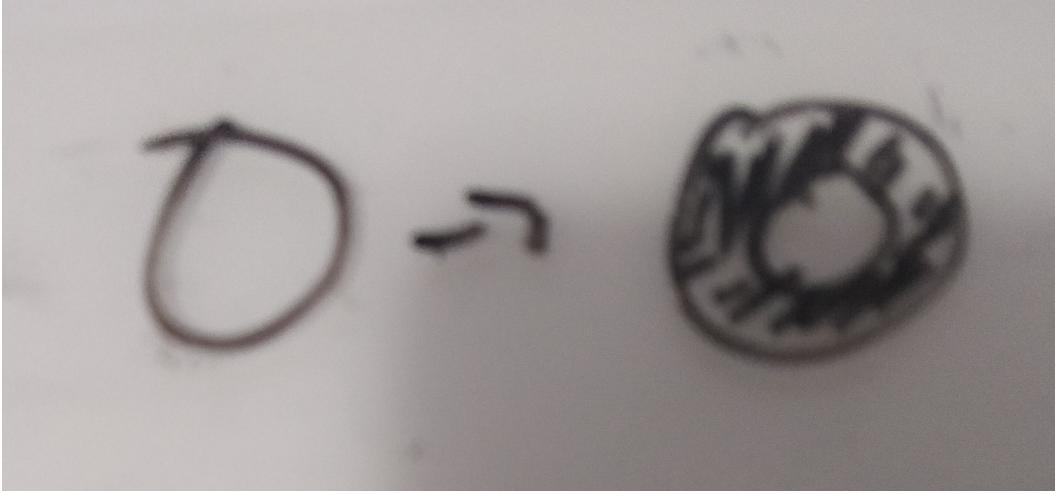


Figure 14: These two glyphs are generated by the same  $Q$  function, but differ in the value of the edge thickness parameter  $\mu_i$ . A valid  $Q$  is one where a shift in  $\mu_i$  is reflected in the glyph generated by  $\rho$ .

344 The glyph in figure 14 has the following characteristics  $P$  specified by  $(xpos, ypos, color, thickness)$   
 345 such that one section is  $\mu = (0, 0, 0, 1)$  and  $Q(\mu) = \rho$  generates a piece of the thin hollow  
 346 circle. The equivariance constraint on  $Q$  is that the action  $m = (e, e, e, x + 2)$ , where  $e$  is  
 347 identity, applied to  $\mu$  such that  $\mu' = (e, e, e, 3)$  has an equivalent action on  $\rho$  that causes  
 348  $Q(\mu')$  to be equivalent to the thicker circle in figure 14.

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:

$$J = \{j \in K \text{ s. t. } \exists \gamma \text{ s.t. } \gamma(0) = k \text{ and } \gamma(1) = j\} \quad (27)$$

where the path[26]  $\gamma$  from  $k$  to  $j$  is a continuous function from the interval  $[0,1]$ . We define the glyph as the graphic generated by  $Q(S_j)$

$$H \xrightleftharpoons[\rho(S_j)]{} S_j \xrightleftharpoons[\xi^{-1}(J)]{\xi(s)} J_k \quad (28)$$

349 such that for every glyph there is at least one corresponding section on  $K$ . This is in  
 350 keeping with the definition of glyph as any differentiable element put forth by Ziemkiewicz  
 351 and Kosara[97]. The primitive point, line, and area marks[9, 21] are specially cased glyphs.

352 **3.3.4 Assembly Q**

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

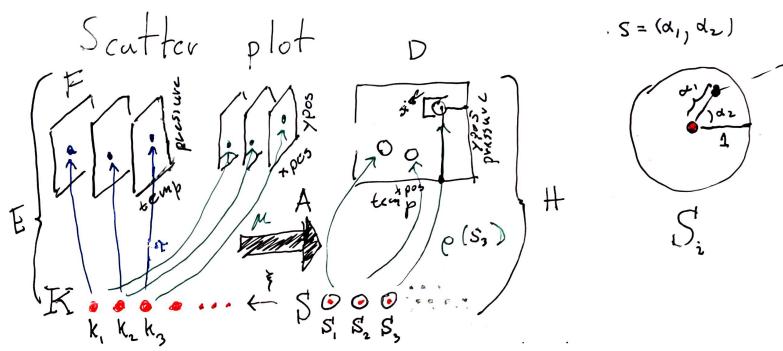


Figure 15: The data is discrete points (temperature, time). Via  $\nu$  these are converted to (xpos, ypos) and pulled over discrete  $S$ . These values are then used to parameterize  $\rho$  which returns a color based on the parameters (xpos,ypos) and position  $\alpha, \beta$  on  $S_k$  that  $\rho$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

The scatter plot in figure ?? 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:

$$x = size \bullet \alpha \bullet \cos(\beta) + xpos \quad (29)$$

$$y = size \bullet \alpha \bullet \sin(\beta) + ypos \quad (30)$$

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

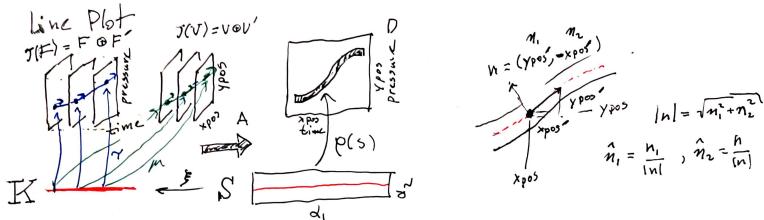


Figure 16: The line fiber (*time, temp*) is thickened with the derivative (*time', temperature'*) because that information will be necessary to figure out the tangent to the point to draw a thick line. This is because the line needs to be pushed perpendicular to the tangent of (xpos, ypos). **this is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

$$|n| = \sqrt{n_1^2 + n_2^2} \quad (31)$$

such that the normal is

$$\hat{n}_1 = \frac{n_1}{|n|}, \quad \hat{n}_2 = \frac{n_2}{|n|} \quad (32)$$

which yields components of  $\rho$

$$x = xpos(\xi(\alpha)) + \beta \hat{n}_1(\xi(\alpha)) \quad (33)$$

$$y = ypos(\xi(\alpha)) + \beta \hat{n}_2(\xi(\alpha)) \quad (34)$$

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

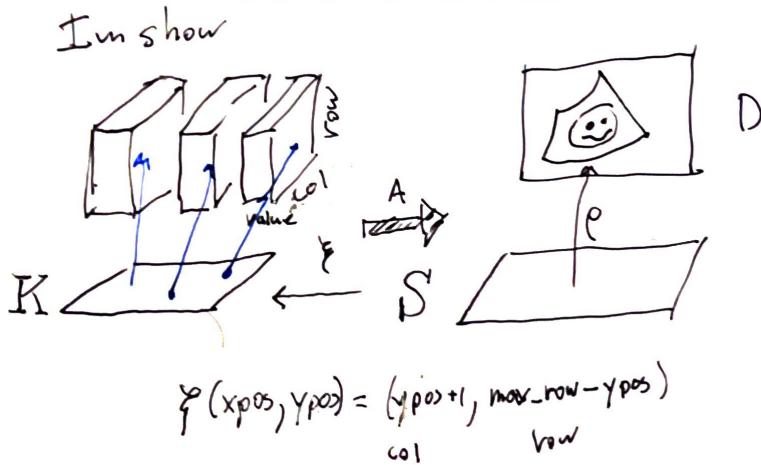


Figure 17: The only visual parameter a image requires is color since  $\xi$  encodes the mapping between position in data and position in graphic.

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

$$R = R(\xi(\alpha, \beta)) \quad (35)$$

$$G = G(\xi(\alpha, \beta)) \quad (36)$$

$$B = B(\xi(\alpha, \beta)) \quad (37)$$

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

361    **3.3.5 Assembly factory  $\hat{Q}$**

362    The graphic base space  $S$  is not accessible in many architectures, including Matplotlib,  
 363    because the rendering is tightly interlaced with the graphical design; instead we can construct  
 364    a factory function  $\hat{Q}$  over  $K$  that can build a  $Q$ . As shown in eq 21,  $Q$  is a bundle map  
 365     $Q : \xi^*V \rightarrow H$  where  $\xi^*V$  and  $H$  are both bundles over  $S$ .

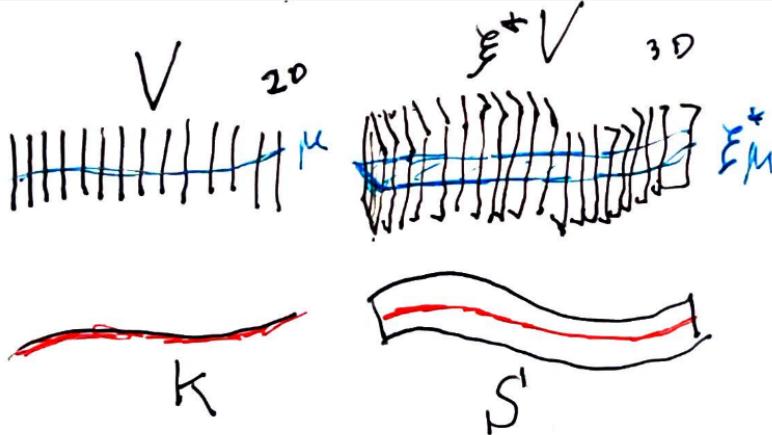


Figure 18: The pullback of the visual bundle  $\xi^*V$  is the replication of a  $\mu$  over all points  $s$  that map back to a single  $k$ . Because the  $\mu$  is the same, we can construct a  $\hat{Q}$  on  $\mu$  over  $k$  that will fabricate the  $Q$  for the equivalent set of  $s$  associated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

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

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

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

383    **3.3.6 Composition of Artists: +**

384    To build graphics that are the composites of multiple artists, we define a simple addition  
 385    operator that is the disjoint union of fiber bundles  $E$ . For example, in figure 19 the scatter

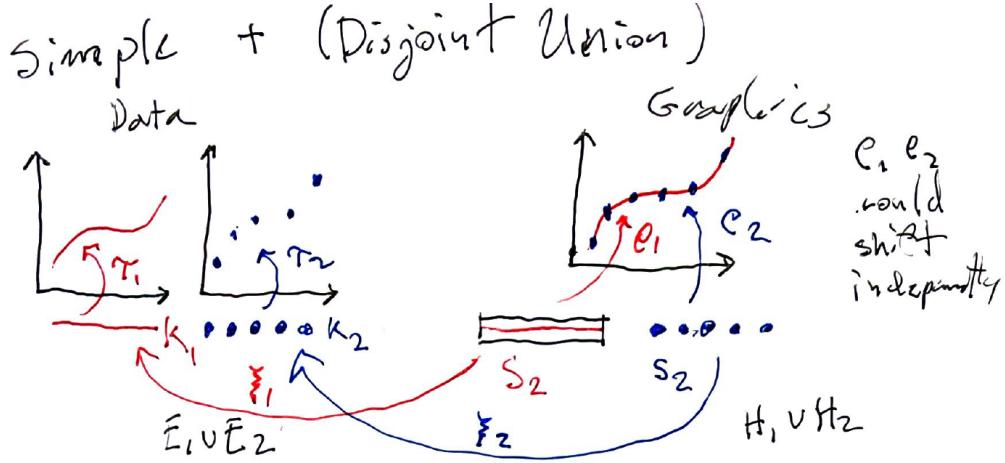


Figure 19:  $\tau_1$  and  $\tau_2$  are distinct datasets passed through artists  $A_1$  and  $A_2$  to generate graphics  $\rho_1$  and  $\rho_2$ . These graphics happen to be rendered to the same image, but otherwise have no intrinsic link.

386 plot  $E_1$  and the line plot  $E_2$  have different  $K$  that are mapped to separate  $S$ . To fully  
 387 display both graphics, the composite graphic  $A_1 + A_2$  needs to include all records on both  
 388  $K_1$  and  $K_2$ , which are the sections on the disjoint union  $K_1 \sqcup K_2$ . This in turn yields disjoint  
 389 graphics  $S_1 \sqcup S_2$  rendered to the same image. Constraints can be placed on the disjoint  
 390 union such as that the fiber components need to have the same  $\nu$  position encodings or that  
 391 the position  $\mu$  need to be in a specified range. There are situations where  $K_2 \hookrightarrow K_1$  that  
 392 underpin more complex, especially interactive, visualizations; these cases require defining a  
 393 more complex addition operator that is out of scope for this work.

### 394 3.3.7 Equivalence class of artists $A'$

395 It is impractical to implement an artist for every single graphic; instead we implement the  
 396 equivalence class of artists  $\{A \in A' : A_1 \equiv A_2\}$  which is essentially an artist where the  $\nu$   
 397 functions have not yet been input. Equivalent artists have the same fiber bundle  $V$  and same  
 398 assembly function  $Q$  but act on different sections  $\mu$ . To further simplify implementation,  
 399 we identify a minimal  $P$  associated with each  $A'$  that defines what visual characteristics of  
 400 the graphic must originate in the data such that the graphic is identifiable as a given chart  
 401 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**

402 For example, a scatter plot of red circles is the output of one artist, a scatter plot of  
 403 green squares the output of another, as shown in figure ???. These two artists are equivalent  
 404 since their only difference is in the literal visual encodings (color, shape). Shape and color  
 405 could also be defined in  $Q$  but the position must come from the fiber  $P = (xpos, ypos)$  since  
 406 fundamentally a scatter plot is the plotting of one position against another[30]. We also use

407 this criteria to identify derivative types, for example the bubble chart[84] is a type of scatter  
 408 where by definition the glyph size is mapped from the data. The criteria for equivalence  
 409 class membership serves as the basis for evaluating invariance[45].

410 **3.4 Making the fiber bundle computable**

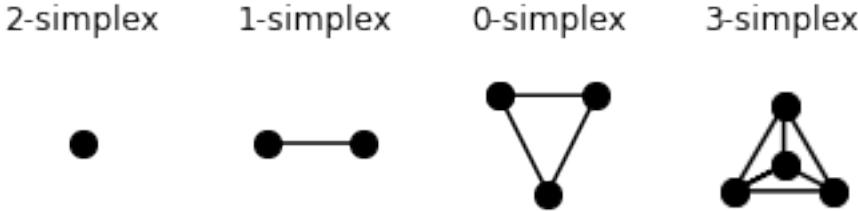


Figure 21: Simplices can encode the connectivity of the data, from fully disconnected (0 simplex) records to all records are connected to at least 3 others

411 One way of expressing the connectivity of records in a dataset is to implement  $K$  as a  
 412 simplicial complex, which is a set of simplices such as those shown in figure 21. The  
 413 advantage of triangulation is that it is general enough to work for more complex topology  
 414 based visualization methods [39] while also providing a consistent interface of vertices, edges,  
 415 and faces for  $\xi$  to map into. When triangulated, the simplices encode the continuity in the  
 416 data

<b>simplex</b>	<b>continuity</b>	$\tau$
vertex	discrete	$\tau(k)$
edge	1D	$\tau(k, j)$
face	2D	$\tau(k, j, )$

Table 2

417 such that each section is bound to a simplex  $k \in K$ . As shown in table 2, in a 1D  
 418 continuous spaces each  $\tau$  lies distance  $j$  along edge  $k$ , while in a 2D continuous space each  
 419  $\tau$  lies at coordinate  $j$ ,on the face  $k$ . This is directly analogous to indexing to express  
 420 connectivity in N-D arrays, while also natively supporting graphs and trees as they are  
 421 simplicial complexes of nodes and edges. Path connected components are then sections  
 422 where edges or faces meet.

423 In general simplicial complexes, in these prototypes we implement graphs.

<sup>424</sup> 4 Prototype Implementation: Matplottoy

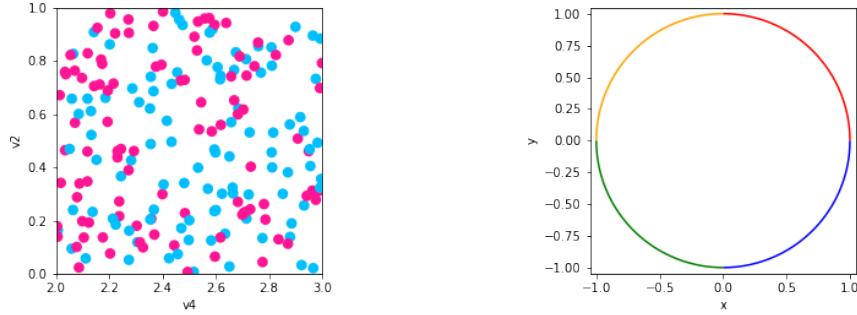


Figure 22: Scatter plot and line plot implemented using prototype artists and data models, building on Matplotlib rendering.

<sup>425</sup> To prototype our model, we implemented the artist classes for the scatter and line plots  
<sup>426</sup> shown in figure 22 because they differ in every attribute: different visual channels  $\nu$  that  
<sup>427</sup> composite to different marks  $Q$  with different continuities  $\xi$ . We make use of the Matplotlib  
<sup>428</sup> figure and axes artists [42, 43] so that we can initially focus on the data to graphic trans-  
<sup>429</sup> formations.

<sup>430</sup> To generate the images in figure 22, we instantiate `fig`, `ax` artists that will contain the  
<sup>431</sup> 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>
--	---

---

432 We then add the  $A'=\text{Point}$  and  $A'=\text{Line}$  artists that construct the scatter and line  
 433 graphics. The arguments to the artist are the data  $E=\text{data}$  that is to be plotted and the  
 434 aesthetic configuration  $\nu=\text{transforms}$ . We implement the artists as equivalence classes  $A'$   
 435 because it would be impractical to implement a new artist for every aesthetic setting, such  
 436 as one artist for red lines and another for green.

#### 437 4.1 Artist Class $A'$

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

---

```

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

```

---

444 The data is fetched in section  $\tau$  via a `view` method on the data because the input to the  
 445 artist is a section on  $E$ . The return `view` object has a `get` method to support querying for  
 446 components that are not in  $F$  which we exploit to support parameters in the visual fiber  
 447 that are not bound to fiber components in  $F$ . The  $\nu$  functions are then applied to the data  
 448 to generate the  $\mu=\text{visual}$  input to  $Q$ . An explicit  $\xi$  is not implemented since that would  
 449 mean copying a single  $\mu$  on  $k$  to all the associated  $s$ , as illustrated in figure 10, and that is  
 450 unnecessary overhead for these scatter and line plots. In  $\hat{Q}=\text{assemble}$  the artist generates  
 451 instructions for the render by setting the attributes that are related to the graphic. These

452 are the settings that would have to be serialized in order to recreate a static version of the  
 453 graphic. Although `assemble` could be implemented outside the class such that it returns  
 454 an object the artist could then parse to set attributes, the attributes are directly set here  
 455 to reduce indirection. The  $\nu$  functions could be evaluated in this function to avoid passing  
 456 over  $K$  twice but are not done so here to demonstrate the separability of  $\nu$  and  $\hat{Q}$ . The last  
 457 step in the artist function is handing itself off to the renderer.

458 The `Point` artist builds on `collection` artists because collections are optimized to ef-  
 459 ficiently draw a sequence of primitive point and area marks. In this prototype, the scatter  
 460 marker shape is fixed as a circle, and the only visual fiber components are  $x$  and  $y$  position,  
 461 size, and the facecolor of the marker.

---

```

1  class Point(mcollections.Collection):
2      def __init__(self, data, transforms, *args, **kwargs):
3          super().__init__(*args, **kwargs)
4          self.data = data
5          self.transforms = transforms
6
7      def assemble(self, visual):
8          # construct geometries of the circle marks in visual coordinates
9          self._paths = [mpath.Path.circle(center=(x,y), radius=s)
10             for (x, y, s) in zip(visual['x'], visual['y'], visual['s'])]
11          # set attributes of marks, these are vectorized
12          # circles and facecolors are lists of the same size
13          self.set_facecolors(visual['facecolors'])
14
15      def draw(self, renderer, *args, **kwargs):
16          # query data for a vertex table  $K$ 
17          view = self.data.view()
18          visual = {p: encoder(view.get(f, None)) for
19                  p, (f, encoder) in self.transforms.items()}
20          self.assemble(visual)
21          # call the renderer that will draw based on properties
22          super().draw(renderer, *args, **kwargs)
```

---

462 The `view` method repackages the data as a fiber component indexed table of vertices, as  
 463 described in section 3.4; even though the `view` is fiber indexed, each vertex at an index  
 464  $k$  has corresponding values in section  $\tau(k_i)$  such that all the data on one vertex maps to  
 465 one marker. To ensure the integrity of the section, `view` must be atomic, meaning that the  
 466 values cannot change after the method is called in `draw` until a new call in `draw`. This table  
 467 is converted to a table of visual variables. It is then passed into `assemble`, where it is used  
 468 to individually construct the vector path of each circular marker with center  $(x, y)$  and size  
 469  $x$  and set the colors of each circle. Since `view` returns a  $\tau$  all these operations could be  
 470 applied on a section on one  $k$  or a subset of  $K$ .

471     The only difference between the `Point` and `Line` objects is in the `view` and `assemble`  
 472     function because line has different continuity from scatter and is represented by a different  
 473     type of 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'])
8
9      def draw(self, renderer, *args, **kwargs):
10         # query data source for edge table
11         view = self.data.view()
12         visual = {p: encoder(view.get(f, None)) for
13                   p, (f, encoder) in self.transforms.items()}
14         self.assemble(visual)
15         super().draw(renderer, *args, **kwargs)
```

---

474     In the `Line` artist, `view` returns a table of edges. Each edge consists of (x,y) points sampled  
 475     along the line defined by the edge and information such as the color of the edge. As  
 476     with `Point`, the data is then converted into visual variables. In `assemble`, this visual  
 477     representation is composed into a set of line segments and then the colors of each line  
 478     segment are set. The colors are guaranteed to correspond to the correct segment because of  
 479     the atomicity constraint on `view`.

## 480     4.2 Encoders $\nu$

481     As mentioned above, the encoding dictionary is specified by the visual fiber component, the  
 482     corresponding data fiber component, and the mapping function. The visual parameter serves  
 483     as the dictionary key because the visual representation is constructed from the encoding  
 484     applied to the data  $\mu = \nu \circ \tau$ . For the scatter plot, the mappings for the visual fiber  
 485     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))}
```

---

486 where the position  $(x,y)$   $\nu$  transformers are identity functions. The size  $s$  transformer is not  
 487 acting on a component of  $F$ , instead it is a  $\nu$  that returns a constant value. While size could  
 488 be embedded inside the `assembly` function, it is added to the transformers to illustrate user  
 489 configured visual parameters that could either be constant or mapped to a component in  $F$ .  
 490 The identity and constant  $\nu$  are explicitly implemented here to demonstrate their implicit  
 491 role in the visual pipeline, but they could be optimized away. More complex encoders can  
 492 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]

```

---

493 where `__init__` can validate that the output of the  $\nu$  is a valid element of the  $P$  com-  
 494 ponent the  $\nu$  function is targeting. Creating a callable class also provides a simple way to  
 495 swap out the specific (data, value) mapping without having to reimplement the validation  
 496 or conversion logic.

497 A test for equivariance can be implemented trivially such that it is independent of data  
 498 or encoder.

---

```

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)

```

---

499 In this example, `is_nominal` checks for equivariance of permutation group actions by ap-  
 500 plying the encoder to a set of values, shuffling values, and checking that (value, encoding)  
 501 pairs remain the same. This equivariance test can be implemented as part of the artist or  
 502 encoder, but for minimal overhead, the equivariant it is implemented as part of the library  
 503 tests.

### 504 4.3 Data $E$

505 The data input into the will often be a wrapper class around an existing data structure,  
 506 but must meet the following criteria:

- 507 1. specify the fiber components  $F$  and connectivity  $K$

508     2. have a that returns an atomic object that encapsulates  $\tau$   
 509     3. the view object must have that returns a fiber component  
 510 To support specifying the fiber bundle, we define an optional `FiberBundle` class

---

```

1  class FiberBundle:
2      def __init__(self, base, fiber):
3          """
4              base: {'tables': ['vertex', 'edge', 'face']}
5              fiber: {'component name': {'type':, 'monoid':, 'range':}}
6          """
7          self.base = base
8          self.fiber = fiber

```

---

511 that asks the user to specify how  $K$  is triangulated and the attributes of  $F$ . The `assembly`  
 512 functions expect tables that match the continuity of the graphic; scatter expects a vertex  
 513 table because it is discontinuous, line expects an edge table because it is 1D continuous.  
 514 The fiber informs appropriate choice of  $\nu$  therefore it is a dictionary of attributes of the  
 515 fiber components. I've basically stripped this out of the artists above so should I just ditch  
 516 this section?

517 To generate the scatter plot in figure 22, we fully specify a dataset with random keys  
 518 and values in a section chosen at random from the corresponding fiber component. The  
 519 fiberbundle FB is a class level attribute since all instances of `VertexSimplex` come from  
 520 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
16     def view(self):
17         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

```

---

521 The view method returns a dictionary where the key is a fiber component name and the  
 522 value is a list of values in the fiber component. The table is built one call to `tau` at a time,  
 523 guaranteeing that all the fiber component values are over the same  $k$ . Table has a `get`  
 524 method as it is a method on Python dictionaries. In contrast, the line in `EdgeSimplex` is  
 525 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, simplex):
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)

```

---

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

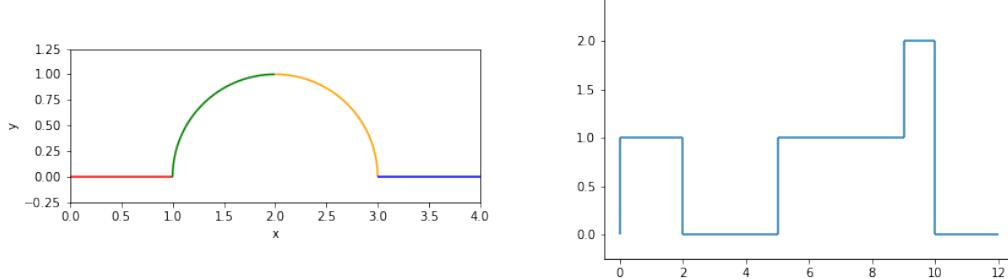


Figure 23: Continuous and discontinuous lines as defined by different data models, but generated with the same  $A'=\text{artist}$

532 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, simplex='edge'):
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, simplex='edge'):
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

```

---

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

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

---

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

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

---

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

#### 541 4.4 Case Study: Penguins

542 For this case study, we use the Palmer Penguins dataset[32, 40] since it is multivariate and  
 543 has a varying number of penguins. We use a version of the data packaged as a pandas  
 544 dataframe[59, 67] since that is a very commonly used Python labled data structure. The  
 545 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):
6         return self._view

```

---

546 The pandas indexer is a key valued set of discrete vertices, so there is no need to repackage  
 547 for triangulation. As with the previous examples, there is no need to implement an explicit  
 548 get method since the `dataframe` object has a get method.

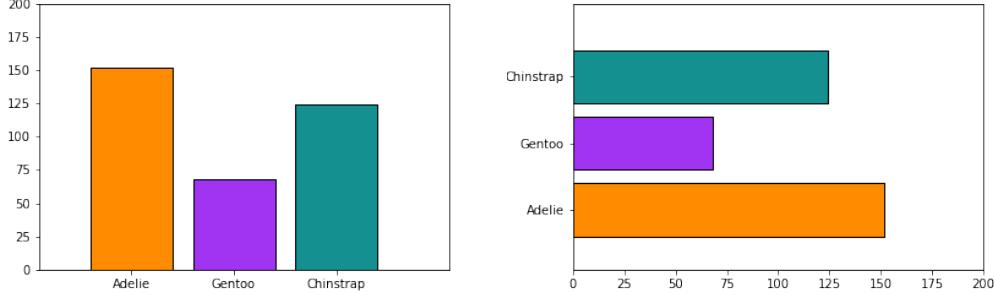


Figure 24: Frequency of Penguin types visualized as discrete bars.

549     The bar charts in figure 24 are generated with a `Bar` artist. They have the same required  
 550      $P$  components of (position, length). In `Bar` an additional parameter is set, `orientation`  
 551     which only applies holistically to the graphic and never to individual data parameters.  
 552     Explicitly differentiate between parameters in  $V$  and ones that are only in  $\hat{Q}$  is another way  
 553     this model allows for cleaner separation of roles in the code.

---

```

1  class Bar(mcollections.Collection):
2      def __init__(self, data, transforms, *args, **kwargs):
3          # parameter of the graphic
4          self.orientation = kwargs.pop('orientation', 'v')
5
6          super().__init__(*args, **kwargs)
7          self.data = data
8          self.transforms = transforms
9
10         @staticmethod
11     def _make_bars(orientation, position, width, floor, length):
12         if orientation in {'vertical', 'v'}:
13             xval, xoff, yval, yoff = position, width, floor, length
14         elif orientation in {'horizontal', 'h'}:
15             xval, xoff, yval, yoff = floor, length, position, width
16         return [[(x, y), (x, y+yo), (x+xo, y+yo), (x+xo, y), (x, y)]
17                 for (x, xo, y, yo) in zip(xval, xoff, yval, yoff)]
18
19
20     def assemble(self, visual):
21         #set some defaults
22         visual['width'] = visual.get('width', itertools.repeat(0.8))
23         visual['floor'] = visual.get('floor', itertools.repeat(0))

```

```

24     visual['facecolors'] = visual.get('facecolors', 'C0')
25     #build bar glyphs based on graphic parameter
26     verts = self._make_bars(self.orientation, visual['position'],
27         visual['width'], visual['floor'], visual['length'])
28     self._paths = [mpath.Path(xy, closed=True) for xy in verts]
29     self.set_edgecolors('k')
30     self.set_facecolors(visual['facecolors'])

31
32     def draw(self, renderer, *args, **kwargs):
33         view = self.data.view()
34         visual = utils.convert_transforms(view, self.transforms)
35         self.assemble(visual)
36         super().draw(renderer, *args, **kwargs)
37         return

```

---

554 The `draw` method identical to the ones above, but here the visual transformations are  
 555 factored out into a separate function. The `assemble` function sets some defaults, constructs  
 556 bars, and sets their edge color to black. The `_make_bars` function is somewhat factored out  
 557 because this is an operation that may be used by other bar making functions that may not  
 558 be able to make use of bars assemble or draw.

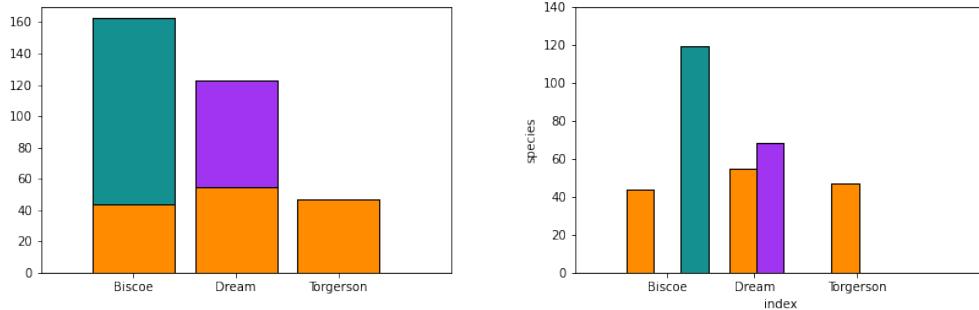


Figure 25: Penguin count disaggregated by island and species

559 For example, the `MultiBar` artist that makes figure 25 reuses `_make_bars` but does  
 560 not reuse the `assemble` function because the composition of elements forces fundamental  
 561 differences in glyph construction. As demonstrated in the `init`, the composite bar chart  
 562 has orientation and whether it is stacked or not. While the stacked bar chart and the grouped  
 563 bar chart could be separate artists, as demonstrated they share so much overlapping code  
 564 that it is far less redundant to implement them together. looking at the mess that is this  
 565 code, I'm a) not convinced these should be combined b) no longer convinced this provides  
 566 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 width 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     #flattened 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

```

---

567 In the `__draw__`, a utility function is used for conversions, but the length transforms  
 568 are held until after assembly because the length is computed by adding the current length  
 569 to the previous and many transforms are not distributable such that  $\nu(x_0 + x_1 + x_2) =$   
 570  $\nu(x_0) + \nu(x_1) + \nu(x_2)$ . Inside `assemble`, the glyphs are either shifted vertically (`stacked`)  
 571 or horizontally (`grouped`) such that the positions are recorded and added to with the next  
 572 group. This function allows multiple columns to be mapped to a visual parameter, but it  
 573 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'})])

```

---

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

---

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

---

579 where `trans` is the same dictionary for both stacked and grouped version, as is the  
580 `DataFrameSection` object `table`. The only difference between the two versions is the  
581 `stacked` flag, and the only difference between figures 24 is the `orientation` argument. By  
582 decomposing the architecture into data, visual encoding, and assembly steps, we are able  
583 to build components that are more flexible and also more self contained than the existing  
584 code base.

585 This API may want to be redesigned such that there's a way to clearly couple the columns  
586 when doing multindex broadcasting

## 587 5 Discussion

588 This work contributes a mathematical description of the mapping  $A$  from data to visual  
589 representation. Combining Butler's proposal of a fiber bundle model of visualization data  
590 with Spivak's formalism of schema lets this mode; support a variety of datasets, includ-  
591 ing discrete relational tables., multivariate high resolution spatio temporal datasets, and  
592 complex networks. Decomposing the artist into encoding  $\nu$ , assembly  $Q$ , and reindexing  $\xi$   
593 provides the specifications for producing visualization where the structure and properties  
594 match those of the input data. These specifications are that the graphic must have continu-  
595 ity equivalent to the data, and that the visual characteristics of the graphics are equivariant  
596 to their corresponding components under monoid actions. This model defines these con-  
597 straints on the transformation function such that they are not specific to any one type of  
598 encoding or visual characteristic. Encoding the graphic space as a fiber bundle provides a  
599 structure rich abstraction of the target graphical design in the target display space.

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

### 611 5.1 Limitations

612 So far this model has only been worked out for a single data set tied to a primitive mark,  
613 but it should be extensible to compositing datasets and complex glyphs. The examples and  
614 prototype have so far only been implemented for the static 2D case, but nothing in the math  
615 limits to 2D and expansion to the animated case should be possible because the model is  
616 formalized in terms of the sheaf. While this model supports equivariance of figurative glyphs

generated form parameters of the data[7, 19], it currently does not have a way to evaluate the semantic accuracy of the figurative representation. Effectiveness is out of scope for this model because it is not part of the structure being preserved, but potentially a developer building a domain specific library with this model could implement effectiveness criteria in the artists. Also, even though the model is designed to be backend independent, it has only really been tested against the AGG backend. It is especially unknown how this framework interfaces with high performance rendering libraries such as openGL[22]. Because this model has been limited to the graphic design space, it does not address the critical task of laying out the graphics in the image

This model and the associated prototype is deeply tied to Matplotlib’s existing architecture. While the model is expected to generalize to other libraries, such as those built on Mackinlay’s APT framework, this has not been worked through. In particular, Mackinlay’s formulation of graphics as a language with semantic and syntax lends itself a declarative interface[50], which Heer and Bostock use to develop a domain specific visualization language that they argue makes it simpler for designers to construct graphics without sacrificing expressivity [38]. Similarly, the model presented in this work formulates visualization as equivariant maps from data space to visual space, and is designed such that developers can build software libraries with data and graphic topologies tuned to specific domains.

## 5.2 Future Work

While the model and prototype demonstrate that generation of simple marks from the data, there is a lot of work left to develop a model that underpins a minimally viable library. Foremost is implementing a data object that encodes data with a 2D continuous topology and an artist that can consume data with a 2D topology to visualize the image[33, 34, 83] and also encoding a separate heatmap[49, 93] artist that consumes 1D discrete data. A second important proof of concept artist is a boxplot[91] because it is a graphic that assumes computation on the data side and the glyph is built from semantically defined components and a list of outliers.

The model supports simple composition of glyphs by overlaying glyphs at the same position, but more work is needed to define an operator where the fiber bundles have shared  $S_2 \hookrightarrow S_1$  such that fibers could be pulled back over the subset. This complex operator is necessary for building semantically meaningful components such as interactive visualizations that update on shared  $S$ , such as brush-linked views[8, 15] and verifying constraints in multiple view systems [62]. The models simple addition supports axes as standalone artists with overlapping visual position encoding, but the complex operator would allow for keeping labels consistent with text when for example the horizontal and vertical positions of the data are changed. In summary, the proposed scope of work for the dissertation is

- expansion of the mathematical framework to include complex addition
- mathematical formulation of a graphic with axes labeling
- implementation of data with 2D continuous topology and a compatible artist
- provisional mathematics and implementation of user level composite artists
- proof of concept domain specific user facing library

Additionally, implementing the complex addition operator would allow for developing a mathematical formalism and prototype of how interactivity would work in this model. Other

660 potential tasks for future work is implementing a data object for a non-trivial fiber bundle  
661 and exploiting the models section level formalism to build distributed data source mod-  
662 els and concurrent artists. This could be pushed further to integrate with topological[39]  
663 and functional [66] data analysis methods. While this paper formulates visualization in  
664 terms of monoidal action homomorphisms between fiberbundles, the model lends itself to a  
665 categorical formulation[29, 54] that could be further explored.

## 666 6 Conclusion

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

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

## 676 References

- 677 [1] *[A Series of Statistical Charts Illustrating the Condition of the Descendants of Former*  
678 *African Slaves Now in Residence in the United States of America] Negro Business Men*  
679 *in the United States.* eng. https://www.loc.gov/item/2014645363/. Image.
- 680 [2] *[A Series of Statistical Charts Illustrating the Condition of the Descendants of Former*  
681 *African Slaves Now in Residence in the United States of America] Negro Population*  
682 *of the United States Compared with the Total Population of Other Countries /.* eng.  
683 https://www.loc.gov/item/2013650368/. Image.
- 684 [3] *Action in nLab.* https://ncatlab.org/nlab/show/action#actions\_of\_a\_monoid.
- 685 [4] James Ahrens, Berk Geveci, and Charles Law. “Paraview: An End-User Tool for Large  
686 Data Visualization”. In: *The visualization handbook* 717.8 (2005).
- 687 [5] *Anti-Grain Geometry -.* http://agg.sourceforge.net/antigrain.com/index.html.
- 688 [6] Professor Denis Auroux. “Math 131: Introduction to Topology”. en. In: (), p. 113.
- 689 [7] F. Beck. “Software Feathers Figurative Visualization of Software Metrics”. In: *2014*  
690 *International Conference on Information Visualization Theory and Applications*  
691 *(IVAPP)*. Jan. 2014, pp. 5–16.
- 692 [8] Richard A. Becker and William S. Cleveland. “Brushing Scatterplots”. In: *Technomet-  
693 rics* 29.2 (May 1987), pp. 127–142. ISSN: 0040-1706. DOI: 10.1080/00401706.1987.  
694 10488204.
- 695 [9] Jacques Bertin. *Semiology of Graphics : Diagrams, Networks, Maps.* English. Red-  
696 lands, Calif.: ESRI Press, 2011. ISBN: 978-1-58948-261-6 1-58948-261-1.
- 697 [10] Enrico Bertini, Andrada Tat, and Daniel Keim. “Quality Metrics in High-  
698 Dimensional Data Visualization: An Overview and Systematization”. In: *IEEE*  
699 *Transactions on Visualization and Computer Graphics* 17.12 (2011), pp. 2203–2212.

- 700 [11] Tim Bienz, Richard Cohn, and Calif.) Adobe Systems (Mountain View. *Portable Doc-*  
 701 *ument Format Reference Manual*. Citeseer, 1993.
- 702 [12] M. Bostock and J. Heer. “Protovis: A Graphical Toolkit for Visualization”. In: *IEEE*  
 703 *Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1121–  
 704 1128. ISSN: 1941-0506. DOI: 10.1109/TVCG.2009.174.
- 705 [13] M. Bostock, V. Ogievetsky, and J. Heer. “D<sup>3</sup> Data-Driven Documents”. In: *IEEE*  
 706 *Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pp. 2301–  
 707 2309. ISSN: 1941-0506. DOI: 10.1109/TVCG.2011.185.
- 708 [14] Brian Wylie and Jeffrey Baumes. “A Unified Toolkit for Information and Scientific  
 709 Visualization”. In: *Proc.SPIE*. Vol. 7243. Jan. 2009. DOI: 10.1117/12.805589.
- 710 [15] Andreas Buja et al. “Interactive Data Visualization Using Focusing and Linking”. In:  
 711 *Proceedings of the 2nd Conference on Visualization ’91*. VIS ’91. Washington, DC,  
 712 USA: IEEE Computer Society Press, 1991, pp. 156–163. ISBN: 0-8186-2245-8.
- 713 [16] D. M. Butler and M. H. Pendley. “A Visualization Model Based on the Mathematics  
 714 of Fiber Bundles”. en. In: *Computers in Physics* 3.5 (1989), p. 45. ISSN: 08941866.  
 715 DOI: 10.1063/1.168345.
- 716 [17] David M. Butler and Steve Bryson. “Vector-Bundle Classes Form Powerful Tool  
 717 for Scientific Visualization”. en. In: *Computers in Physics* 6.6 (1992), p. 576. ISSN:  
 718 08941866. DOI: 10.1063/1.4823118.
- 719 [18] L. Byrne, D. Angus, and J. Wiles. “Acquired Codes of Meaning in Data Visualization  
 720 and Infographics: Beyond Perceptual Primitives”. In: *IEEE Transactions on Visual-*  
 721 *ization and Computer Graphics* 22.1 (Jan. 2016), pp. 509–518. ISSN: 1077-2626. DOI:  
 722 10.1109/TVCG.2015.2467321.
- 723 [19] Lydia Byrne, Daniel Angus, and Janet Wiles. “Figurative Frames: A Critical Vocab-  
 724 uary for Images in Information Visualization”. In: *Information Visualization* 18.1  
 725 (Aug. 2017), pp. 45–67. ISSN: 1473-8716. DOI: 10.1177/1473871617724212.
- 726 [20] *Cairographics.Org*. <https://www.cairographics.org/>.
- 727 [21] Sheelagh Carpendale. *Visual Representation from Semiology of Graphics by J. Bertin*.  
 728 en.
- 729 [22] George S. Carson. “Standards Pipeline: The OpenGL Specification”. In: *SIGGRAPH*  
 730 *Comput. Graph.* 31.2 (May 1997), pp. 17–18. ISSN: 0097-8930. DOI: 10.1145/271283.  
 731 271292.
- 732 [23] John M Chambers et al. *Graphical Methods for Data Analysis*. Vol. 5. Wadsworth  
 733 Belmont, CA, 1983.
- 734 [24] William S. Cleveland. “Research in Statistical Graphics”. In: *Journal of the American*  
 735 *Statistical Association* 82.398 (June 1987), p. 419. ISSN: 01621459. DOI: 10.2307/  
 736 2289443.
- 737 [25] William S. Cleveland and Robert McGill. “Graphical Perception: Theory, Experi-  
 738 *mentation, and Application to the Development of Graphical Methods*”. In: *Journal of the*  
 739 *American Statistical Association* 79.387 (Sept. 1984), pp. 531–554. ISSN: 0162-1459.  
 740 DOI: 10.1080/01621459.1984.10478080.
- 741 [26] “Connected Space”. en. In: *Wikipedia* (Dec. 2020).
- 742 [27] *Data Representation in Mayavi — Mayavi 4.7.2 Documentation*. <https://docs.enthought.com/mayavi/mayavi/da>

- [28] T. W. E. B. Du Bois Center at the University of Massachusetts, W. Battle-Baptiste, and B. Rusert. *W. E. B. Du Bois's Data Portraits: Visualizing Black America*. Princeton Architectural Press, 2018. ISBN: 978-1-61689-706-2.
- [29] Brendan Fong and David I. Spivak. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. en. First. Cambridge University Press, July 2019. ISBN: 978-1-108-66880-4 978-1-108-48229-5 978-1-108-71182-1. DOI: 10.1017/9781108668804.
- [30] Michael Friendly. "A Brief History of Data Visualization". en. In: *Handbook of Data Visualization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 15–56. ISBN: 978-3-540-33036-3 978-3-540-33037-0. DOI: 10.1007/978-3-540-33037-0\_2.
- [31] Berk Geveci et al. "VTK". In: *The Architecture of Open Source Applications* 1 (2012), pp. 387–402.
- [32] Kristen B. Gorman, Tony D. Williams, and William R. Fraser. "Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus Pygoscelis)". In: *PLOS ONE* 9.3 (Mar. 2014), e90081. DOI: 10.1371/journal.pone.0090081.
- [33] Robert B Haber and David A McNabb. "Visualization Idioms: A Conceptual Model for Scientific Visualization Systems". In: *Visualization in scientific computing* 74 (1990), p. 93.
- [34] Charles D Hansen and Chris R Johnson. *Visualization Handbook*. Elsevier, 2011.
- [35] Marcus D. Hanwell et al. "The Visualization Toolkit (VTK): Rewriting the Rendering Code for Modern Graphics Cards". en. In: *SoftwareX* 1-2 (Sept. 2015), pp. 9–12. ISSN: 23527110. DOI: 10.1016/j.softx.2015.04.001.
- [36] Charles R Harris et al. "Array Programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362.
- [37] J. Heer and M. Agrawala. "Software Design Patterns for Information Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (2006), pp. 853–860. DOI: 10.1109/TVCG.2006.178.
- [38] Jeffrey Heer and Michael Bostock. "Declarative Language Design for Interactive Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 16.6 (Nov. 2010), pp. 1149–1156. ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.144.
- [39] C. Heine et al. "A Survey of Topology-Based Methods in Visualization". In: *Computer Graphics Forum* 35.3 (June 2016), pp. 643–667. ISSN: 0167-7055. DOI: 10.1111/cgf.12933.
- [40] Allison Marie Horst, Alison Presmanes Hill, and Kristen B Gorman. *Palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. Manual. 2020. DOI: 10.5281/zenodo.3960218.
- [41] Stephan Hoyer and Joe Hamman. "Xarray: ND Labeled Arrays and Datasets in Python". In: *Journal of Open Research Software* 5.1 (2017).
- [42] J. D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science Engineering* 9.3 (May 2007), pp. 90–95. ISSN: 1558-366X. DOI: 10.1109/MCSE.2007.55.
- [43] John Hunter and Michael Droettboom. *The Architecture of Open Source Applications (Volume 2): Matplotlib*. <https://www.aosabook.org/en/matplotlib.html>.

- 787 [44] “Jet Bundle”. en. In: *Wikipedia* (Dec. 2020).
- 788 [45] Gordon Kindlmann and Carlos Scheidegger. “An Algebraic Process for Visualization  
789 Design”. In: *IEEE transactions on visualization and computer graphics* 20.12 (2014),  
790 pp. 2181–2190.
- 791 [46] John Krygier and Denis Wood. *Making Maps: A Visual Guide to Map Design for GIS*.  
792 English. 1 edition. New York: The Guilford Press, Aug. 2005. ISBN: 978-1-59385-200-9.
- 793 [47] W A Lea. “A Formalization of Measurement Scale Forms”. en. In: (), p. 44.
- 794 [48] *Locally Trivial Fibre Bundle - Encyclopedia of Mathematics*. [https://encyclopediaofmath.org/wiki/Locally\\_trivial](https://encyclopediaofmath.org/wiki/Locally_trivial)
- 795 [49] Toussaint Loua. *Atlas Statistique de La Population de Paris*. J. Dejey & cie, 1873.
- 796 [50] Kenneth C. Louden. *Programming Languages : Principles and Practice*. English. Pa-  
797 cific Grove, Calif: Brooks/Cole, 2010. ISBN: 978-0-534-95341-6 0-534-95341-7.
- 798 [51] Jock Mackinlay. “Automating the Design of Graphical Presentations of Relational  
799 Information”. In: *ACM Transactions on Graphics* 5.2 (Apr. 1986), pp. 110–141. ISSN:  
800 0730-0301. DOI: 10.1145/22949.22950.
- 801 [52] JOCK D. MACKINLAY. “AUTOMATIC DESIGN OF GRAPHICAL PRESENTA-  
802 TIONS (DATABASE, USER INTERFACE, ARTIFICIAL INTELLIGENCE, INFOR-  
803 MATION TECHNOLOGY)”. English. PhD Thesis. 1987.
- 804 [53] Connie Malamed. *Information Display Tips*. [https://understandinggraphics.com/visualizations/information-  
805 display-tips/](https://understandinggraphics.com/visualizations/information-display-tips/). Blog. Jan. 2010.
- 806 [54] Bartosz Milewski. “Category Theory for Programmers”. en. In: (), p. 498.
- 807 [55] “Monoid”. en. In: *Wikipedia* (Jan. 2021).
- 808 [56] Tamara Munzner. “Ch 2: Data Abstraction”. In: *CPSC547: Information Visualization,  
809 Fall 2015-2016* () .
- 810 [57] Tamara Munzner. *Visualization Analysis and Design*. AK Peters Visualization Series.  
811 CRC press, Oct. 2014. ISBN: 978-1-4665-0891-0.
- 812 [58] Jana Musilová and Stanislav Hronek. “The Calculus of Variations on Jet Bundles as a  
813 Universal Approach for a Variational Formulation of Fundamental Physical Theories”.  
814 In: *Communications in Mathematics* 24.2 (Dec. 2016), pp. 173–193. ISSN: 2336-1298.  
815 DOI: 10.1515/cm-2016-0012.
- 816 [59] Muhammad Chenariyan Nakhaee. *Mcnakhaee/Palmerpenguins*. Jan. 2021.
- 817 [60] Donald A. Norman. *Things That Make Us Smart: Defending Human Attributes in the  
818 Age of the Machine*. USA: Addison-Wesley Longman Publishing Co., Inc., 1993. ISBN:  
819 0-201-62695-0.
- 820 [61] Z. Pousman, J. Stasko, and M. Mateas. “Casual Information Visualization: Depictions  
821 of Data in Everyday Life”. In: *IEEE Transactions on Visualization and Computer  
822 Graphics* 13.6 (Nov. 2007), pp. 1145–1152. ISSN: 1941-0506. DOI: 10.1109/TVC  
823 .2007.70541.
- 824 [62] Z. Qu and J. Hullman. “Keeping Multiple Views Consistent: Constraints, Validations,  
825 and Exceptions in Visualization Authoring”. In: *IEEE Transactions on Visualization  
826 and Computer Graphics* 24.1 (Jan. 2018), pp. 468–477. ISSN: 1941-0506. DOI: 10.1109/  
827 TVCG.2017.2744198.

- 828 [63] A. Quint. “Scalable Vector Graphics”. In: *IEEE MultiMedia* 10.3 (July 2003), pp. 99–  
 829 102. ISSN: 1941-0166. DOI: 10.1109/MMUL.2003.1218261.
- 830 [64] “Quotient Space (Topology)”. en. In: *Wikipedia* (Nov. 2020).
- 831 [65] P. Ramachandran and G. Varoquaux. “Mayavi: 3D Visualization of Scientific Data”.  
 832 In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 40–51. ISSN: 1558-366X.  
 833 DOI: 10.1109/MCSE.2011.35.
- 834 [66] James O Ramsay. *Functional Data Analysis*. Wiley Online Library, 2006.
- 835 [67] Jeff Reback et al. *Pandas-Dev/Pandas: Pandas 1.0.3*. Zenodo. Mar. 2020. DOI: 10.  
 836 5281/zenodo.3715232.
- 837 [68] “Retraction (Topology)”. en. In: *Wikipedia* (July 2020).
- 838 [69] Matthew Rocklin. “Dask: Parallel Computation with Blocked Algorithms and Task  
 839 Scheduling”. In: *Proceedings of the 14th Python in Science Conference*. Vol. 126. Cite-  
 840 seer, 2015.
- 841 [70] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. “Declarative Inter-  
 842 action Design for Data Visualization”. en. In: *Proceedings of the 27th Annual ACM*  
 843 *Symposium on User Interface Software and Technology*. Honolulu Hawaii USA: ACM,  
 844 Oct. 2014, pp. 669–678. ISBN: 978-1-4503-3069-5. DOI: 10.1145/2642918.2647360.
- 845 [71] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. “NIH Image to Im-  
 846 ageJ: 25 Years of Image Analysis”. In: *Nature Methods* 9.7 (July 2012), pp. 671–675.  
 847 ISSN: 1548-7105. DOI: 10.1038/nmeth.2089.
- 848 [72] “Semigroup Action”. en. In: *Wikipedia* (Jan. 2021).
- 849 [73] E.H. Spanier. *Algebraic Topology*. McGraw-Hill Series in Higher Mathematics.  
 850 Springer, 1989. ISBN: 978-0-387-94426-5.
- 851 [74] David I Spivak. *Databases Are Categories*. en. Slides. June 2010.
- 852 [75] David I Spivak. “SIMPLICIAL DATABASES”. en. In: (), p. 35.
- 853 [76] “Stalk (Sheaf)”. en. In: *Wikipedia* (Oct. 2019).
- 854 [77] S. S. Stevens. “On the Theory of Scales of Measurement”. In: *Science* 103.2684 (1946),  
 855 pp. 677–680. ISSN: 00368075, 10959203.
- 856 [78] Michele Stieven. *A Monad Is Just a Monoid...* en. <https://medium.com/@michelestieven/a-monad-is-just-a-monoid-a02bd2524f66>. Apr. 2020.
- 858 [79] Software Studies. *Culturevis/Imageplot*. Jan. 2021.
- 859 [80] T. Sugibuchi, N. Spyros, and E. Siminenko. “A Framework to Analyze Informa-  
 860 tion Visualization Based on the Functional Data Model”. In: *2009 13th International*  
 861 *Conference Information Visualisation*. 2009, pp. 18–24. DOI: 10.1109/IV.2009.56.
- 862 [81] *[The Georgia Negro] City and Rural Population. 1890*. eng. <https://www.loc.gov/item/2013650430/>.  
 863 Image. 1900.
- 864 [82] *[The Georgia Negro] Negro Property in Two Cities of Georgia*. eng. <https://www.loc.gov/item/2013650443/>.  
 865 Image. 1900.
- 866 [83] M. Tory and T. Moller. “Rethinking Visualization: A High-Level Taxonomy”. In:  
 867 *IEEE Symposium on Information Visualization*. Oct. 2004, pp. 151–158. DOI: 10.  
 868 1109/INFVIS.2004.59.

- 869 [84] Edward R. Tufte. *The Visual Display of Quantitative Information*. English. Cheshire,  
 870 Conn.: Graphics Press, 2001. ISBN: 0-9613921-4-2 978-0-9613921-4-7 978-1-930824-13-3  
 871 1-930824-13-0.
- 872 [85] John W. Tukey. “We Need Both Exploratory and Confirmatory”. In: *The American  
 873 Statistician* 34.1 (Feb. 1980), pp. 23–25. ISSN: 0003-1305. DOI: 10.1080/00031305.  
 874 1980.10482706.
- 875 [86] Jacob VanderPlas et al. “Altair: Interactive Statistical Visualizations for Python”. en.  
 876 In: *Journal of Open Source Software* 3.32 (Dec. 2018), p. 1057. ISSN: 2475-9066. DOI:  
 877 10.21105/joss.01057.
- 878 [87] P. Vickers, J. Faith, and N. Rossiter. “Understanding Visualization: A Formal Ap-  
 879 proach Using Category Theory and Semiotics”. In: *IEEE Transactions on Visualiza-  
 880 tion and Computer Graphics* 19.6 (June 2013), pp. 1048–1061. ISSN: 1941-0506. DOI:  
 881 10.1109/TVCG.2012.294.
- 882 [88] C. Ware. *Information Visualization: Perception for Design*. Interactive Technologies.  
 883 Elsevier Science, 2019. ISBN: 978-0-12-812876-3.
- 884 [89] Eric W. Weisstein. *Similarity Transformation*. en. <https://mathworld.wolfram.com/SimilarityTransformation.html>  
 885 Text.
- 886 [90] Hadley Wickham. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New  
 887 York, 2016. ISBN: 978-3-319-24277-4.
- 888 [91] Hadley Wickham and Lisa Stryjewski. “40 Years of Boxplots”. In: *The American  
 889 Statistician* (2011).
- 890 [92] Leland Wilkinson. *The Grammar of Graphics*. en. 2nd ed. Statistics and Computing.  
 891 New York: Springer-Verlag New York, Inc., 2005. ISBN: 978-0-387-24544-7.
- 892 [93] Leland Wilkinson and Michael Friendly. “The History of the Cluster Heat Map”.  
 893 In: *The American Statistician* 63.2 (May 2009), pp. 179–184. ISSN: 0003-1305. DOI:  
 894 10.1198/tas.2009.0033.
- 895 [94] Krist Wongsuphasawat. *Navigating the Wide World of Data Visualization Libraries  
 896 (on the Web)*. 2021.
- 897 [95] *Writing Plugins*. en. [https://imagej.net/Writing\\_plugins](https://imagej.net/Writing_plugins).
- 898 [96] Brent A Yorgey. “Monoids: Theme and Variations (Functional Pearl)”. en. In: (),  
 899 p. 12.
- 900 [97] Caroline Ziemkiewicz and Robert Kosara. “Embedding Information Visualization  
 901 within Visual Representation”. In: *Advances in Information and Intelligent Systems*.  
 902 Ed. by Zbigniew W. Ras and William Ribarsky. Berlin, Heidelberg: Springer Berlin  
 903 Heidelberg, 2009, pp. 307–326. ISBN: 978-3-642-04141-9. DOI: 10.1007/978-3-642-  
 904 04141-9\_15.