

## 1 Introduction

In data visualization it is assumed the graphical representation of data match the properties of the data, and in this work we propose that the mathematical notion of equivariance formalizes this expectation. We meld the infoviz communities interest in heterogenous often discrete datasets with the scientific visualization communities emphasis on continuous and sometimes topologically complex datasets. To demonstrate the practical value of our model, we propose a model driven re-architecture of the artist layer of the Python visualization library Matplotlib. In addition to providing a way to ensure the library preserves structure, we propose a functional approach to improve modularity, maintainability, and point to ways in which the library could better support concurrency and interactivity.

## 11 2 Background

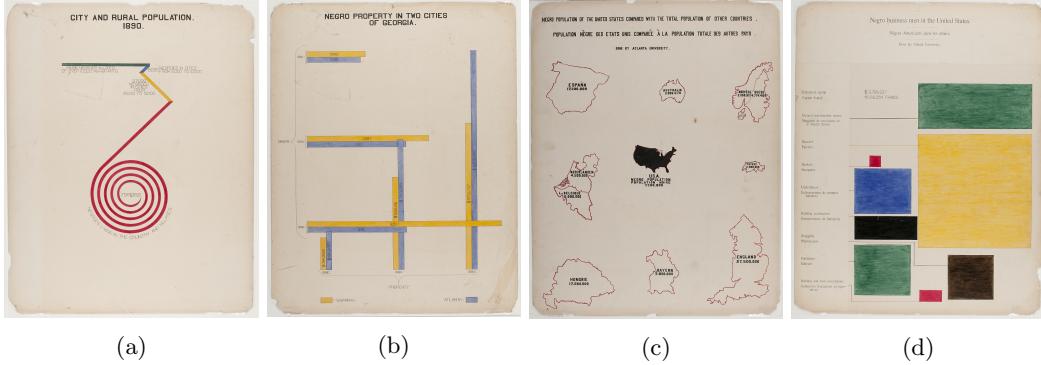


Figure 1: Du Bois’ data portraits[17] of post reconstruction Black American life exemplify that the fundamental characteristics of data visualization is that the visual elements vary in proportion to the source data. In figure 1a, the length of each segment maps to population; in figure 1b, the bar charts are intersected to show the number of property owners and how much they own in a given year; in figure 1c the countries are scaled to population size; and figure 1d 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, 55, 56]

This work aims to develop a model of visualization such that a tool built using this model could support visualizations as varied as those of Du Bois in figure 1; to do so, we first discuss the criteria by which a visualization is evaluated. Byrne et al. propose that visualizations have graphic representations that are mappings from data to visuals and figurative representations that have meaning due to their similarity in shape to external concepts [10]. In figure 1c, Du Bois combines a graphical representation where glyph size varies by population with a figurative representation of those glyphs as the countries the data is from, which means that the semantic and numerical properties of the data are preserved in the graph. Tufte specifies that visual representations must be in proportion to the quantitative data being represented for a chart to be faithful and that there should be no extra information in the graphic or figurative elements of the graph, but otherwise his notion of graphic integrity

is heavily context dependent[58]. As is Norman’s Naturalness Principal, which states that visualizations are more understandable when the properties of the representation match the properties of the information being represented[40]. Bertin takes it as a given that data properties match visual properties, so much so that Munzner argues it is inherently built into his classification system [37] which is displayed in figure 2.

	<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 2: 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 [32] who reproduced it from *Making Maps: A Visual Guide to Map Design for GIS* [27]

As described by Mackinlay, a visualization tool produces a graphical design and an image rendered based on that design. He defines the graphical design as the set encoding relations from data to visual representation[30], and the design rendered in an idealized abstract space is what throughout this paper we will refer to as a graphic. Mackinlay proposes that a visualization tool’s expressiveness is a measure of how much of the structure of the data the tool encodes, while the tools effectiveness describes how much design choices are made in deference to perceptual saliency [12–14, 37]. Mackinlay’s definition of expressiveness is

35 formalized at the visual encoding level, which as shown in figure 2 refers to the components  
 36 of a graphic such as the color or position of a glyph. Bertin first classified these graphic  
 37 components as retinal variables and discussed which types of data they can express [5] and  
 38 how they are composited on point, line, and area graphical marks, as shown in figure 2  
 39 correspond. Marks can be generalized to glyphs, which are graphical objects that convey  
 40 one or more attributes of the data entity mapped to it[60]. Mackinlay's expressiveness  
 41 criteria is well defined for the visual variables, such that he suggests the viability of a strict  
 42 encoding relation that is a homomorphic mapping which preserves some binary operator  
 43 from one domain to another [31]. We expand on this suggestion by proposing that monoid  
 44 action equivariance is a strict condition of building valid encoders. Mackinlay does not  
 45 provide a generalized criteria for plot types, instead embedding the requirements within the  
 46 definition of the charts.

## 47 2.1 Data

48 Tory and Möller propose that assumptions about the structure of data are built into the  
 49 visual algorithms that display that information [57]. Specifically they note that discrete and  
 50 continuous data and their attributes form a discipline independent design space [41].

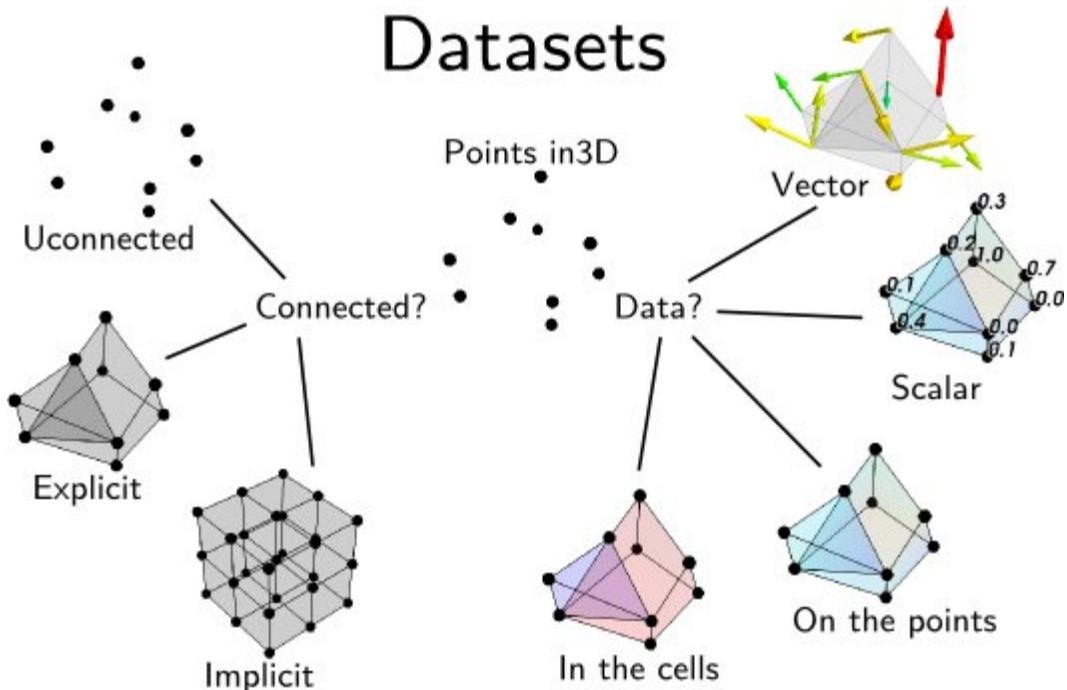


Figure 3: One way to describe data is by the connectivity of the points in the dataset. A database for example is often discrete unconnected points, while an image is an implicitly connected 2D grid. This image is from the Data Representation chapter of the MayaVi 4.7.2 documentation.[16]

51 As shown in figure 3, there are many types of continuity in data. A database typically  
 52 consists of unconnected records, while an image is an implicit 2D grid and a network is some

53 sort of explicitly connected graph. In this work we will refer to the points of the dataset as  
 54 *records* to indicate that a point can be a vector. Each *component* of the record is a single  
 55 object, such as a temperature, a color, or an image. The way in which these records are  
 56 connected is the *connectivity*, *continuity*, or more generally *topology*.

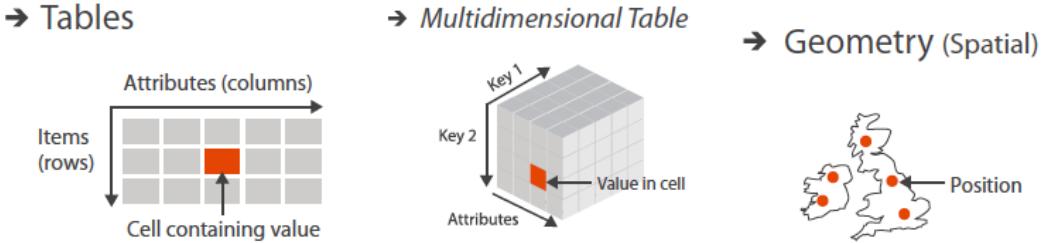


Figure 4: Keys are unique lookup values used to find individual observations in the dataset. Keys are positional references, and can be coordinates on a map or unique values such as a primary key in a database or a (time, latitude, longitude) index in a data cube. Image modified from a diagram from Munzner’s *Visualization Analysis and Design* [37]

57 Often this topology has metadata associated with it, as shown in figure 4. Munzner  
 58 denotes this metadata as *keys* that can be used to locate the record, and further to locate  
 59 the specific *value* [36]. In figure 4 tables have keys identifying the row and column and  
 60 sometimes also the depth, while maps have a location key. We propose that information  
 61 rich keys such as location are additional components of the record, and that instead there  
 62 are coordinate free structural keys that identify the location of the record within a dataset  
 63 of any continuity.

64 In this work, we extend Butler’s topology driven representation of visualization data  
 65 [8, 9]. Butler proposes that fiber bundles are a good model for visualization data because  
 66 it allows for encoding the connectivity separately from the records and supports discrete  
 67 and ND continuous datasets. Since Butler’s model lacks a robust way of describing the  
 68 components of the record, we fold in Spivak’s Simplicial formulation of databases [50, 51] so  
 69 that we can encode a schema like description of the data in the fiber bundle. We then propose  
 70 criteria on expressivity that take into account both the components and the continuity of  
 71 the data.

## 72 2.2 Tools

73 A motivator for this work is that currently Matplotlib carries implicit assumptions about  
 74 data continuity in how each function interfaces with the input data. This work proposes a  
 75 unified internal representation that encodes connectivity in a common interface. Matplotlib  
 76 aims to natively support data of varying connectivities, so tools primarily concerned with  
 77 visualizing relational data of the type found in databases are insufficient models. Many of  
 78 these tools are built on top of Wilkenson’s Grammar of Graphics [63] which itself incor-  
 79 porates Mackinlay’s A Presentation Tool (APT) display algebra; GoG derivative include  
 80 ggplot[62], protovis[6] and D3 [7], vega[46] and altair[59]. While many of these tools sup-  
 81 port images, the first class data container is a table like object of discrete records. Tools  
 82 that primarily support images are also insufficient. For example ImageJ[47] and the Im-  
 83 agePlot[54]macro have some support for visualizing non image components of a complex

84 data set, but mostly in service to the image being visualized. Plugins exist, but must work  
 85 around the everything is an image data model[64]. There are also visualization tools that do  
 86 not carry implicit assumptions about structure, for example vtk[19, 21] and its derivatives  
 87 such as MayaVi[43]. Not totally positive but I think VTK is deeply coupled to the renderer  
 88 and that's why we're not using it as a model, but not positive and this is literally what  
 89 Marc worked on...

### 90 2.3 contribution

91 The contribution of this work is a model we call the topological artist model (TAM) in  
 92 which data and graphics can be viewed as sections of fiber bundles. This model allows for (1)  
 93 decomposing the translation of data fields (variables) into visual channels via an equivariant  
 94 map on the fibers and (2) a topology-preserving map of the base spaces that translates the  
 95 dataset connectivity into graphical elements. Furthermore, this model supports an algebraic  
 96 sum operation such that more complex visualizations can be built from simple ones. To  
 97 demonstrate the practical value of the model, we built a prototype where we represent the  
 98 topological base spaces using triangulation, make use of programming types for the fiber,  
 99 and build on Matplotlib's existing infrastructure for the rendering.

## 100 3 Topological Artist Model

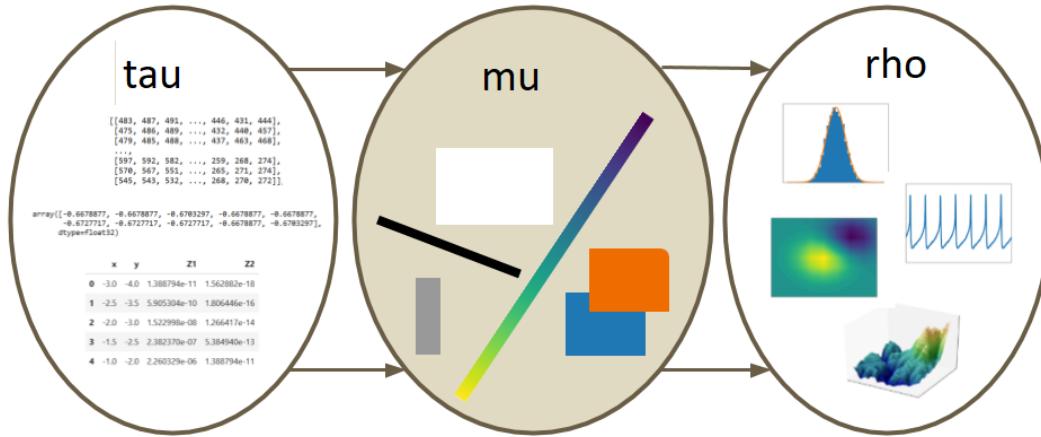


Figure 5: Visualization is equivariant maps between data and visual encoding of the variables and assembly of those encodings into a graphic. not gonna name these bubbles tau, mu, rho, but might keep the same basic structure of different types of data and encodings

101 should this be in 3rd person passive?

102 Visualization is generally thought of as structure preserving maps from data into graphics,  
 103 and in this section we formally define that structure and how it is preserved via equivariant  
 104 maps. We can then specify that a faithful visual mapping is structure preserving, and apply  
 105 these constraints to visualizations we may want to develop or implement. We model the data,  
 106 visual characteristic, and graphic stages of visualization, shown in figure 5, as topological  
 107 structures that encapsulate types of variables and continuity; by doing so we can develop

108 implementations that keep track of both in ways that let us distribute computation while  
 109 still allowing assembly and dynamic update of the graphic.

110 We introduce a mathematical description of the visualization pipeline where artist  $\mathcal{A}$   
 111 functions transform data space  $\mathcal{E}$  to an intermediate representation in a prerendered graphic  
 112 space  $\mathcal{H}$ .

$$\mathcal{A} : \mathcal{E} \rightarrow \mathcal{H} \quad (1)$$

113 We first describe how we model data(3.1), graphics(3.2), and intermediate visual char-  
 114 acteristics (3.3) as fiber bundles. We then discuss the equivariant maps between data and  
 115 visual characteristics (3.3.2) and visual characteristics and graphics (3.3.3) that make up  
 116 the artist.

### 117 3.1 Data Space $E$

118 We build on Butler’s proposal of using fiber bundles as a common data representation  
 119 format for visualization data[8, 9] because fiber bundles are mathematical structures that  
 120 are flexible enough express all the types of data described in section ??.

121 We model data as the fiber bundle  $(E, K, \pi, F)$ , where  $E$   $F$  and  $K$  are topological  
 122 spaces that encode

123  $F$  the properties of the variables in the fiber (3.1.1)

124  $K$  the continuity of the records in the base space (3.1.3)

125  $\tau$  collections of records (3.1.4).

and  $E$  is the total space of data that  $F$  lives in. The bundle is the projection map  $\pi$

$$F \hookrightarrow E \xrightarrow{\pi} K \quad (2)$$

126 that binds the variables  $F$  continuity  $K$ . The fiber bundles mentioned in this work  
 127 are assumed to be trivial[29, 49], unless otherwise specified, because the trivial bundle is  
 128  $E = K \times F$  such that extra structure in the total space  $E$  falls out and discussion can be  
 129 focused on the fiber and base space.

#### 130 3.1.1 Variables: Fiber Space $F$

The fiber is a topological space that is the set of possible values of the data; the values themselves can be any dimension and type and have any continuity. We use Spivak’s description of simplicial database schemas [51] as the basis of our fiber space because he 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  $\mathbb{U}_\sigma$

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

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

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

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:

$$F = \mathbb{U}_{T_1} \times \dots \mathbb{U}_{T_i} \dots \times \mathbb{U}_{T_n} \quad (5)$$

which is equivalent to

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

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

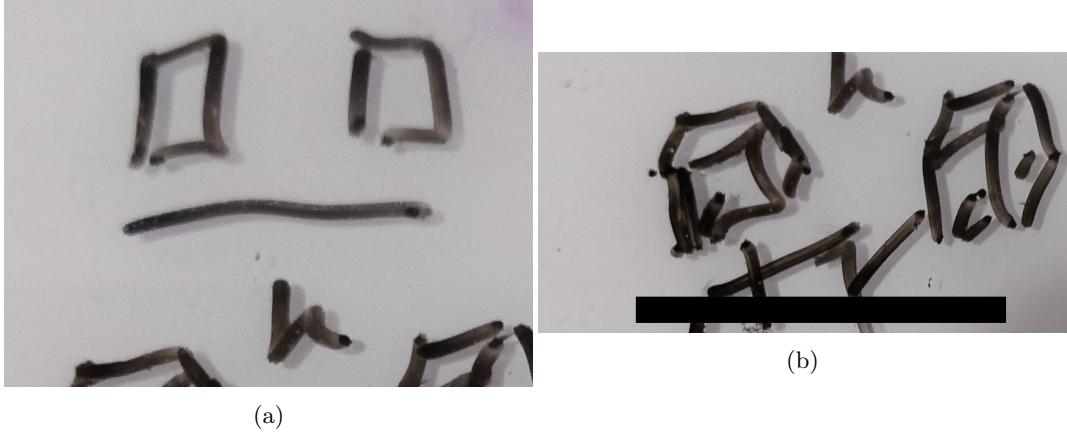


Figure 6: These two datasets have the same base space  $K$  but figure 6a has fiber  $F = \mathbb{R} \times \mathbb{R}$  which is (time, temperature) while figure 6b has fiber  $\mathbb{R}^+ \times \mathbb{R}^2$  which is (time, wind=(speed, direction))

For example, the data in figure 6a is a pair of times and °C temperature measurements taken at those times. Time is a positive number of type `datetime` which can be resolved to positive floats  $\mathbb{U}_{\text{datetime}} = \mathbb{R}^+$ . Temperature values are real 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 ( $c_1 = \text{temperature}$ ,  $T_1 = \text{float}$ ,  $\mathbb{U}_\sigma = \mathbb{R}$ ). In figure 6b, temperature is replaced with wind. This wind variable is of type `wind` and has two components speed and direction  $\{(s, d) \in \mathbb{R}^2 \mid 0 \leq s, 0 \leq d \leq 360\}$ . Therefore, the fiber is

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

132 such that  $F_1$  is specified by ( $c_1 = \text{wind}$ ,  $T_1 = \text{wind}$ ,  $\mathbb{U}_\sigma = \mathbb{R}^2$ )

133 **3.1.2 Measurement Scales: Monoid Actions**

134 After specifying  $F$  we next describe the ways in which we can transform the values by  
135 identifying the monoid actions  $M$  on the  $F$ . We use monoids as the abstraction because  
136 they encode compositiblity, which maps well to the data transformation process in a software  
137 library [65].

A monoid [34]  $M_i$  is a set with an associative binary operator  $* : M_i \times M_i \rightarrow M_i$ . A monoid has an identity element  $e \in M_i$  such that  $e * a = a * e = a$  for all  $a \in M_i$ . A left monoid action [3, 48] of  $M_i$  is a set  $F_i$  with an action  $\bullet : M \times F_i \rightarrow F_i$  with the properties:

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

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

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

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

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

140 Steven's described the measurement scales[28, 53] in terms of the monoid actions on the  
141 measurements: nominal data is permutable, ordinal data is monotonic, interval data is trans-  
142 llatable, and ratio data is scalable [61]. For example, given the fiber  $(c = \text{temperature}, T =$   
143  $\text{float}, \mathbb{U}_\sigma = \mathbb{R}$ ) which is interval data:

- 144 • monoid operator addition  $* = +$
- 145 • monoid operations:  $f : x \mapsto x + 1$ ,  $g : x \mapsto x + 2$
- 146 • monoid action operator composition  $\bullet = \circ$

then the translation monoid actions on temperature satisfy the condition

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

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

148 **3.1.3 Continuity: Base Space  $K$**

149 The advantage of fiber bundles is they provide a way to encode the continuity in a dataset  
150 as the base space  $K$  without making assumptions as to what that continuity is. In turn this  
151 representation of continuity can then be used to keep track of how the data fits together,  
152 for example if a visualization of a very large dataset calls for parallelization.

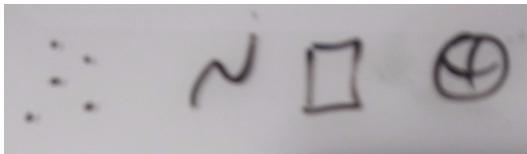


Figure 7: The topological base space  $K$  encodes the connectivity of the data space, for example if the data is independent points or a map or on a sphere

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

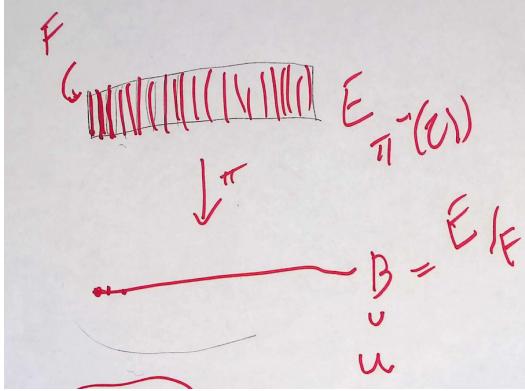


Figure 8: The base space  $E$  is divided into fiber segments  $F$ . The base space  $K$  acts as an index into the records in the fibers. *this figure might be good all the way up top to lay out the components of fb*

155 Formally  $K$  is the quotient space [42] of  $E$  meaning it is the finest space[4] such that  
 156 every  $k \in K$  has a corresponding fiber  $F_k$ [42]. In figure 8,  $E$  is a rectangle divided by  
 157 vertical fibers  $F$ , so the minimal  $K$  for which there is always a mapping  $\pi : E \rightarrow K$  is the  
 158 line.

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

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

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

161 The datasets in figure 9 have the same fiber of (temperature, time). In figure 9a the  
 162 fibers lie over discrete  $K$  such that the records in the datasets in the fiber bundles are  
 163 discrete. The same fiber in figure 9b lies over a continuous interval  $K$  such that the records  
 164 are samples from a continuous function defined on  $K$ .

### 165 3.1.4 Data: Sections $\tau$

While the fiber and base space describe the general structure of all data that lives in the fiber bundle, the sections  $\tau : K \rightarrow E$  define the datasets that live in the fiber. We generalize Spivak's description of the section as a table of records [51] to any sort of structured dataset such that

$$\begin{array}{ccc} F & \hookrightarrow & E \\ & \pi \downarrow \uparrow \tau & \\ & & K \end{array} \quad (12)$$

such that there is always a map  $\pi(\tau(k)) = k$ . There can be many sections  $\tau$ ; the space of global sections is  $\Gamma(E)$ . For a trivial fiber bundle, the section is

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

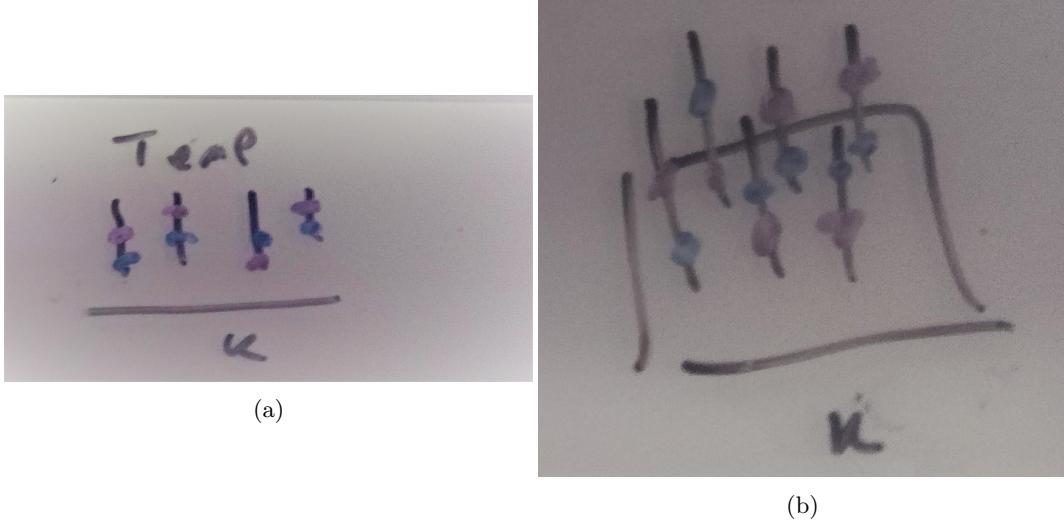


Figure 9: These two datasets have the same (time, temperature) fiber. In figure ?? the total space  $E$  is discrete over points  $k \in K$ , meaning the records in the fiber are also discrete. In figure ??  $E$  lies over the continuous interval  $K$ , meaning the records in the fiber are sampled from a continuous space. *revamp figure: F=Plane, k1 = dots, k2=line*

where  $g : K \rightarrow F$  is the index function into the fiber. Because we can decompose the bundle and the fiber, we can formulate  $\tau$  as

$$\tau = (\tau_0, \dots, \tau_i, \dots, \tau_n) \quad (14)$$

<sup>166</sup> where each section  $\tau_i$  is a variable or set of variables.

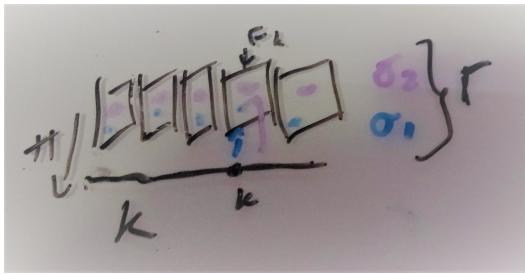


Figure 10: Fiber (time, temperature) with an interval  $K$  basespace. The sections  $\tau_i$  and  $\tau_j$  are constrained such that the time variable must be monotonic, which means each section is a timeseries of temperature values. They are included in the global set of sections  $\tau_1, \tau_2 \in \Gamma(E)$

<sup>167</sup> In the example in figure 10, the fiber is *(time, temperature)* as described in figure 6  
<sup>168</sup> and the base space is the interval  $K$ . The section  $\tau_i$  resolves to a series of monotonically  
<sup>169</sup> increasing in time records of *(time, temperature)* values. Section  $\tau_j$  returns a different  
<sup>170</sup> timeseries of *(time, temperature)* values. Both sections are included in the global set of  
<sup>171</sup> sections  $\tau_1, \tau_2 \in \Gamma(E)$ .

172 **3.1.5 Sheaf and Stalk**

173 Often a graphic may need to be updated with live data or support zooming in on a segment  
 174 of the dataset; to support working with a subset of data, we can use the sheaf  $\mathcal{O}(E)$ . All fiber  
 175 bundles are locally trivial, which means that  $E$  restricted over a small enough neighborhood  
 176  $U \subset K$  is a locally trivial bundle over  $U$ [29]. The sheaf  $\mathcal{O}(E)$  is the localized section of  
 177 fibers  $\iota^*\tau : U \rightarrow \iota^*E$

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

178 pulled back over the neighborhood  $U$  via the inclusion map  $\iota : U \rightarrow K$ . The localized section  
 179 is the germ  $\xi^*\tau$ . The neighborhood of points  $k_i$  surrounding the point  $k$  the sheaf lies over  
 180 is the stalk  $\mathcal{F}_b$  [49, 52]. While  $E$  is only the fiber  $F_k$  over a specific  $k$ , the stalk includes  
 181 nearby records because the sheaf lies over the neighborhood  $U$ . While this can be useful  
 182 for visual transforms, often the extra needed information can be found in the smaller jet  
 183 bundle  $\mathcal{J}$  [26, 38]. For example, line thickness requires the derivative of the given position  
 184 to be rendered, which can be found in  $E' = E + \mathcal{J}(E)$

185 **3.2 Graphic:  $H$**

186 We can separate the structure of the graphic from the properties of the output format by  
 187 modeling the space of graphics as a fiber bundle  $(H, S, \pi, D)$ . As with data, the fiber bundle  
 188 is for a class of graphics with shared base space  $S$ ( 3.2.1) and fiber  $D$ ( 3.2.2) and the sections  
 189  $\rho$ ( 3.2.3) encode a graphic where the visual characteristics are fully specified.

190 **3.2.1 Idealized Display  $D$**

The fiber  $D$  is an idealized infinite resolution version of the target display space, for example  
 a 2D screen or 3D printer. In this work, we assume a 2D opaque image  $F = \mathbb{R}^5$  with elements

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

191 such that a rendered graphic only consists of 2D position and color. To support overplotting  
 192 and transparency, the fiber could be  $F = \mathbb{R}^7$  such that  $(x, y, z, r, g, b, a) \in D$  specifies the  
 193 target display.

194 **3.2.2 Continuity of the Graphic  $S$**

An assumption of graphical representations is that they match the continuity of the data[18,  
 58], but the underlying topology  $S$  of a graphic may need more dimensions than the data  
 topology  $K$  so that the glyph can be defined in  $F$ . Therefore we define the base space  
 mapping from graphic  $S$  to data  $K$

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



Figure 11: The scatter and line graphic base spaces have one more dimension of continuity than  $K$  so that  $S$  can encode physical aspects of the glyph, such as shape (a circle) or thickness. The heatmap has the same continuity in the graphic  $S$  as in the data  $K$ . **add  $\alpha, \beta$  coordinates to figures**

as the deformation retraction [45]  $\xi : S \rightarrow K$  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  $K$ .

In figure 11 each disk  $S_k$  indexes how elements in  $D$  are glued together to generate a single glyph that is the visual representation of a single record in  $F_k$ . For the line, 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.

### 3.2.3 Rendering $\rho$

A section  $\rho : S \rightarrow H$  defines a piece of the graphical representation of the data. Evaluated on a single  $s$   $\rho$  returns a single element in  $H$ . For a 2D screen, the pixel is defined as a region  $p = [y_{top}, y_{bottom}, x_{right}, x_{left}]$  of the rendered graphic. Since the  $x$  and  $y$  in  $p$  are in the same coordinate system as the  $x$  and  $y$  components of  $D$  the inverse map of the bounding box  $S_p = \rho_{xy}^{-1}(p)$  is a region  $S_p \subset S$ . Integrating over this region on  $S$

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

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

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

yields the color of the pixel  $p$ .

As shown in figure 12, the output space queries into the graphic bundle to render the image. We select a pixel  $p$  in the output space, inverse map the region of the pixel into  $S_p \subset S$ , then compute the section  $gsection$  over the region  $S_p$ . The section yields the set of elements in  $D$  that specify the  $(r, g, b)$  values corresponding to the region  $p$ . The color of the pixel is then obtained by taking the integral of  $\rho_{rgb}(S_p)$ .

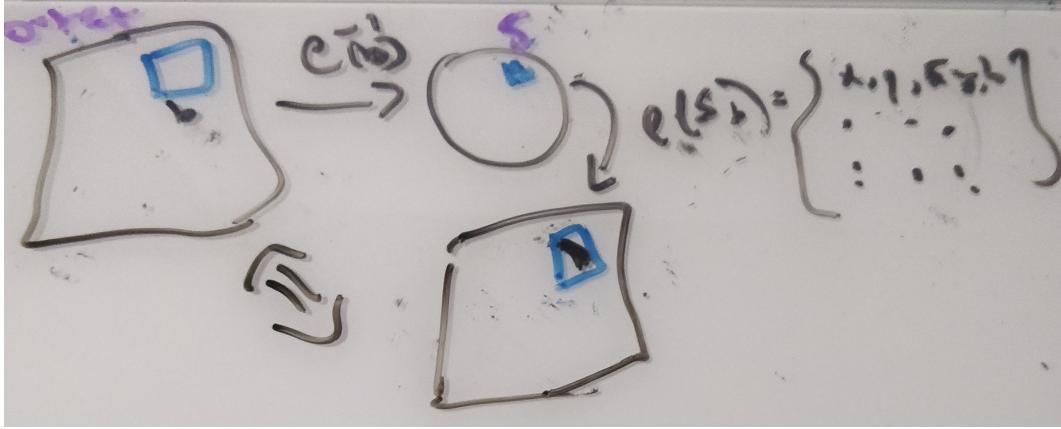


Figure 12: To render a graphic, a pixel  $p$  is selected in the display space, which is defined in the same coordinates as the  $x$  and  $y$  components in  $D$ . The inverse mapping  $\rho_{xy}(p)$  returns a region  $S_p \subset S$ .  $\rho(S_p)$  returns the list of elements  $(x, y, r, g, b) \in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

### 210 3.3 Artist

211 The artist is the function that converts data into graphics; its name is taken from the  
 212 analogous part of Matplotlib[25] that builds visual elements to pass off to the renderer. The  
 213 artist  $A$  is a mapping from  $E$  padded with data from  $\mathcal{J}(E)$  to a graphic that is a section  $\rho$   
 214 in  $\Gamma(H)$

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

215 with an intermediate fiber bundle  $V$  to hold visual representations and stages

- 216 1.  $\xi$  binding the continuity in the graphic to the continuity in the data ( 3.2.2)  
 217 2.  $\nu$  conversion of data into visual characteristics ( 3.3.2)  
 218 3.  $Q$  assembly of visual variables into a glyph ( 3.3.3)

219 of the visual transformation illustrated in figure 5. The functions  $\xi$   $\nu$  and  $Q$  are defined  
 220 such that they can be evaluated on a single section  $\tau$ , which allows the artist to be imple-  
 221 mented such that it does not need all the data. This allows for artists tuned to distributed  
 222 and streaming data.

#### 223 3.3.1 Visual Fiber Bundle $V$

224 The visual fiber bundle  $(V, K, \pi, P)$  has section  $\mu : V \rightarrow K$  that resolves to a visual  
 225 variable [5, 35] in fiber  $P$ . The fiber space  $P$  is defined in terms of the parameters of  
 226 the visualization specification- for example aesthetics in ggplot [62], channels in vega[46] or  
 227 parameters in VTK[21] and Matplotlib.

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

228     Table 1 is a sample of the fiber space for Matplotlib [24]. A section  $\mu$  is a tuple of  
 229     visual values that specifies the visual characteristics of a part of the graphic. For example,  
 230     given a fiber of  $\{xpos, ypos, color\}$  one possible section could be  $\{.5, .5, (255, 20, 147)\}$ . The  
 231      $\text{codomain}(\nu_i)$  determines the monoid actions on  $\mu_i$ . These fiber components are implicit  
 232     in the library, by making them explicit as components of the fiber we can build consistent  
 233     definitions and expectations of how these parameters behave.

### 234     3.3.2 Visual Channels

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  $v\text{fiber}_i$ . A validly constructed  $\nu$  is one where the diagram of the monoid transform  $m$

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

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  $t(x) = x + 2$

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

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

235     On the other hand figure 13 illustrates a valid  $\nu$  mapping from **Strings** to symbols. The  
 236     group action on these sets is permutation, so shuffling the words must have an equivalent

```

[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 13: 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.

237 shuffle of the symbols they are mapped to. To preserve ordinal and partial order monoid  
 238 actions,  $\nu$  must be a monotonic function such that given  $x_1, x_2 \in E_i$ , if  $delement_1 \leq$   
 239  $delement_2$  then  $\nu(x_1) \leq \nu(x_2)$ . For interval scale data,  $\nu$  is equivariant under translation  
 240 monoid actions if  $\nu(x + c) = \nu(x) + \nu(c)$ . For ratio data, there must be equivalent scaling  
 241  $\nu(xc) = \nu(x)\nu(c)$ .

### 242 3.3.3 Assembling Marks

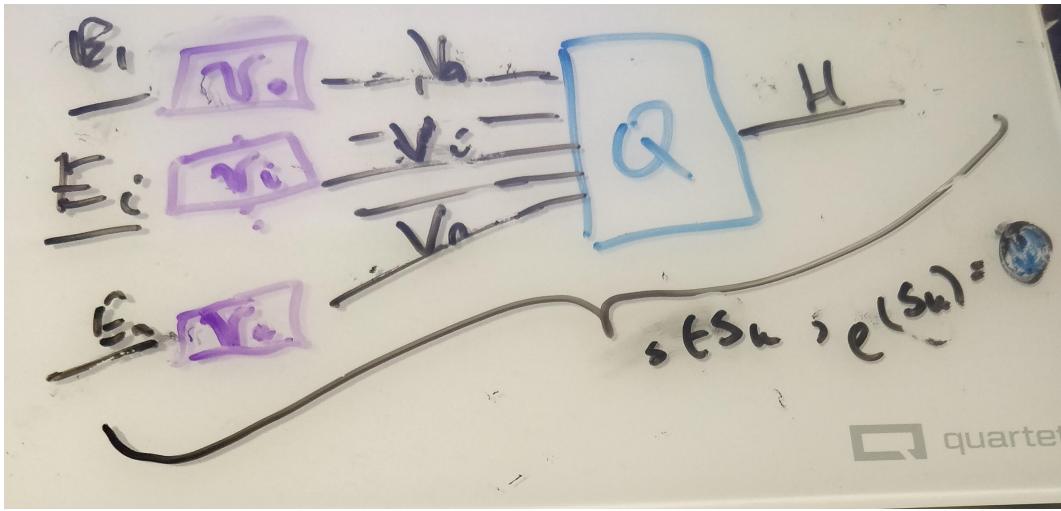


Figure 14:  $\nu$  functions convert data  $\tau_i$  to visual characteristics  $\mu_i$ , then  $Q$  assembles  $\mu_i$  into a graphic  $\rho$  such that there is a map  $\xi$  preserving the continuity of the data.  $\rho$  applied to a region of connected components  $S_j$  generates a graphical mark.

243 As shown in figure 14, the assembly function  $Q$  combines the fiber  $F_i$  wise  $\nu$  transforms  
 244 into a single glyph. Together,  $\nu$  and  $Q$  are a map-reduce operation: map the data into  
 245 their visual encodings, reduce the encodings into a glyph. As with  $\nu$  the constraint on  $Q$  is  
 246 that for every monoid actions on the input  $\mu$  there is a corresponding monoid action on the  
 247 output  $\rho$ .

248 Since we define the equivariant map as  $Q : \mu \mapsto \rho$ , we define an action on the subset  
 249 of graphics  $Q(\Gamma(V)) \in \Gamma(H)$  that  $Q$  can generate. We then define the constraint on  $Q$  such  
 250 that if  $Q$  is applied to  $\mu, \mu'$  that generate the same  $\rho$  then the output of both sections acted  
 251 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

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

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

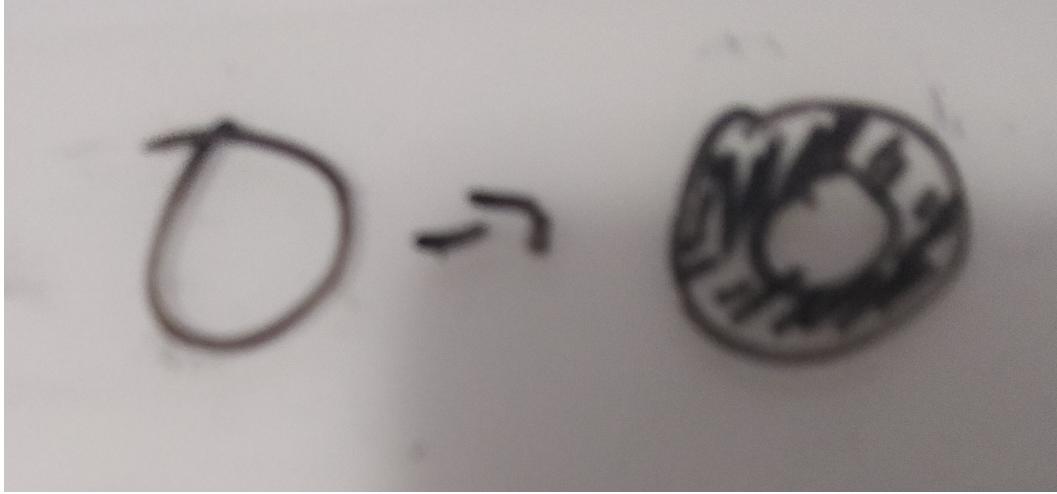


Figure 15: These two glyphs are generated by the same  $Q$  function, but differ in the value of the edge thickness parameter  $\mu_i$ . A valid  $Q$  is one where a shift in  $\mu_i$  is reflected in the glyph generated by  $\rho$ .

255     The glyph in figure 15 has the following characteristics  $P$  specified by  $(xpos, ypos, color, thickness)$   
 256    such that one section is  $\mu = (0, 0, 0, 1)$  and  $Q(\mu) = \rho$  generates a piece of the thin hollow  
 257    circle. The equivariance constraint on  $Q$  is that the action  $m = (e, e, e, x + 2)$ , where  $e$  is  
 258    identity, applied to  $\mu$  such that  $\mu' = (e, e, e, 3)$  has an equivalent action on  $\rho$  that causes  
 259     $Q(\mu')$  to be equivalent to the thicker circle in figure 15.

260     DEGENERATE Q - drawing a blank if this is necessary and if so how  
 261     Check a well defined map  $M \times Y \rightarrow Y$

To output a mark [5, 11],  $Q$  is called with all the regions  $s$  that map back to a set of connected components  $J \subset K$ :

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

262    where the path[15]  $\gamma$  from  $k$  to  $j$  is a continuous function from the interval  $[0, 1]$ . We define  
 263    the mark as the graphic generated by  $Q(S_j)$

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

264    such that for every mark there is at least one corresponding section on  $K$ .

### 265    3.3.4 Sample Qs

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

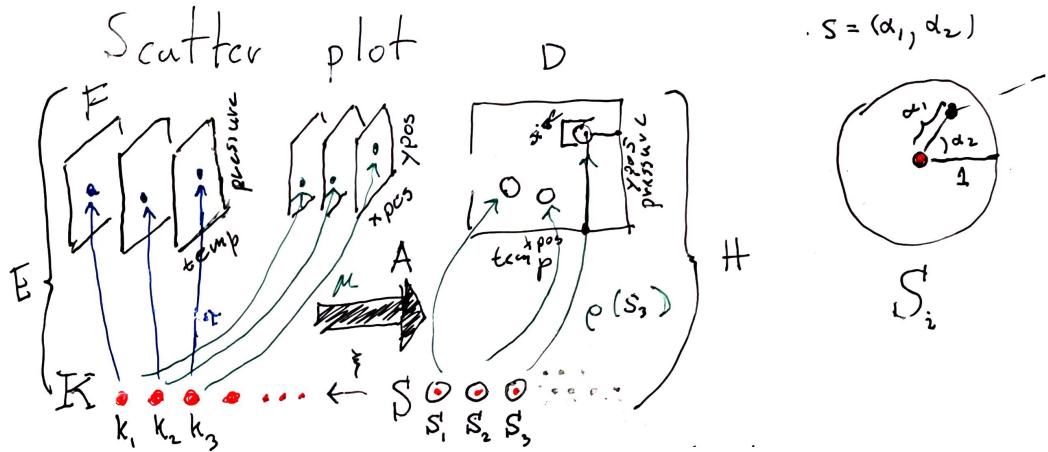


Figure 16: The data is discrete points (temperature, time). Via  $\nu$  these are converted to (xpos, ypos) and pulled over discrete  $S$ . These values are then used to parameterize  $\rho$  which returns a color based on the parameters (xpos,ypos) and position  $\alpha, \beta$  on  $S_k$  that  $\rho$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

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

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

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

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

268

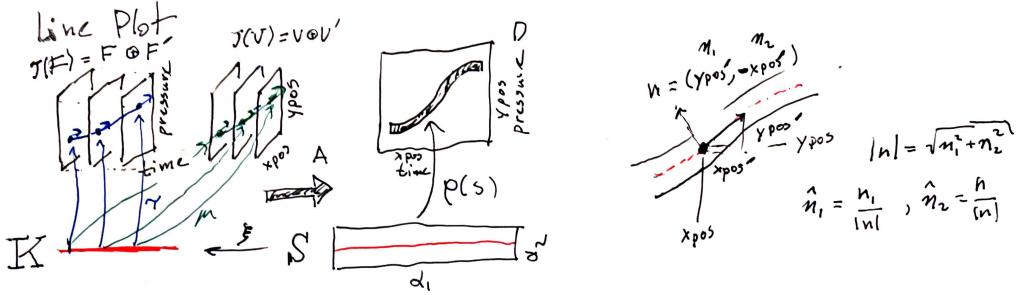


Figure 17: The line fiber (*time*, *temp*) is thickened with the derivative (*time'*, *temperature'*) because that information will be necessary to figure out the tangent to the point to draw a thick line. This is because the line needs to be pushed perpendicular to the tangent of (*xpos*, *ypos*). *this 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 16 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  $\rho$

$$x = xpos(\xi(\alpha)) + \hat{\beta}(n_1)(\xi(\alpha)) \quad (33)$$

$$y = ypos(\xi(\alpha)) + \hat{\beta}(n_2)(\xi(\alpha)) \quad (34)$$

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

**Q: heatmap**

272

Figure 19: components of axes + glyphs

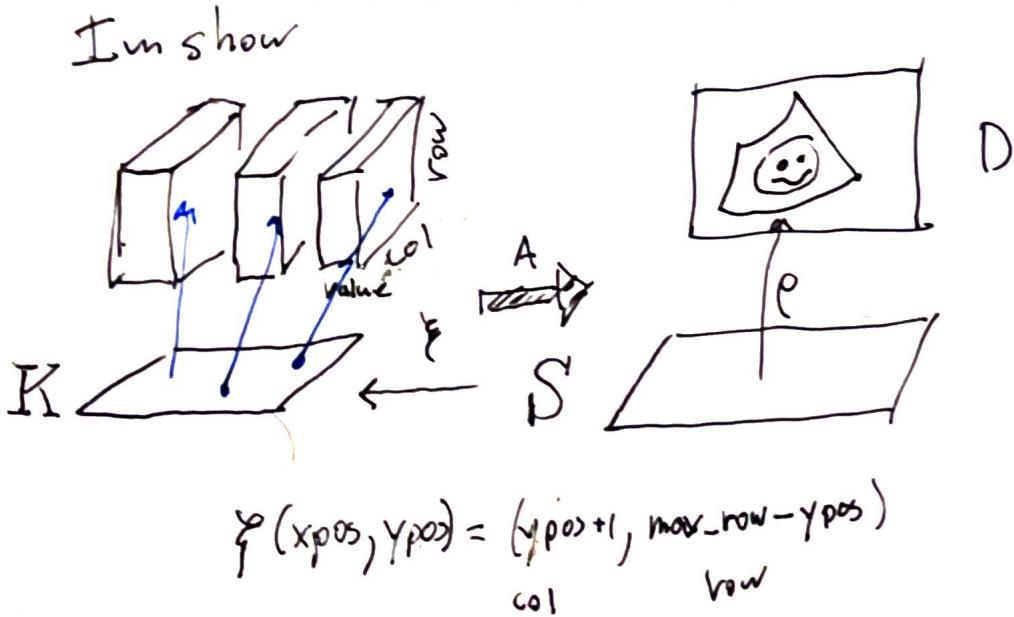


Figure 18: The only visual parameter a heatmap requires is color since  $\xi$  encodes the mapping between position in data and position in graphic.

<sup>273</sup> The heatmap  $Q(\text{color})$  in figure 18 is a direct lookup  $\xi : S \rightarrow K$  such that

$$R = R(\xi(\alpha, \beta)) \tag{35}$$

$$G = G(\xi(\alpha, \beta)) \tag{36}$$

$$B = B(\xi(\alpha, \beta)) \tag{37}$$

<sup>274</sup> where  $\xi$  may do some translating to a convention expected by  $Q$  for example reorienting the  
<sup>275</sup> array such that the first row in the data is at the bottom of the graphic.

### <sup>276</sup> 3.3.5 + union

<sup>277</sup> disjoin union of fibers so we can have an axes/real figure, - diagram operator w/o overplotting,  
<sup>278</sup> library won't auto stack, have to bake that into data, haskell data/depth, hover, mouse  
<sup>279</sup> over curve, picked a point selecting a little k, + operator/disjoint union in different k, same  
<sup>280</sup> D

### <sup>281</sup> 3.3.6 Equivalence class of artists $A'$

<sup>282</sup> As formulated above, every artist function  $A$  has fixed  $\nu$  and  $Q$  which generates a distinct  
<sup>283</sup> graphic  $\rho$ . It is impractical to implement an artist for every single graphic; instead we

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

284 implement the equivalence class of artists  $\{A \in A' : A_1 \equiv A_2\}$ . Equivalent artists have  
 285 the same fiber bundle  $V$  and same assembly function  $Q$  but act on different sections  $\mu$ .  
 286 To further simplify implementation, we identify a minimal  $P$  associated with each  $A'$  that  
 287 defines what visual characteristics of the graphic must originate in the dataneeds citation,  
 288 maybe friendlys history or acquired codes of meaning. For example, a scatter plot of red  
 289 circles is the output of one artist, a scatter plot of green squares the output of another, as are  
 290 the rest of the graphs in figure ???. These two artists are equivalent since their only difference  
 291 is in the literal visual encodings (color, shape). Shape and color could also be defined in  $Q$   
 292 but the position must come from the fiber  $P = (xpos, ypos)$  since fundamentally a scatter  
 293 plot is the plotting of one position against another[18]. We also use this criteria to identify  
 294 derivative types, for example the bubble chart[58] is a type of scatter where by definition  
 295 the glyph size is mapped from the data.

296 **3.4 Making the fiber bundle computable**

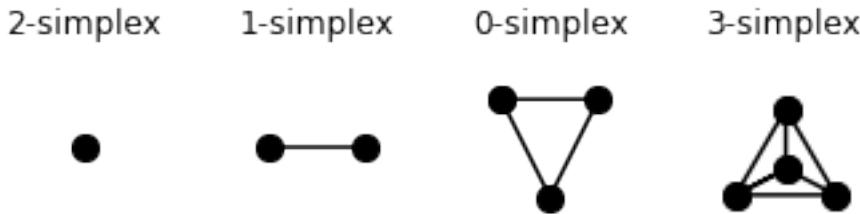


Figure 21: Simplices can encode the connectivity of the data, from fully disconnected (0 simplex) records to all records are connected to at least 3 others

297 One way of expressing the connectivity of records in a dataset is to implement  $K$  as a  
 298 simplicial complex, which is a set of simplices such as those shown in figure 21. The  
 299 advantage of triangulation is that it is general enough to work for more complex topology  
 300 based visualization methods [22] while also providing a consistent interface of vertices, edges,  
 301 and faces for  $\xi$  to map into. When triangulated, the simplices encode the continuity in the  
 302 data

simplex	continuity	$\tau$
vertex	discrete	$\tau(k)$
edge	1D	$\tau(k, \alpha)$
face	2D	$\tau(k, \alpha, \beta)$

Table 2

303 such that each section is bound to a simplex  $k \in K$ . As shown in table 2, in a 1D  
 304 continuous spaces each  $\tau$  lies distance  $\alpha$  along edge  $k$ , while in a 2D continuous space each  
 305  $\tau$  lies at coordinate  $\alpha, \beta$  on the face  $k$ . This is directly analogous to indexing to express  
 306 connectivity in N-D arrays, while also natively supporting graphs and trees as they are  
 307 simplicial complices of nodes and edges. Path connected components are then sections  
 308 where edges or faces meet.

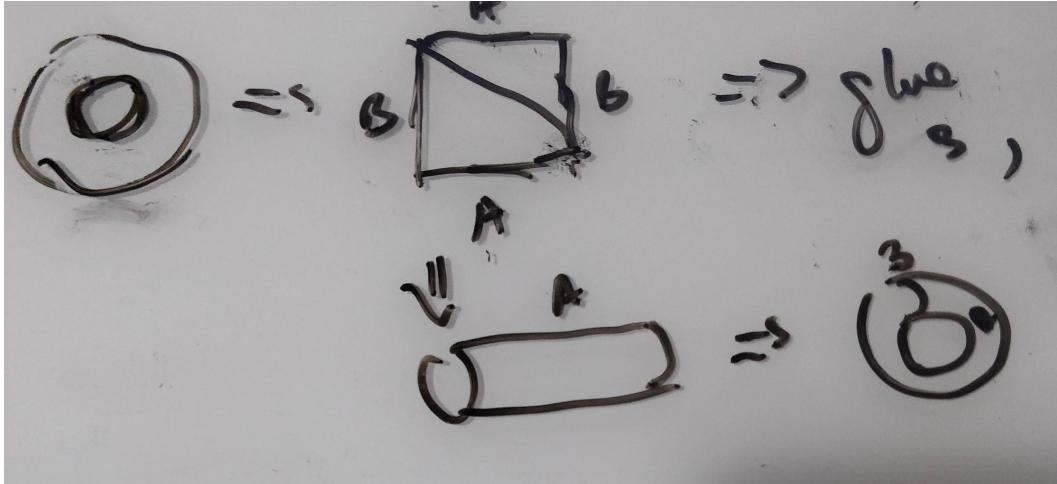


Figure 22: The torus  $E$  is unraveled into a simplicial complex of 2 faces  $K$ . Transition functions are defined on the edges of  $K$  such that surface can be glued back into the torus.  
**add cross sections  $a$  and  $b$  to ring and color same as edges in complex**

309 One way of encoding the torus in figure 22 while retaining the continuity of both cross  
 310 sections  $a, b$  is to unravel it into a simplicial complex of two triangles with labeled edges.  
 311 Transition functions  $\delta$  are defined on the edges such that  $a$  can be glued to  $a'$  and  $b$  to  $b'$  to  
 312 reconstruct the torus. This simplicial complex is then used as the base space encoding the  
 313 continuity of data that lies in the torus. A constraint on the transition functions is that the  
 314 monoid actions on the fibers on the edges of  $E$  are commutative  $M * F \mapsto \delta(MF) = M * \delta(F)$

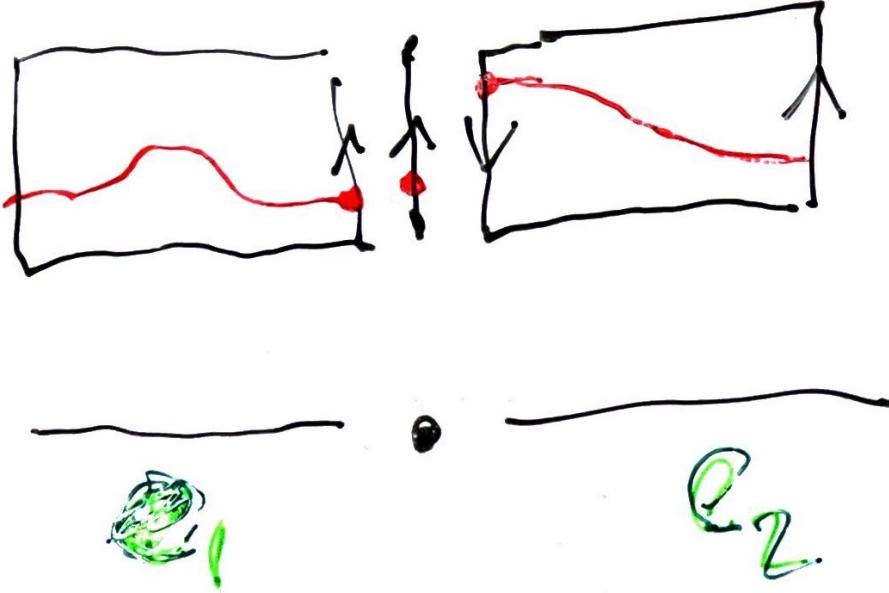


Figure 23: Many non-trivial spaces can be made locally trivial by dividing  $E$  into locally trivial subspaces and defining transition functions between the edges on  $K$  for how to glue the two subspaces such that the  $\tau$  are continuous.

315     Another advantage of triangulation is that it provides a way to encode non-trivial  
 316     structures such as the Möbius strip[33]. As shown in figure 23, one way of making the  
 317     Möbius strip trivial is to separate it into two spaces  $E_1$  and  $E_2$  and then define transition  
 318     functions that specify that the edges of  $E_1$  need to be reversed to line up with  $E_2$  such that  
 319     the sections along the edges meet. As with the torus, the transition functions must preserve  
 320     monoid commutativity.

321 4 Prototype Implementation: Matplottoy

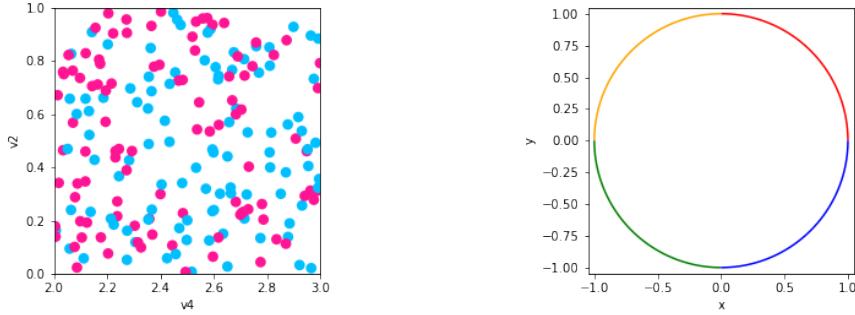


Figure 24: Scatter plot and line plot implemented using prototype artists and data models, building on Matplotlib rendering.

322 To prototype our model, we implemented the artist classes for the scatter and line plots  
323 shown in figure 24 because they differ in every attribute: different visual channels  $\nu$  that  
324 composite to different marks  $Q$  with different continuities  $\xi$ . We make use of the Matplotlib  
325 figure and axes artists [24, 25] so that we can initially focus on the data to graphic trans-  
326 formations.

327 To generate the images in figure 24, we instantiate `fig`, `ax` artists that will contain the  
328 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>
--	---

---

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

### 334 4.1 Artist Class $A'$

335 The artist is the piece of the matplotlib architecture that constructs an internal representation  
 336 of the graphic that the render then uses to draw the graphic. In the prototype artist,  
 337 `transform` is a dictionary of the form `{parameter:(variable, encoder)}` where parameter  
 338 is a component in  $P$ , variable is a component in  $F$ , and the  $\nu$  encoders are passed in as  
 339 functions or callable objects. The data bundle  $E$  is passed in as a `data` object. By binding  
 340 `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)

```

---

341 The data is fetched in section  $\tau$  via a `view` method on the data because the input to the  
 342 artist is a section on  $E$ . The return `view` object has a `get` method to support querying for  
 343 components that are not in  $F$  which we exploit to support parameters in the visual fiber  
 344 that are not bound to fiber components in  $F$ . The  $\nu$  functions are then applied to the data  
 345 to generate the  $\mu=\text{visual}$  input to  $Q$ . An explicit  $\xi$  is not implemented since that would  
 346 mean copying a single  $\mu$  on  $k$  to all the associated  $s$ , as illustrated in figure 11, and that is  
 347 unnecessary overhead for these scatter and line plots. In  $\hat{Q}=\text{assemble}$  the artist generates  
 348 instructions for the render by setting the attributes that are related to the graphic. These

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

355 The `Point` artist builds on `collection` artists because collections are optimized to ef-  
 356 ficiently draw a sequence of primitive point and area marks. In this prototype, the scatter  
 357 marker shape is fixed as a circle, and the only visual fiber components are  $x$  and  $y$  position,  
 358 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)
```

---

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

368     The only difference between the `Point` and `Line` objects is in the `view` and `assemble`  
369     function because line has different continuity from scatter and is represented by a different  
370     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)
```

---

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

## 377     4.2 Encoders $\nu$

378     As mentioned above, the encoding dictionary is specified by the visual fiber component, the  
379     corresponding data fiber component, and the mapping function. The `visual` parameter serves  
380     as the dictionary key because the visual representation is constructed from the encoding  
381     applied to the data  $\mu = \nu \circ \tau$ . For the scatter plot, the mappings for the visual fiber  
382     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))}
```

---

383 where the position  $(x,y)$   $\nu$  transformers are identity functions. The size  $s$  transformer is not  
 384 acting on a component of  $F$ , instead it is a  $\nu$  that returns a constant value. While size could  
 385 be embedded inside the `assembly` function, it is added to the transformers to illustrate user  
 386 configured visual parameters that could either be constant or mapped to a component in  $F$ .  
 387 The identity and constant  $\nu$  are explicitly implemented here to demonstrate their implicit  
 388 role in the visual pipeline, but they could be optimized away. More complex encoders can  
 389 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]
```

---

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

394 A test for equivariance can be implemented trivially such that it is independent of data  
 395 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)
```

---

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

### 401 4.3 Data $E$

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

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

405     2. have a that returns an atomic object that encapsulates  $\tau$   
 406     3. the view object must have that returns a fiber component  
 407 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

```

---

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

414     To generate the scatter plot in figure 24, we fully specify a dataset with random keys  
 415     and values in a section chosen at random from the corresponding fiber component. The  
 416     fiberbundle FB is a class level attribute since all instances of `codeVertexSimplex` come from  
 417     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

```

---

418 The view method returns a dictionary where the key is a fiber component name and the  
 419 value is a list of values in the fiber component. The table is built one call to `tau` at a time,  
 420 guaranteeing that all the fiber component values are over the same  $k$ . Table has a `get`  
 421 method as it is a method on Python dictionaries. In contrast, the line in `EdgeSimplex` is  
 422 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)

```

---

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

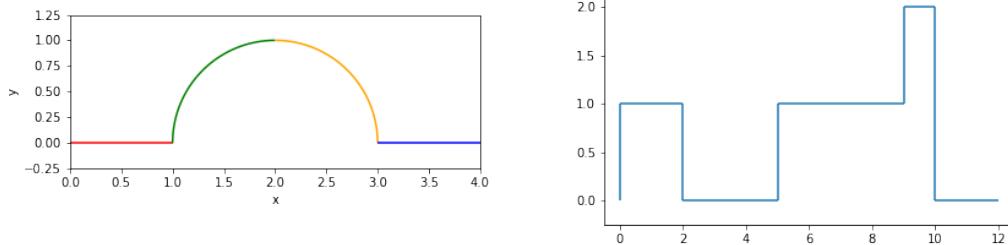


Figure 25: Continuous and discontinuous lines as defined by different data models, but generated with the same  $A'=\text{artist}$

429 The graphics in figure 25 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

```

---

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

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

---

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

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

---

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

#### 438 4.4 Case Study: Penguins

439 For this case study, we use the Palmer Penguins dataset[20, 23] since it is multivariate and  
440 has a varying number of penguins. We use a version of the data packaged as a pandas  
441 dataframe[39, 44] since that is a very commonly used Python labled data structure. The  
442 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

```

---

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

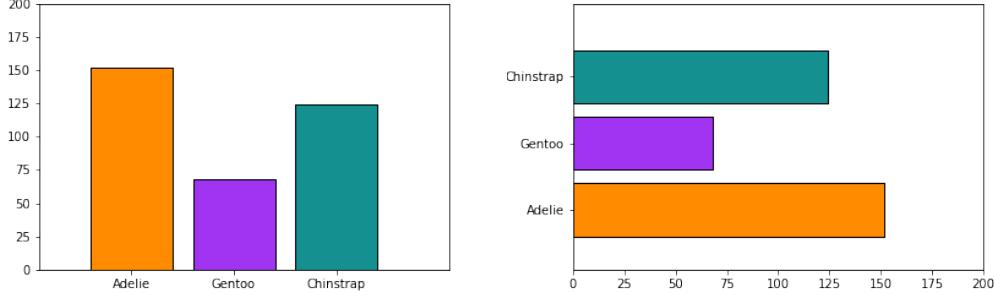


Figure 26: Frequency of Penguin types visualized as discrete bars.

446     The bar charts in figure 26 are generated with a `Bar` artist. The have the same required  
 447      $P$  components of (position, length). In of `Bar` an additional parameter is set, `orientation`  
 448     which only applies holistically to the graphic and never to individual data parameters.  
 449     Explicitly differentiate between parameters in  $V$  and ones that are only in  $\hat{Q}$  is another way  
 450     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

```

---

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

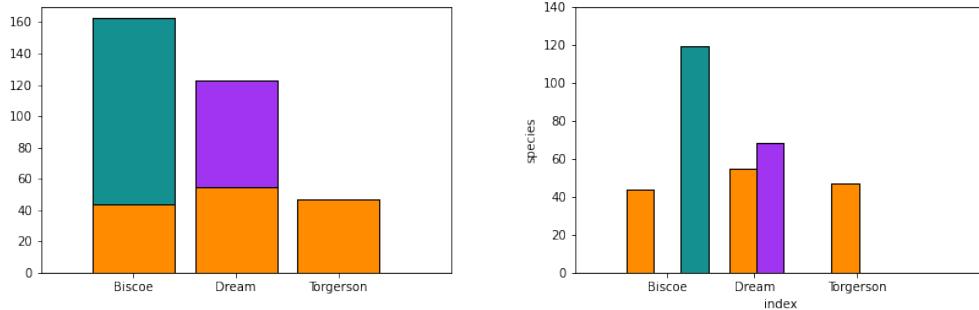


Figure 27: Penguin count disaggregated by island and species

456 For example, the `MultiBar` artist that makes figure 27 reuses `_make_bars` but does  
457 not reuse the `assemble` function because the composition of elements forces fundamental  
458 differences in glyph construction. As demonstrated in the `init`, the composite bar chart  
459 has orientation and whether it is stacked or not. While the stacked bar chart and the grouped  
460 bar chart could be separate artists, as demonstrated they share so much overlapping code  
461 that it is far less redundant to implement them together. looking at the mess that is this  
462 code, I'm a) not convinced these should be combined b) no longer convinced this provides  
463 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

```

---

464 In the `__draw__`, a utility function is used for conversions, but the length transforms  
 465 are held until after assembly because the length is computed by adding the current length  
 466 to the previous and many transforms are not distributable such that  $\nu(x_0 + x_1 + x_2) =$   
 467  $\nu(x_0) + \nu(x_1) + \nu(x_2)$ . Inside `assemble`, the glyphs are either shifted vertically (`stacked`)  
 468 or horizontally (`grouped`) such that the positions are recorded and added to with the next  
 469 group. This function allows multiple columns to be mapped to a visual parameter, but it  
 470 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'})])

```

---

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

---

```

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

```

---

476 where `trans` is the same dictionary for both stacked and grouped version, as is the  
 477 `DataFrameSection` object `table`. The only difference between the two versions is the  
 478 `stacked` flag, and the only difference between figures 26 is the `orientation` argument. By  
 479 decomposing the architecture into data, visual encoding, and assembly steps, we are able  
 480 to build components that are more flexible and also more self contained than the existing  
 481 code base.

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

## 484 References

- 485 [1] *[A Series of Statistical Charts Illustrating the Condition of the Descendants of Former*  
*African Slaves Now in Residence in the United States of America] Negro Business Men*  
*in the United States.* eng. <https://www.loc.gov/item/2014645363/>. Image.
- 486 [2] *[A Series of Statistical Charts Illustrating the Condition of the Descendants of Former*  
*African Slaves Now in Residence in the United States of America] Negro Population*  
*of the United States Compared with the Total Population of Other Countries /.* eng.  
<https://www.loc.gov/item/2013650368/>. Image.
- 487 [3] *Action in nLab.* [https://ncatlab.org/nlab/show/action#actions\\_of\\_a\\_monoid](https://ncatlab.org/nlab/show/action#actions_of_a_monoid).
- 488 [4] Professor Denis Auroux. “Math 131: Introduction to Topology”. en. In: (), p. 113.
- 489 [5] Jacques Bertin. “II. The Properties of the Graphic System”. English. In: *Semiology of*  
*Graphics.* Redlands, Calif.: ESRI Press, 2011. ISBN: 978-1-58948-261-6 1-58948-261-1.
- 490 [6] M. Bostock and J. Heer. “Protovis: A Graphical Toolkit for Visualization”. In: *IEEE*  
*Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1121–  
 491 1128. ISSN: 1941-0506. DOI: [10.1109/TVCG.2009.174](https://doi.org/10.1109/TVCG.2009.174).
- 492 [7] M. Bostock, V. Ogievetsky, and J. Heer. “D<sup>3</sup> Data-Driven Documents”. In: *IEEE*  
*Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pp. 2301–  
 493 2309. ISSN: 1941-0506. DOI: [10.1109/TVCG.2011.185](https://doi.org/10.1109/TVCG.2011.185).
- 494 [8] D. M. Butler and M. H. Pendley. “A Visualization Model Based on the Mathematics  
 495 of Fiber Bundles”. en. In: *Computers in Physics* 3.5 (1989), p. 45. ISSN: 08941866.  
 496 DOI: [10.1063/1.168345](https://doi.org/10.1063/1.168345).
- 497 [9] David M. Butler and Steve Bryson. “Vector-Bundle Classes Form Powerful Tool  
 498 for Scientific Visualization”. en. In: *Computers in Physics* 6.6 (1992), p. 576. ISSN:  
 499 08941866. DOI: [10.1063/1.4823118](https://doi.org/10.1063/1.4823118).
- 500 [10] L. Byrne, D. Angus, and J. Wiles. “Acquired Codes of Meaning in Data Visualization  
 501 and Infographics: Beyond Perceptual Primitives”. In: *IEEE Transactions on Visualiza-*  
*502 tion and Computer Graphics* 22.1 (Jan. 2016), pp. 509–518. ISSN: 1077-2626. DOI:  
 503 [10.1109/TVCG.2015.2467321](https://doi.org/10.1109/TVCG.2015.2467321).

- 512 [11] Sheelagh Carpendale. *Visual Representation from Semiology of Graphics by J. Bertin*.  
 513 en.
- 514 [12] John M Chambers et al. *Graphical Methods for Data Analysis*. Vol. 5. Wadsworth  
 515 Belmont, CA, 1983.
- 516 [13] William S. Cleveland. "Research in Statistical Graphics". In: *Journal of the American*  
 517 *Statistical Association* 82.398 (June 1987), p. 419. ISSN: 01621459. DOI: 10.2307/  
 518 2289443.
- 519 [14] William S. Cleveland and Robert McGill. "Graphical Perception: Theory, Experi-  
 520 mentation, and Application to the Development of Graphical Methods". In: *Journal of the*  
 521 *American Statistical Association* 79.387 (Sept. 1984), pp. 531–554. ISSN: 0162-1459.  
 522 DOI: 10.1080/01621459.1984.10478080.
- 523 [15] "Connected Space". en. In: *Wikipedia* (Dec. 2020).
- 524 [16] *Data Representation in Mayavi — Mayavi 4.7.2 Documentation*. <https://docs.enthought.com/mayavi/mayavi/d>
- 525 [17] T. W. E. B. Du Bois Center at the University of Massachusetts, W. Battle-Baptiste,  
 526 and B. Rusert. *W. E. B. Du Bois's Data Portraits: Visualizing Black America*. Prince-  
 527 ton Architectural Press, 2018. ISBN: 978-1-61689-706-2.
- 528 [18] Michael Friendly. "A Brief History of Data Visualization". en. In: *Handbook of Data*  
 529 *Visualization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 15–56. ISBN:  
 530 978-3-540-33036-3 978-3-540-33037-0. DOI: 10.1007/978-3-540-33037-0\_2.
- 531 [19] Berk Geveci et al. "VTK". In: *The Architecture of Open Source Applications* 1 (2012),  
 532 pp. 387–402.
- 533 [20] Kristen B. Gorman, Tony D. Williams, and William R. Fraser. "Ecological Sexual  
 534 Dimorphism and Environmental Variability within a Community of Antarctic Pen-  
 535 guins (Genus Pygoscelis)". In: *PLOS ONE* 9.3 (Mar. 2014), e90081. DOI: 10.1371/journal.pone.0090081.
- 537 [21] Marcus D. Hanwell et al. "The Visualization Toolkit (VTK): Rewriting the Rendering  
 538 Code for Modern Graphics Cards". en. In: *SoftwareX* 1-2 (Sept. 2015), pp. 9–12. ISSN:  
 539 23527110. DOI: 10.1016/j.softx.2015.04.001.
- 540 [22] C. Heine et al. "A Survey of Topology-Based Methods in Visualization". In: *Computer*  
 541 *Graphics Forum* 35.3 (June 2016), pp. 643–667. ISSN: 0167-7055. DOI: 10.1111/cgf.  
 542 12933.
- 543 [23] Allison Marie Horst, Alison Presmanes Hill, and Kristen B Gorman. *Palmerpenguins:*  
 544 *Palmer Archipelago (Antarctica) Penguin Data*. Manual. 2020. DOI: 10.5281/zenodo.  
 545 3960218.
- 546 [24] J. D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science*  
 547 *Engineering* 9.3 (May 2007), pp. 90–95. ISSN: 1558-366X. DOI: 10.1109/MCSE.2007.  
 548 55.
- 549 [25] John Hunter and Michael Droettboom. *The Architecture of Open Source Applications*  
 550 (*Volume 2*): *Matplotlib*. <https://www.aosabook.org/en/matplotlib.html>.
- 551 [26] "Jet Bundle". en. In: *Wikipedia* (Dec. 2020).
- 552 [27] John Krygier and Denis Wood. *Making Maps: A Visual Guide to Map Design for GIS*.  
 553 English. 1 edition. New York: The Guilford Press, Aug. 2005. ISBN: 978-1-59385-200-9.
- 554 [28] W A Lea. "A Formalization of Measurement Scale Forms". en. In: (), p. 44.

- 555 [29] *Locally Trivial Fibre Bundle - Encyclopedia of Mathematics*. [https://encyclopediaofmath.org/wiki/Locally\\_trivial](https://encyclopediaofmath.org/wiki/Locally_trivial)
- 556 [30] Jock Mackinlay. “Automating the Design of Graphical Presentations of Relational  
557 Information”. In: *ACM Transactions on Graphics* 5.2 (Apr. 1986), pp. 110–141. ISSN:  
558 0730-0301. DOI: 10.1145/22949.22950.
- 559 [31] JOCK D. MACKINLAY. “AUTOMATIC DESIGN OF GRAPHICAL PRESENTA-  
560 TIONS (DATABASE, USER INTERFACE, ARTIFICIAL INTELLIGENCE, INFOR-  
561 MATION TECHNOLOGY)”. English. PhD Thesis. 1987.
- 562 [32] Connie Malamed. *Information Display Tips*. [https://understandinggraphics.com/visualizations/information-](https://understandinggraphics.com/visualizations/information-display-tips/)  
563 [display-tips/](#). Blog. Jan. 2010.
- 564 [33] *Möbius Strip in nLab*. <https://ncatlab.org/nlab/show/M%C3%B6bius+strip>.
- 565 [34] “Monoid”. en. In: *Wikipedia* (Jan. 2021).
- 566 [35] T Munzner. “Marks and Channels”. In: *Visualization Analysis and Design*, pp. 94–  
567 114.
- 568 [36] Tamara Munzner. “Ch 2: Data Abstraction”. In: *CPSC547: Information Visualization, Fall 2015-2016* ().
- 570 [37] Tamara Munzner. *Visualization Analysis and Design*. AK Peters Visualization Series.  
571 CRC press, Oct. 2014. ISBN: 978-1-4665-0891-0.
- 572 [38] Jana Musilová and Stanislav Hronek. “The Calculus of Variations on Jet Bundles as a  
573 Universal Approach for a Variational Formulation of Fundamental Physical Theories”.  
574 In: *Communications in Mathematics* 24.2 (Dec. 2016), pp. 173–193. ISSN: 2336-1298.  
575 DOI: 10.1515/cm-2016-0012.
- 576 [39] Muhammad Chenariyan Nakhaee. *Mcnakhaee/Palmerpenguins*. Jan. 2021.
- 577 [40] Donald A. Norman. *Things That Make Us Smart: Defending Human Attributes in the Age of the Machine*. USA: Addison-Wesley Longman Publishing Co., Inc., 1993. ISBN:  
578 0-201-62695-0.
- 580 [41] Z. Pousman, J. Stasko, and M. Mateas. “Casual Information Visualization: Depictions  
581 of Data in Everyday Life”. In: *IEEE Transactions on Visualization and Computer  
582 Graphics* 13.6 (Nov. 2007), pp. 1145–1152. ISSN: 1941-0506. DOI: 10.1109/TVCG.  
583 2007.70541.
- 584 [42] “Quotient Space (Topology)”. en. In: *Wikipedia* (Nov. 2020).
- 585 [43] P. Ramachandran and G. Varoquaux. “Mayavi: 3D Visualization of Scientific Data”.  
586 In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 40–51. ISSN: 1558-366X.  
587 DOI: 10.1109/MCSE.2011.35.
- 588 [44] Jeff Reback et al. *Pandas-Dev/Pandas: Pandas 1.0.3*. Zenodo. Mar. 2020. DOI: 10.  
589 5281/zenodo.3715232.
- 590 [45] “Retraction (Topology)”. en. In: *Wikipedia* (July 2020).
- 591 [46] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. “Declarative Inter-  
592 action Design for Data Visualization”. en. In: *Proceedings of the 27th Annual ACM  
593 Symposium on User Interface Software and Technology*. Honolulu Hawaii USA: ACM,  
594 Oct. 2014, pp. 669–678. ISBN: 978-1-4503-3069-5. DOI: 10.1145/2642918.2647360.
- 595 [47] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. “NIH Image to Im-  
596 ageJ: 25 Years of Image Analysis”. In: *Nature Methods* 9.7 (July 2012), pp. 671–675.  
597 ISSN: 1548-7105. DOI: 10.1038/nmeth.2089.

- 598 [48] “Semigroup Action”. en. In: *Wikipedia* (Jan. 2021).
- 599 [49] E.H. Spanier. *Algebraic Topology*. McGraw-Hill Series in Higher Mathematics.  
600 Springer, 1989. ISBN: 978-0-387-94426-5.
- 601 [50] David I Spivak. *Databases Are Categories*. en. Slides. June 2010.
- 602 [51] David I Spivak. “SIMPLICIAL DATABASES”. en. In: (), p. 35.
- 603 [52] “Stalk (Sheaf)”. en. In: *Wikipedia* (Oct. 2019).
- 604 [53] S. S. Stevens. “On the Theory of Scales of Measurement”. In: *Science* 103.2684 (1946),  
605 pp. 677–680. ISSN: 00368075, 10959203.
- 606 [54] Software Studies. *Culturevis/Imageplot*. Jan. 2021.
- 607 [55] *[The Georgia Negro] City and Rural Population. 1890*. eng. <https://www.loc.gov/item/2013650430/>.  
608 Image. 1900.
- 609 [56] *[The Georgia Negro] Negro Property in Two Cities of Georgia*. eng. <https://www.loc.gov/item/2013650443/>.  
610 Image. 1900.
- 611 [57] M. Tory and T. Moller. “Rethinking Visualization: A High-Level Taxonomy”. In:  
612 *IEEE Symposium on Information Visualization*. Oct. 2004, pp. 151–158. doi: 10.  
613 1109/INFVIS.2004.59.
- 614 [58] Edward R. Tufte. *The Visual Display of Quantitative Information*. English. Cheshire,  
615 Conn.: Graphics Press, 2001. ISBN: 0-9613921-4-2 978-0-9613921-4-7 978-1-930824-13-3  
616 1-930824-13-0.
- 617 [59] Jacob VanderPlas et al. “Altair: Interactive Statistical Visualizations for Python”. en.  
618 In: *Journal of Open Source Software* 3.32 (Dec. 2018), p. 1057. ISSN: 2475-9066. doi:  
619 10.21105/joss.01057.
- 620 [60] C. Ware. *Information Visualization: Perception for Design*. Interactive Technologies.  
621 Elsevier Science, 2019. ISBN: 978-0-12-812876-3.
- 622 [61] Eric W. Weisstein. *Similarity Transformation*. en. <https://mathworld.wolfram.com/SimilarityTransformation.html>  
623 Text.
- 624 [62] Hadley Wickham. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New  
625 York, 2016. ISBN: 978-3-319-24277-4.
- 626 [63] Leland Wilkinson. *The Grammar of Graphics*. en. 2nd ed. Statistics and Computing.  
627 New York: Springer-Verlag New York, Inc., 2005. ISBN: 978-0-387-24544-7.
- 628 [64] *Writing Plugins*. en. <https://imagej.net/Writing-plugins>.
- 629 [65] Brent A Yorgey. “Monoids: Theme and Variations (Functional Pearl)”. en. In: (),  
630 p. 12.