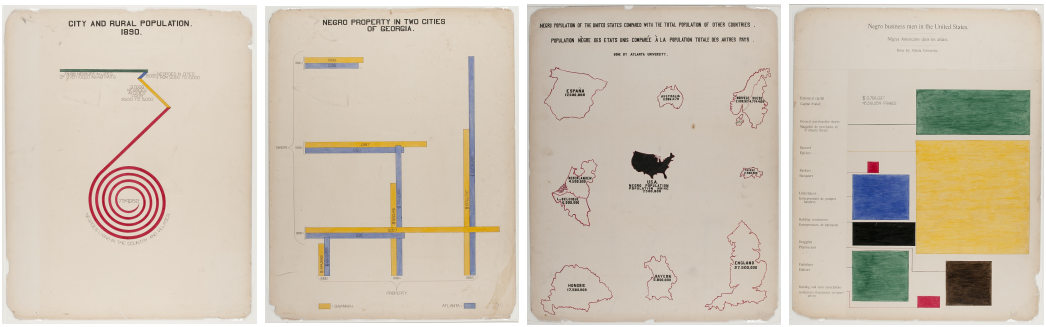
1 Introduction

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

2 Background



(a) (b) (c) (d)

Figure 1: Du Bois' data portraits[22] of post reconstruction Black American life exemplify that the fundemental 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, 69, 70]

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 [13]. 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[72]. 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[52]. 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 [49] which is displayed in figure 2.

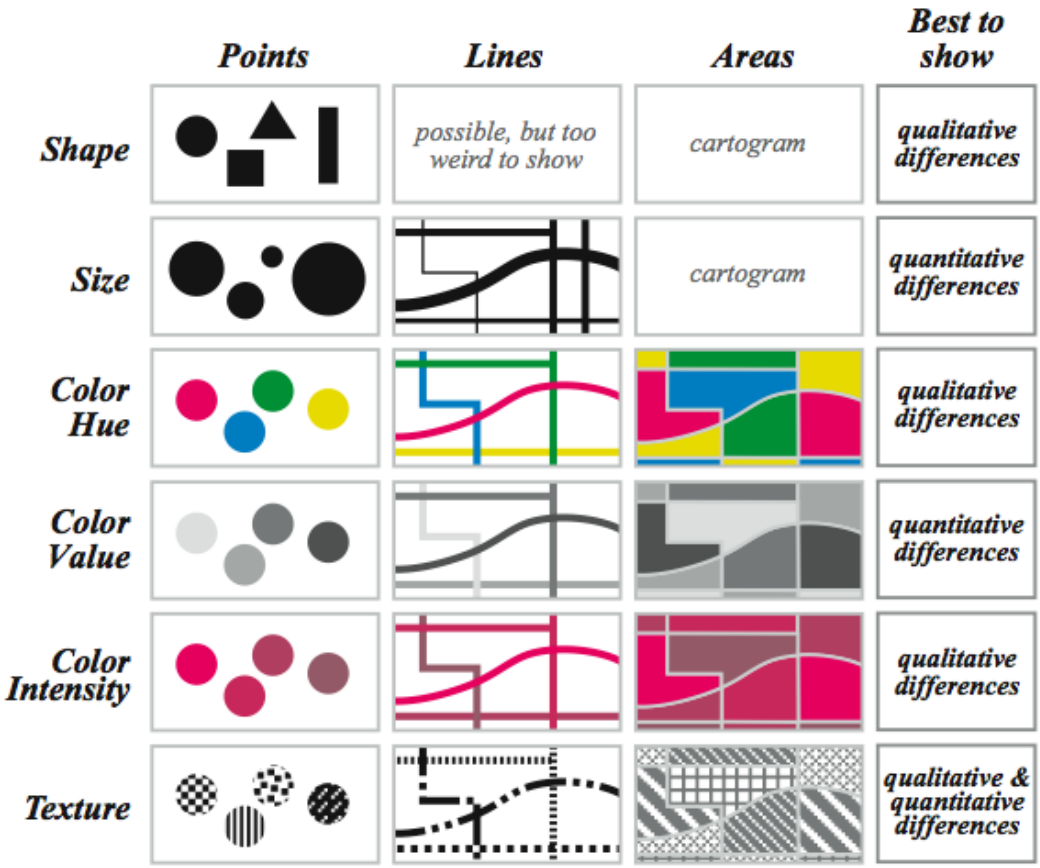


Figure 2: Retinal variables are a codification of how position, size, shape, color and texture are used to illustrate variations in the components of a visualization. The best to show column describes which types of information can be expressed in the corresponding visual encoding. This tabular form of Bertin's retinal variables is from Understanding Graphics [43] who reproduced it from *Making Maps: A Visual Guide to Map Design for GIS* [36]

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[41], 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 [17–19, 49]. Mackinlay's definition of expressiveness is

formalized at the visual encoding level, which as shown in figure 2 refers to the components of a graphic such as the color or position of a glyph. Bertin first classified these graphic components as retinal variables and discussed which types of data they can express [7] and how they are composited on point, line, and area graphical marks, as shown in figure 2 correspond. Marks can be generalized to glyphs, which are graphical objects that convey one or more attributes of the data entity mapped to it[49, 74]. Mackinlay's expressiveness criteria is well defined for the visual variables, such that he suggests the viability of a strict encoding relation that is a homomorphic mapping which preserves some binary operator from one domain to another [42]. We expand on this suggestion by proposing that monoid action equivariance is a strict condition of building valid encoders. Mackinlay does not provide a generalized criteria for plot types, instead embedding the requirements within the definition of the charts.

2.1 Data

Tory and Möller propose that assumptions about the structure of data are built into the visual algorithms that display that information [71]. Specifically they note that discrete and continuous data and their attributes form a discipline independent design space [53].

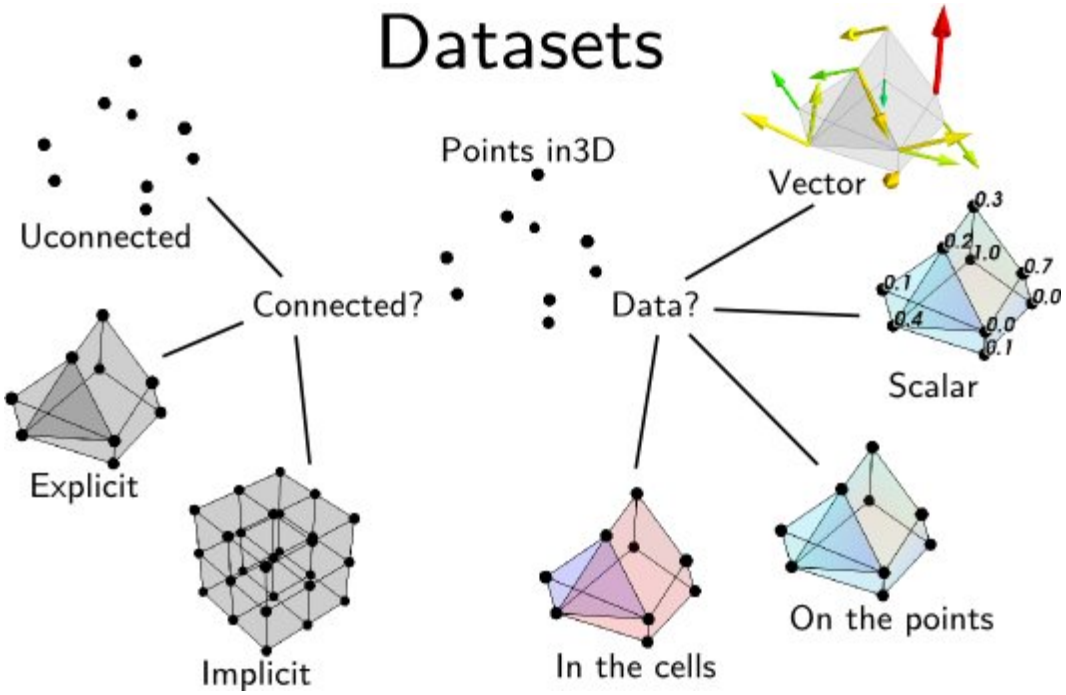


Figure 3: One way to describe data is by the connectivity of the points in the dataset. A database for example is often discrete unconnected points, while an image is an implicitely connected 2D grid. This image is from the Data Representation chapter of the MayaVi 4.7.2 documentation.[21]

As shown in figure 3, there are many types of continuity in data. 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. Each *component* of the record is a single object, such as a temperature, a color, or an image. The way in which these records are connected is the *connectivity*, *continuity*, or more generally *topology*.

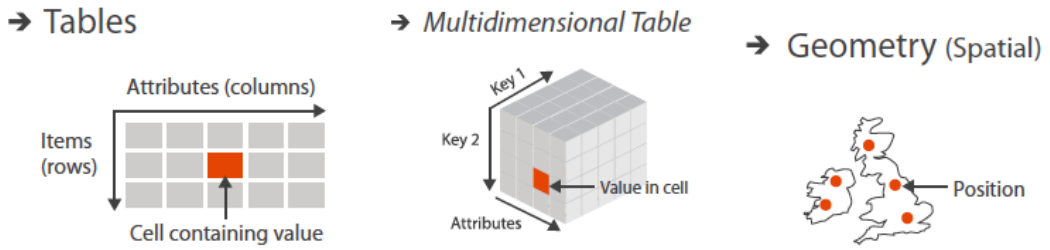


Figure 4: Keys are unique lookup values used to find individual observations in the dataset. Keys are positional references, and can be coordinates on a map or unique values such as a primary key in a database or a (time, latitude, longitude) index in a data cube. Image modified from a diagram from Munzner's *Visualization Analysis and Design* [49]

Often this topology has metadata associated with it, as shown in figure 4. Munzner denotes this metadata as *keys* that can be used to locate the record, and further to locate the specific *value* [48]. In figure 4 tables have keys identifying the row and column and sometimes also the depth, while maps have a location key. We propose that information rich keys such as location are additional components of the record, and that instead there are coordinate free structural keys that identify the location of the record within a dataset of any continuity.

In this work, we extend Butler's topology driven representation of visualization data [11, 12]. Butler proposes that fiber bundles are a good model for visualization data because it allows for encoding the connectivity seperately from the records and supports discrete and ND continuous datasets. Since Butler's model lacks a robust way of describing the components of the record, we fold in Spivak's Simplacial formulation of databases [64, 65] so that we can encode a schema like discription of the data in the fiber bundle. We then propose criteria on expressivity that take into account both the components and the continuity of the data.

2.2 Tools

A motivator for this work is that currently Matplotlib carries implicit assumptions about data continuity in how each function interfaces with the input data. This work proposes a unified internal representation that encodes connectivity in a common interface. Matplotlib aims to natively support data of varying connectivities, so tools primarily concerned with visualizing relational data of the type found in databases are insufficient models. Many of these tools are built on top of Wilkenson's Grammar of Graphics [78] which itself incorporates Mackinlay's A Presentation Tool (APT) display algebra; GoG derivative include ggplot[76], protovis[8] and D3 [9], vega[60] and altair[73]. While many of these tools support images, the first class data container is a table like object of discrete records. Tools that primarily support images are also insufficient. For example ImageJ[61] and the ImagePlot[68]macro have some support for visualizing non image components of a complex

data set, but mostly in service to the image being visualized. Plugins exist, but must work around the everything is an image data model[80]. There are also visualization tools that do not carry implicit assumptions about structure, for example vtk[25, 29] and its derivatives such as MayaVi[56]. Not totally positive but I think VTK is deeply coupled to the renderer and that's why we're not using it as a model, but not positive and this is literally what Marc worked on...

2.3 contribution

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

3 Topological Artist Model

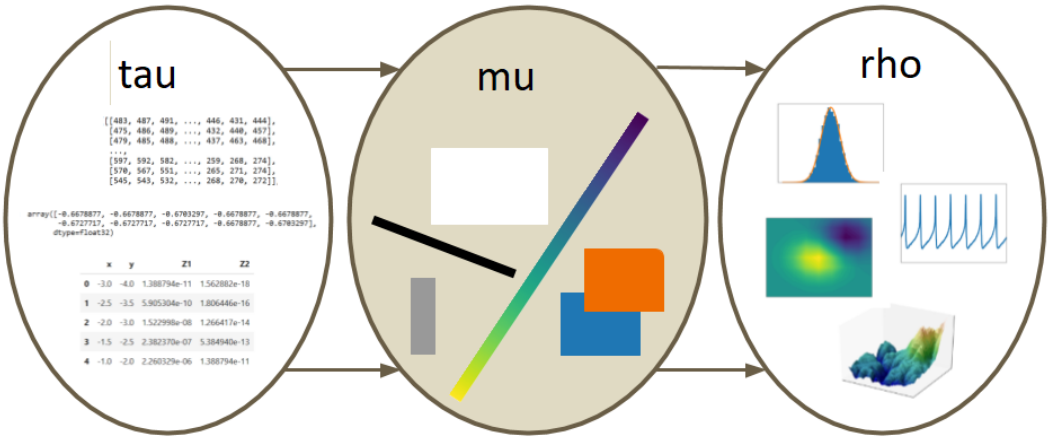


Figure 5: Visualization is equivariant maps between data and visual encoding of the variables and assembly of those encodings into a graphic. not gonna name these bubbles tau, mu, rho, but might keep the same basic structure of different types of data and encodings

can express continuous functions & not just discrete points Visualization is generally thought of as structure preserving maps from data into graphics, and in this section we formally define that structure and how it is preserved via equivariant maps. We can then specify that a faithful visual mapping is structure preserving, and apply these constraints to visualizations we may want to develop or implement. We model the data, visual characteristic, and graphic stages of visualization, shown in figure 5, as topological structures that encapsulate types of variables and continuity; by doing so we can develop implementations that keep track of

both in ways that let us distribute computation while still allowing assembly and dynamic update of the graphic.

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

$$\mathscr{A} : \mathscr{E} \to \mathscr{H} \tag{1}$$

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

3.1 Data Space E

We build on Butler's proposal of using fiber bundles as a common data representation format for visualization data[11, 12] because fiber bundles are mathematical structures that are flexible enough express all the types of data described in section **??**.

We model data as the fiber bundle (E, K, π, F), where E F and K are topological spaces that encode

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

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

τ collections of records (3.1.4).

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

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

that binds the variables F continuity K. The fiber bundles mentioned in this work are assumed to be trivial[38, 63], unless otherwise specified, because the trivial bundle is $E = K \times F$ such that extra structure in the total space E falls out and discussion can be focused on the fiber and base space.

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 [65] 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 $\{T_0, \ldots, T_n\} \in \mathbf{DT}$, where $\mathbf{DT}$ are the data types of the variables in the dataset. He then defines the single variable set $\mathbb{U}_\sigma$

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

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

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

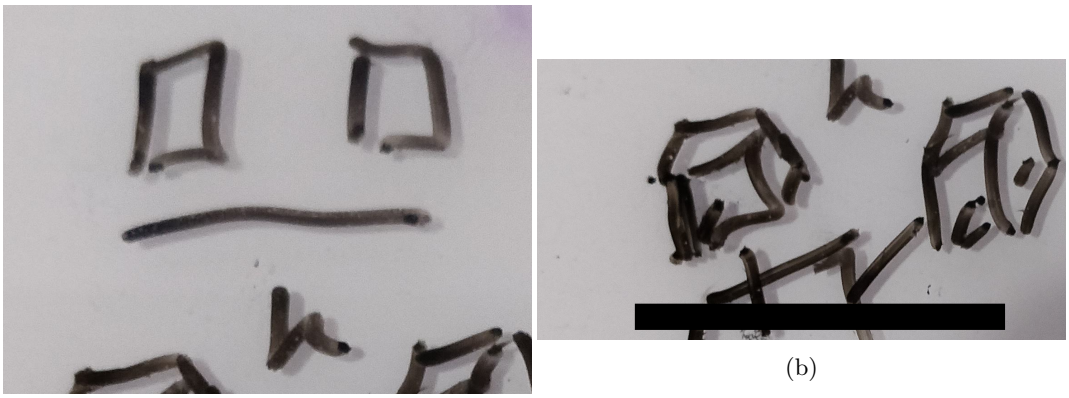
where σ is the schema binding variable name c to its datatype T. A dataset with multiple variables has a fiber that is the cartesian cross product of $\mathbb{U}_\sigma$ applied to all the columns:

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

which is equivalent to

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

which allows us to decouple F into components F_i.



(a) (b)

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 $\mathbb{U}_{\texttt{datetime}} = \mathbb{R}^+$. Temperature values are real numbers $\mathbb{U}_{\texttt{float}} = \mathbb{R}$. The fiber is

$$\mathbb{U} = \mathbb{R}^+ \times \mathbb{R} \tag{7}$$

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

$$F = \mathbb{R}^+ \times \mathbb{R}^2 \tag{8}$$

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

3.1.2 Measurement Scales: Monoid Actions

After specifying F we next describe the ways in which we can transform the values by identifying the monoid actions M on the F. We use monoids as the abstraction because they encode composibilty, which maps well to the data transformation process in a software library [81].

A monoid [46] M_i is a set with an associative binary operator $* : M_i \times M_i \to M_i$. 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, 62] of M_i is a set F_i with an action $\bullet : M \times F_i \to F_i$ with the properties:

$$\textbf{associativity} \text{ for all } f, g \in M_i \text{ and } x \in F_i,\ f \bullet (g \bullet x) = (f * g) \bullet x$$
$$\textbf{identity} \text{ for all } x \in F_i, e \in M_i,\ e \bullet x = x$$

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

$$M = M_0 \times \ldots \times M_i \times \ldots \times \ldots M_n \tag{9}$$

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

Steven's described the measurement scales[37, 67] in terms of the monoid actions on the measurements: nominal data is permutable, ordinal data is monotonic, interval data is translatable, and ratio data is scalable [75]. For example, given the fiber ($c = temperature$, $T =$ `float`, $\mathbb{U}_\sigma = \mathbb{R}$) which is interval data:

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

then the translation monoid actions on temperature satisfy the condition

$$\begin{array}{ccc} \mathbb{R} & & \\ {\scriptstyle x+1^\circ}\downarrow & \searrow^{(x+1^\circ)\circ(x+2^\circ)} & \\ \mathbb{R} & \xrightarrow[x+2^\circ]{} & \mathbb{R} \end{array} \tag{10}$$

where 1° and 2° are valid distances between two temperatures x.

3.1.3 Continuity: Base Space K

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

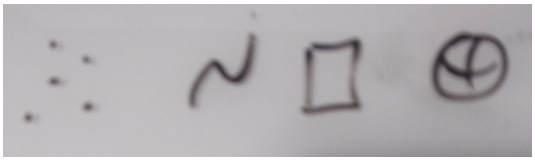


Figure 7: The topological base space K encodes the connectivity of the data space, for example if the data is independent points or a map or on a sphere

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

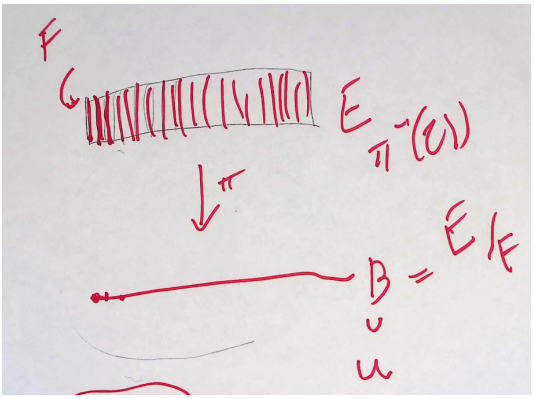


Figure 8: The base space E is divided into fiber segments F. The base space K acts as an index into the records in the fibers. this figure might be good all the way up top to lay out the components of fb

Formally K is the quotient space [55] of E meaning it is the finest space[4] such that every $k \in K$ has a corresponding fiber F_k[55]. In figure 8, E is a rectangle divided by vertical fibers F, so the minimal K for which there is always a mapping $\pi : E \to K$ is the line.

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

$$\pi : E_1 \oplus \ldots \oplus E_i \oplus \ldots \oplus E_n \to K \tag{11}$$

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

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

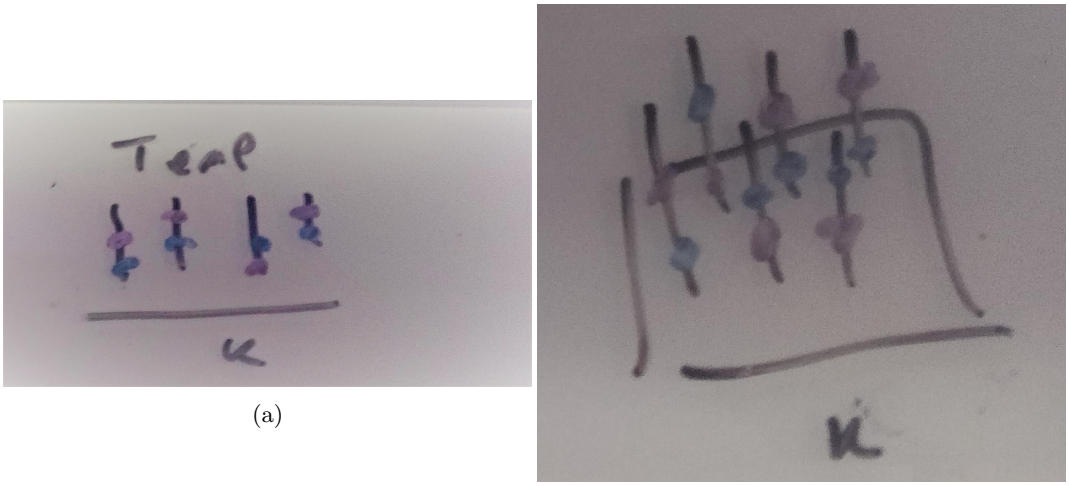
3.1.4 Data: Sections τ

While the fiber and base space describe the general structure of all data that lives in the fiber bundle, the sections $\tau : K \to E$ define the datasets that live in the fiber. We generalize Spivak's description of the section as a table of records [65] to any sort of structured dataset such that

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

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

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



(a)

(b)

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

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

where each section τ_i is a variable or set of variables.

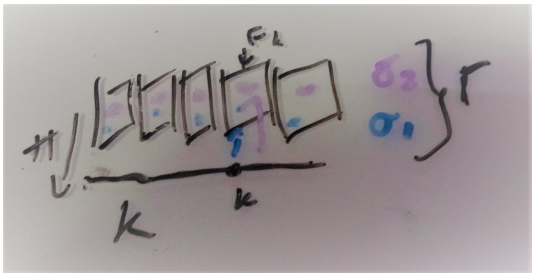


Figure 10: Fiber (time, temperature) with an interval K basespace. The sections τ_i and τ_j are constrained such that the time variable must be monotonic, which means each section is a timeseries of temperature values. They are included in the global set of sections $\tau_1, \tau_2 \in \Gamma(E)$

In the example in figure 10, the fiber is (*time*, *temperature*) as described in figure 6 and the base space is the interval K. The section τ_i resolves to a series of monotonically increasing in time records of (time, temperature) values. Section τ_j returns a different timeseries of (time, temperature) values. Both sections are included in the global set of sections $\tau_1, \tau_2 \in \Gamma(E)$.

3.1.5 Sheaf and Stalk

Often a graphic may need to be updated with live data or support zooming in on a segement of the dataset; to support working with a subset of data, we can use the sheaf $\mathcal{O}(E)$. All fiber bundles are locally trivial, which means that E restricted over a small enough neighborhood $U \subset K$ is a locally trivial bundle over U[38]. The sheaf $O(E)$ is the localized section of fibers $\iota^*\tau : U \to \iota^*E$

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

pulled back over the neighborhood U via the inclusion map $\iota : U \to K$. The localized section is the germ $\xi^*\tau$. The neighborhood of points k_i surrounding the point k the sheaf lies over is the stalk $\mathcal{F}_b$ [63, 66]. While E is only the fiber F_k over a specific k, the stalk includes nearby records because the sheaf lies over the neighborhood U. While this can be useful for visual transforms, often the extra needed information can be found in the smaller jet bundle $\mathcal{J}$ [35, 50]. For example, line thickness requires the derivative of the given position to be rendered, which can be found in $E' = E + \mathcal{J}(E)$

3.2 Graphic: H

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

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

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

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

3.2.2 Continuity of the Graphic S

An assumption of graphical representations is that they match the continuity of the data[24, 72], 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

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

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 α, β coordinates to figures

as the deformation retraction [59] $\xi : S \to K$ 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 β over a point α_i specifies the thickness of the line in S for the corresponding τ on K. The heatmap has the same continuity in data space and graphic space such that no extra dimensions are needed.

3.2.3 Renderering ρ

A section $\rho : S \to H$ defines a piece of the graphical representation of the data. Evaluated on a single s ρ returns a single element in DFor 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

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

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

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

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 ρ 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 $\rho_{rgb}(S_p)$.

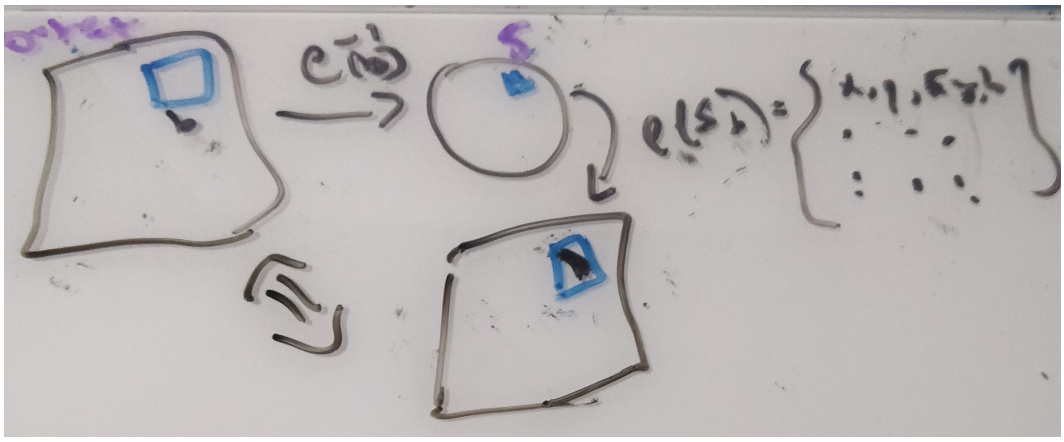


Figure 12: To render a graphic, a pixel p is selected in the display space, which is defined in the same coordinates as the x and y components in DThe inverse mapping $\rho_{xy}{}^{(}p)$ returns a region $S_p \subset S$. $\rho(S_p)$ returns the list of elements $(x,\, y,\, r,\, g,\, b) \in D$ that lie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

3.3 Artist

The artist is the function that converts data into graphics; its name is taken from the analogous part of Matplotlib[34] that builds visual elements to pass off to the renderer. The artist A is a mapping from E padded with data from $\mathcal{J}(E)$ to a graphic that is a section ρ in $\Gamma(H)$

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

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

1. ξ binding the continuity in the graphic to the continuity in the data (3.2.2)
2. ν conversion of data into visual characteristics (3.3.2)
3. Q assembly of visual variables into a glyph (3.3.3)

of the visual transformation illustrated in figure 5. The functions ξ ν and Q are defined such that they can be evaluated on a single section τ, which allows the artist to be implemented such that it does not need all the data. This allows for artists tuned to distributed and streaming data.

3.3.1 Visual Fiber Bundle V

The visual fiber bundle $(V,\, K,\, \pi,\, P)$ has section $\mu : V \to K$ that resolves to a visual variable [7, 47] in fiber P. The fiber space P is defined in terms of the parameters of the visualization specification- for example aesthetics in ggplot [76], channels in vega[60] or parameters in VTK[29] and Matplotlib.

$\boldsymbol{\nu_i}$	$\boldsymbol{\mu_i}$	$\boldsymbol{codomain(\nu_i)}$
position	x, y, z, theta, r	$\mathbb{R}$
size	linewidth, markersize	$\mathbb{R}^+$
shape	markerstyle	$\{f_0, \ldots, f_n\}$
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

Table 1 is a sample of the fiber space for Matplotlib [33]. A section $\mu$is a tuple of visual values that specifies the visual characteristics of a part of the graphic. For example, given a fiber of $\{xpos, ypos, color\}$ one possible section could be $\{.5, .5, (255, 20, 147)\}$. The $codomain(\nu_i)$ determines the monoid actions on μ_i. These fiber components are implicit in the library, by making them explicit as components of the fiber we can build consistent definitions and expectations of how these parameters behave.

3.3.2 Visual Channels ν

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

$$\{\nu_0, \ldots, \nu_n\} : \{\tau_0, \ldots, \tau_n\} \mapsto \{\mu_0, \ldots, \mu_n\} \tag{22}$$

where $\nu_i : \tau_i \mapsto \mu_i$ is an equivariant map such that there is a monoid homomorphism from F_i to $vfiber_i$. A validly constructed ν is one where the diagram of the monoid transform m

$$\begin{array}{ccc} E_i & \xrightarrow{\nu_i} & V_i \\ {\scriptstyle m_x}\downarrow & & \downarrow{\scriptstyle m_v} \\ E_i & \xrightarrow{\nu_i} & V_i \end{array} \tag{23}$$

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 $t(x) = x + 2$

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

$$.5 \neq .5 + .5 \tag{25}$$

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

```
[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, ν maps the strings to the emojis. For ν to be equivariant, a shuffle in the words should have an equivalent shuffle in the emojies, and a shuffle in the emojies should have an equivalent shuffle in the words.

shuffle of the symbols they are mapped to. To preserve ordinal and partial order monoid actions, ν 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, ν is equivariant under translation monoid actions if $\nu(x + c) = \nu(x) + \nu(c)$. For ratio data, there must be equivalent scaling $\nu(xc) = \nu(x)\nu(c)$.

3.3.3 Assembling Glyphs Q

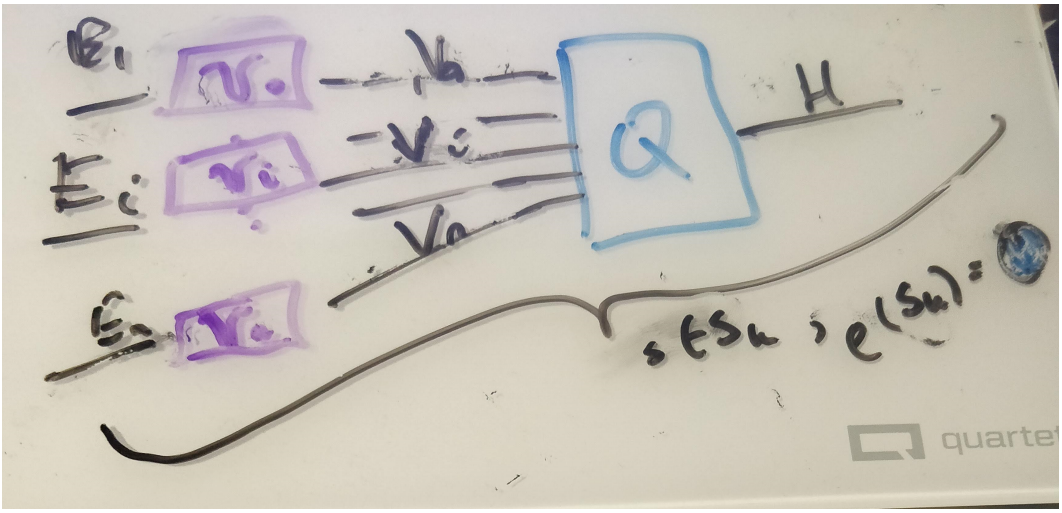


Figure 14: ν functions convert data τ_i to visual characteristics μ_i, then Q assembles μ_i into a graphic ρ such that there is a map ξ preserving the continuity of the data. ρ applied to a region of connected components S_j generates a graphical mark.

As shown in figure 14, the assembly function Q combines the fiber F_i wise ν transforms into a single glyph. Together, ν and Q are a map-reduce operation: map the data into their visual encodings, reduce the encodings into a glyph. As with ν the constraint on Q is that for every monoid actions on the input μ there is a corresponding monoid action on the output ρ.

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 Qsuch that if Q is applied to μ, μ' that generate the same ρ then the output of both sections acted 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

$$Q(\mu) = Q(\mu') \implies Q(m \circ \mu) = Q(m \circ \mu') \tag{26}$$

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

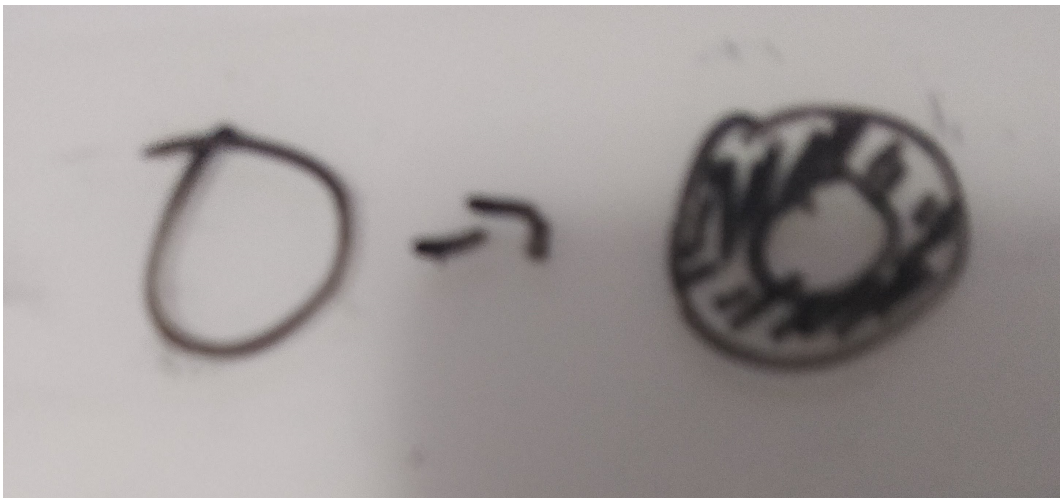


Figure 15: These two glyphs are generated by the same Q function, but differ in the value of the edge thickness parameter μ_i. A valid Q is one where a shift in μ_i is reflected in the glyph generated by ρ.

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

To output a mark [7, 15], Q is called with all the regions s that map back to a set of connected components $J \subset K$:

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

where the path[20] γ from k to j is a continuous function from the interval [0,1]. We define the mark as the graphic generated by $Q(S_j)$

$$H \underset{\rho(S_j)}{\overset{\longrightarrow}{\longleftarrow}} S_j \underset{\xi^{-1}(J)}{\overset{\xi(s)}{\longrightarrow\!\!\!\!\longleftarrow}} J_k \tag{28}$$

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

3.3.4 Sample Glyphs Q

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

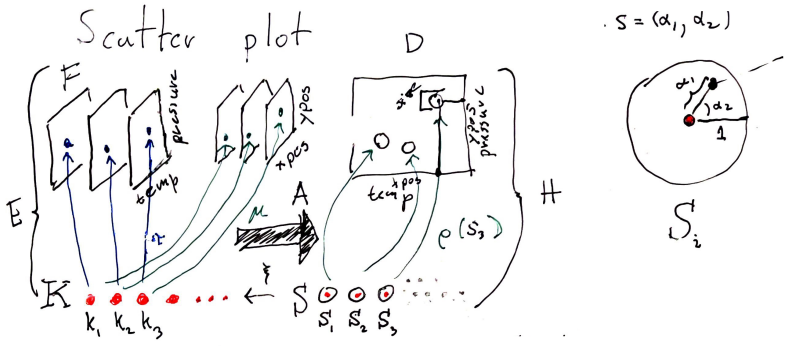


Figure 16: The data is discrete points (temperature, time). Via ν these are converted to (xpos, ypos) and pulled over discrete S. These values are then used to parameterize ρwhich returns a color based on the parameters (xpos,ypos) and position α, β on S_k that ρ is evaluated on.

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

$$x = size \bullet \alpha \bullet \cos(\beta) + xpos \tag{29}$$

$$y = size \bullet \alpha \bullet \sin(\beta) + ypos \tag{30}$$

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

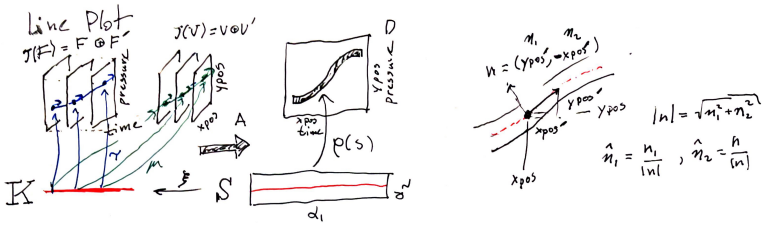


Figure 17: The line fiber ($time$, $temp$) is thickend with the derivative ($time'$, $temperature'$ because that information will be necessary to figure out the tangent to the point to draw a thick line. This is because the line needs to be pushed perpendicular to the tangent of (xpos, ypos). this is gonna move once this gets regenerated w/ labels The data is converted to visual characteristics (xpos, ypos). The α coordinates on S specifies the position of the line, the β coordinate specifies thickness.

The line plot $Q(xpos, \hat{n_1}, ypos, \hat{n_2})(\alpha, \beta)$ shown in fig 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

$$|n| = \sqrt{{n_1}^2 + {n_2}^2} \tag{31}$$

such that the normal is

$$\hat{n_1} = \frac{n_1}{|n|},\ \hat{n_2} = \frac{n_2}{|n|} \tag{32}$$

which yields components of ρ

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

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

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

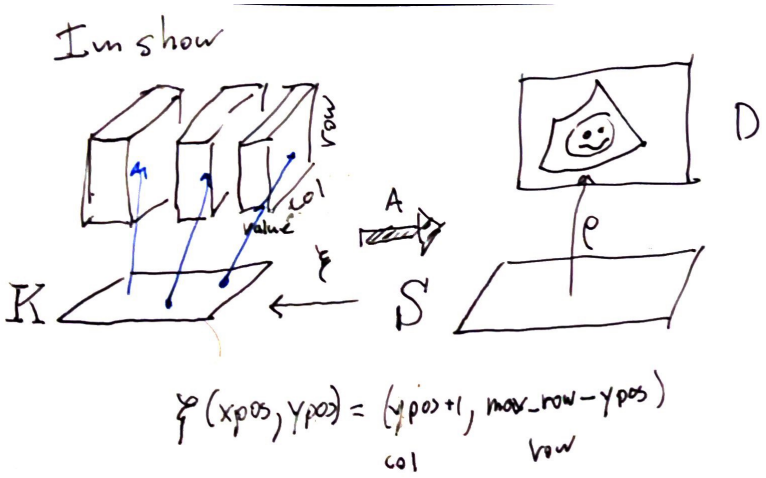


Figure 18: The only visual parameter a heatmap requires is color since ξ encodes the mapping between position in data and position in graphic.

The heatmap $Q(color)$ in figure 18 is a direct lookup $\xi : S \to K$ such that

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

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

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

where ξ 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.

3.3.5 Assembly factory $\hat{Q}$

As shown in eq 21, Q is a bundle map $Q : \xi^* V \to H$ where $\xi^* V$ and H are both bundles over S. Because $\xi^{-1}(k)$ $subset S$ such that many s go to one k, by definition of the pull back $\xi^* V\ |_{\xi^{-1}(k)} = \xi^{-1}(k) \times P$. This means that the fiber $V$is copied over the preimage in $\xi^{-1}(k)$. Given a section $\xi^* \mu$ pulled back from μ on $\pi : V \to K$ and $s \in \xi^{-1}(k)$, then the section $\xi^* \mu(s) = \xi^*(\mu(k))$ is the image of $(k, \mu(k)) \mapsto (s, \xi^* \mu(s))$ for all s where $\xi(s) = k$. This is illustrated in figure 19, where a μ over J is copied over all $\xi(s) = k$.

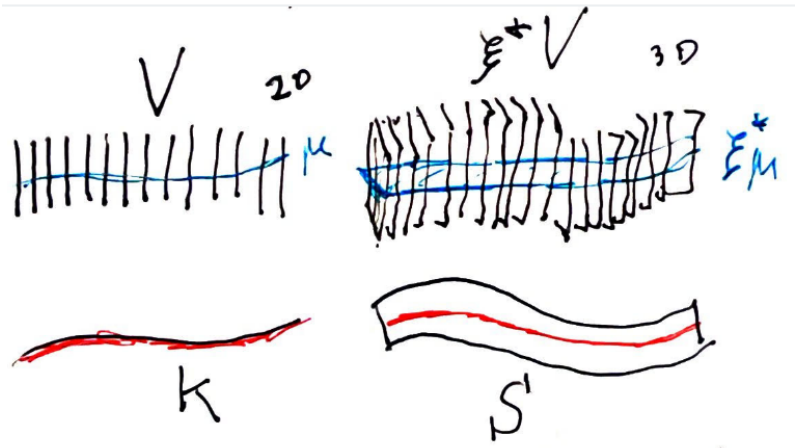


Figure 19: The pullback of the visual bundle $\xi^* V$ is the replication of a μ over all points s that map back to a single k. Because the μ is the same, we can constract a $\hat{Q}$ on μ over k that will fabricate the Q for the equivalent set of s associated to that k

When Q maps sections $Q : \Gamma(\xi^* V) \to \Gamma(H)$, if we restrict Q input to $\xi^* \mu$, then $Q(\xi^* \mu) = \rho$ depends on s. This means that $\rho(s) \coloneqq Q(\xi^* \mu)(s)$, but since $\xi^* \mu$ is constant on $\xi^{-1}(k)$, we can define a function $\hat{Q}(\mu(k))(s) \coloneqq Q((\xi^* \mu)(s))$ where $\xi^{-1}(k) = k$.

In fact, $\hat{Q}$ is a map from visual to graphic $\hat{Q} : \Gamma(V) \to \Gamma(H)$ locally over k such that $\hat{Q} : \Gamma(V_k) \to \Gamma(H \mid_{\xi^{-1}(k)})$. This allows us to construct a $\hat{Q}$ that only depends on K, such that for each $\mu(k)$ there is part of $\rho \mid_{\xi^{-1}(k)}$. The advantage of $\hat{Q}$ is that $S$is part of the rendering stage of the pipeline and therefore hidden from the Matplotlib artists and therefore there is no direct way to construct a true Q.

3.3.6 Composition of Artists:

In this paper we define a simple addition operator that is the disjoint union of fiber bundles E. For example, in figure 20 the scatter plot E_1 and the line plot E_2 have different Kthat are mapped to seperate S. To fully display both graphics, the composite graphic $A_1 + A_2$ needs to include all records on both K_1 and K_2, which are the sections on the disjoint union $K_1 \sqcup K_2$. This in turn yields disjoint graphics $S_1 \sqcup S_2$ rendered to the same image. Constraints can be placed on the disjoint union such as that the fiber components need to have the same νposition encodings or that the position μneed to be in a specified range. There are situations where $K_2 \hookrightarrow K_1$ that underpin more complex, especially interactive, visualizations; these cases require defining a more complex addition operator that is out of scope for this work.

3.3.7 Equivalance class of artists A'

As formulated above, every artist function A has fixed ν and Q which generates a distinct graphic ρ. It is impractical to implement an artist for every single graphic; instead we implement the equivalence class of artists $\{A \in A' : A_1 \equiv A_2\}$. Equivalent artists have the same fiber bundle V and same assembly function Q but act on different sections μ.

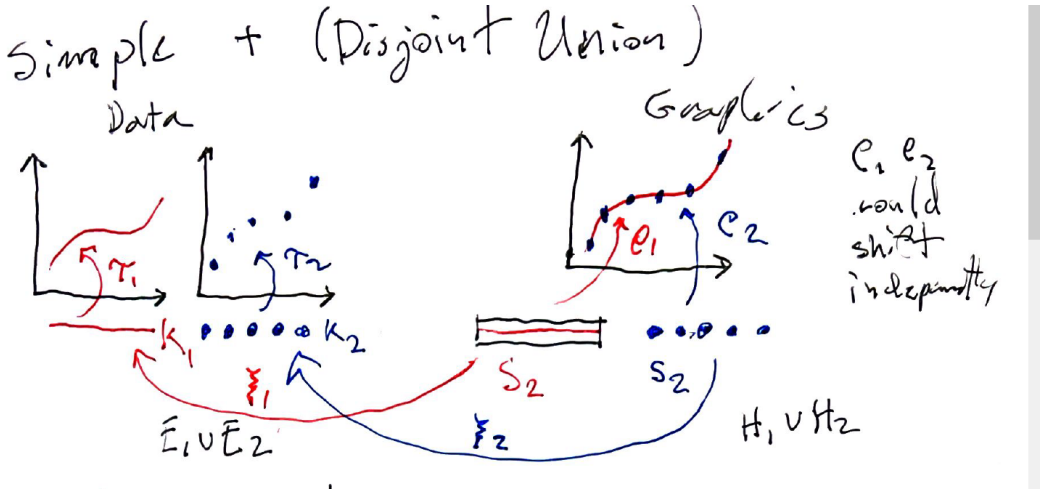


Figure 20: τ_1 and τ_2 are distinct datasets passed through artists A_1 and A_2 to generate graphics ρ_1 and ρ_2. These graphics happen to be rendered to the same image, but otherwise have no intrinsinc link.

To further simplify implementation, we identify a minimal P associated with each A' that defines what visual characteristics of the graphic must originate in the data such that the graphic is identifiable as a given chart type.

Figure 21: Each of these graphics is generated by a different artist Awhich is the equivalance class of scatter plots A'this is gonna be a whole bunch of scatter plots

For example, a scatter plot of red circles is the output of one artist, a scatter plot of green squares the output of another, as are the rest of the graphs in figure **??**. These two artists are equivalent since their only difference is in the literal visual encodings (color, shape). Shape and color could also be defined in Q but the position must come from the fiber $P = (xpos, ypos)$ since fundementally a scatter plot is the plotting of one position against another[24]. We also use this criteria to identify derivative types, for example the bubble chart[72] is a type of scatter where by definition the glyph size is mapped from the data.

3.4 Making the fiber bundle computable

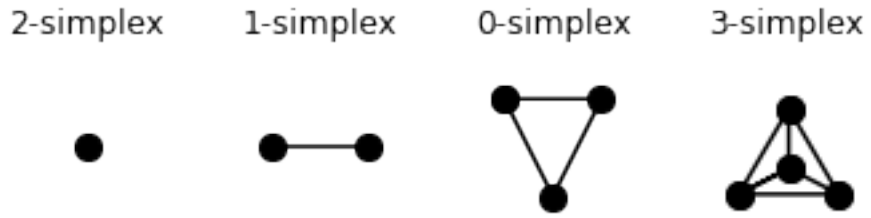


Figure 22: Simplices can encode the connectivity of the data, from fully disconnected (0 simplex) records to all records are connected to at least 3 others

One way of expressing the connectivity of records in a dataset is to implement K as a simplacial complex, which is a set of simplicies such as those shown in figure 22. The advantage of triangulization is that it is general enough to work for more complex topology based visualization methods [31] while also providing a consistent interface of vertices, edges, and faces for ξ to map into. When triangulated, the simplices encode the continuity in the data

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

Table 2

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

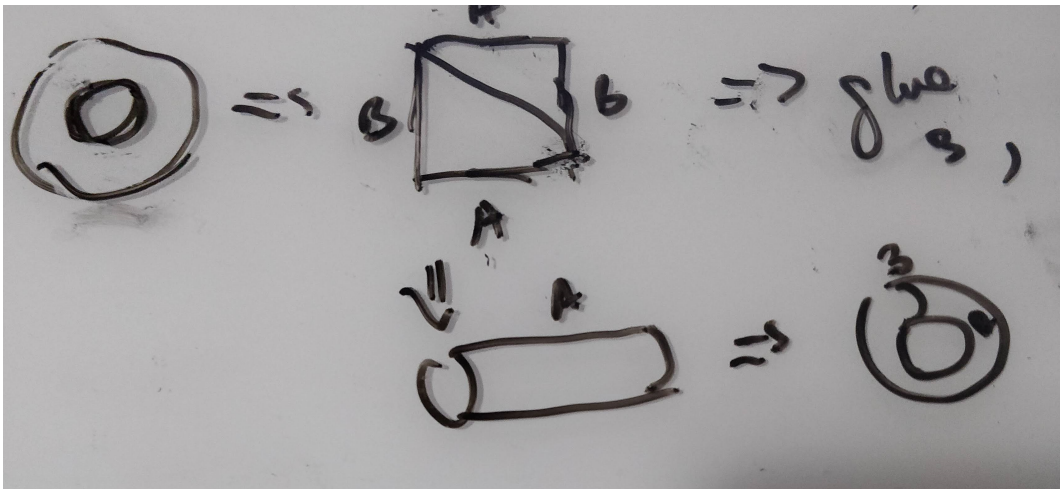


Figure 23: The torus $E$is unraveled into a simplacial complex of 2 faces K. Transition functions are defined on the edges of Ksuch that surface can be glued back into the torus. add cross sections a and b to ring and color same as edges in complex

One way of encoding the torus in figure 23 while retaining the continuity of both cross sections a, b is to unravel it into a simplacial complex of two triangles with labeled edges. Transition functions δ are defined on the edges such that a can be glued to a' and b to b' to reconstruct the torus. This simplacial complex is then used as the base space encoding the continuity of data that lies in the torus. A constraint on the transition functions is that the monoid actions on the fibers on the edges of E are commutative $M * F \mapsto \delta(MF) = M * \delta(F)$

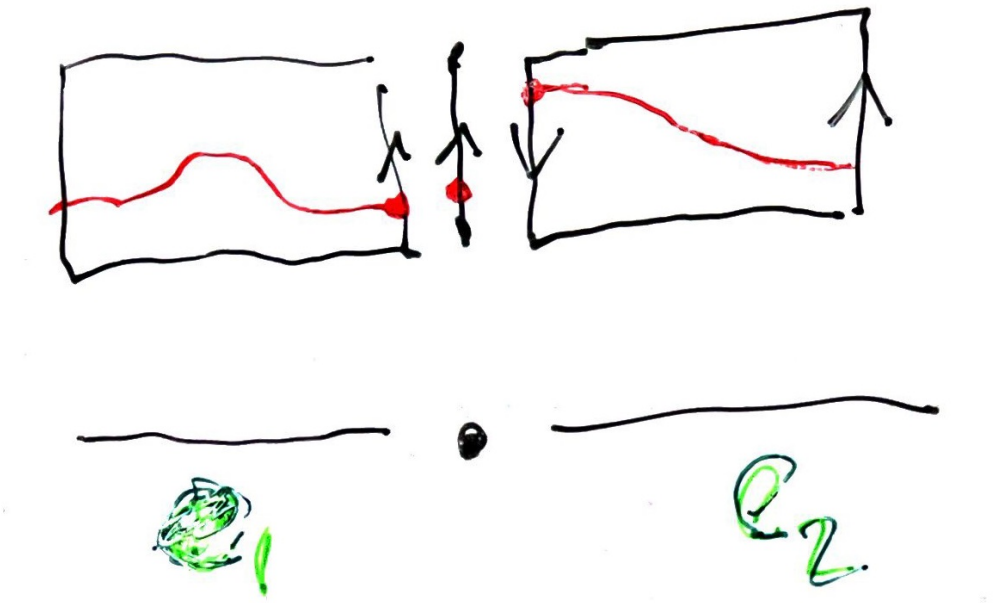


Figure 24: Many non-trivial spaces can be made locally trivial by dividing Einto locally trivial subspaces and defining transition functions between the edges on $K$for how to glue the two subspaces such that the $\tau$are continuous.

Another advantages of triangulization is that it provides a way to encode non-trivial structures such as the mobius strip[45]. As shown in figure 24, one way of making the mobius strip trivial is to seperate it into two spaces E_1 and E_2 and then define transition functions that specify that the edges of E_1 need to be reversed to line up with E_2 such that the sections along the edges meet. As with the torus, the transition functions must preserve monoid commutativity.

4 Prototype Implementation: Matplottoy

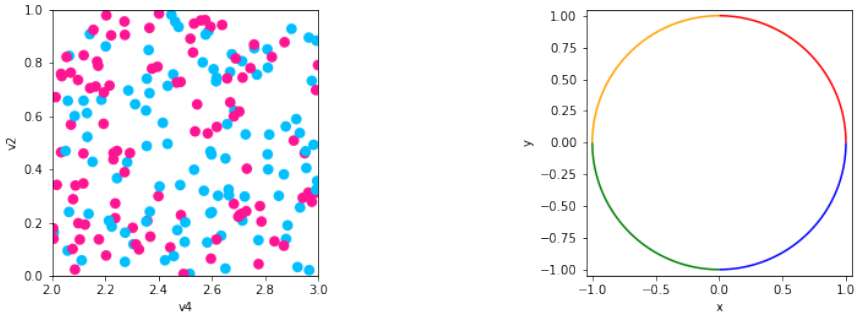


Figure 25: Scatter plot and line plot implemented using prototype artists and data models, building on Matplotlib rendering.

To prototype our model, we implemented the artist classes for the scatter and line plots shown in figure 25 because they differ in every attribute: different visual channels ν that composite to different marks Q with different continuities $\xi\dot{\text{W}}$e make use of the Matplotlib figure and axes artists [33, 34] so that we can initially focus on the data to graphic transformations.

To generate the images in figure 25, we instantiate `fig, ax` artists that will contain the new `Point, Line` primitive objects we implemented based on our topology model.

```
fig, ax = plt.subplots()
artist = Point(data, transforms)
ax.add_artist(artist)
```

```
fig, ax = plt.subplots()
artist = Line(data, transforms)
ax.add_artist(artist)
```

We then add the A'=`Point` and A'=`Line` artists that construct the scatter and line graphics. The arguments to the artist are the data E=`data` that is to be plotted and the aesthetic configuration ν=`transforms`. We implement the artists as equivalence classes A' because it would be impractical to implement a new artist for every aesthetic setting, such as one artist for red lines and another for green.

4.1 Artist Class A'

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

```
class ArtistClass(matplotlib.artist.Artist):
    def __init__(self, data, transforms, *args, **kwargs):
        # properties that are specific to the graphic but not the channels
        self.data = data
        self.transforms = transforms
        super().__init__(*args, **kwargs)

    def assemble(self, visual):
        # set the properties of the graphic

    def draw(self, renderer, *args, **kwargs):
        # returns K, indexed on fiber then key
        view = self.data.view()
        # visual channel encoding applied fiberwise
        visual = {p: encoder(view.get(f, None)) for
                     p, (f, encoder) in self.transforms.items()}
        self.assemble(visual)
        # pass configurations off to the renderer
        super().draw(renderer, *args, **kwargs)
```

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

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

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

```
class Point(mcollections.Collection):
    def __init__(self, data, transforms, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.data = data
        self.transforms = transforms

    def assemble(self, visual):
        # construct geometries of the circle marks in visual coordinates
        self._paths = [mpath.Path.circle(center=(x,y), radius=s)
                       for (x, y, s) in zip(visual['x'],visual['y'], visual['s'])]
        # set attributes of marks, these are vectorized
        # circles and facecolors are lists of the same size
        self.set_facecolors(visual['facecolors'])

    def draw(self, renderer, *args, **kwargs):
        # query data for a vertex table K
        view = self.data.view()
        visual = {p: encoder(view.get(f, None)) for
                     p, (f, encoder) in self.transforms.items()}
        self.assemble(visual)
        # call the renderer that will draw based on properties
        super().draw(renderer, *args, **kwargs)
```

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

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

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

    def draw(self, renderer, *args, **kwargs):
        # query data source for edge table
        view = self.data.view()
        visual = {p: encoder(view.get(f, None)) for
                     p, (f, encoder) in self.transforms.items()}
        self.assemble(visual)
        super().draw(renderer, *args, **kwargs)
```

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

4.2 Encoders ν

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

```
cmap =  color.Categorical({'true':'deeppink', 'false':'deepskyblue'})
transforms = {'y': ('v1', lambda x: x),
              'x': ('v3', lambda x: x),
             'facecolors': ('v2', cmap),
             's':(None ,lambda _: itertools.repeat(.02))}
```

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

```
class Categorical:
    def __init__(self, mapping):
        # check that the conversion is to valid colors
        assert(mcolors.is_color_like(color) for color in mapping.values())
        self._mapping = mapping

    def __call__(self, value):
        # convert value to a color
        return [mcolors.to_rgba(self._mapping[v]) for v in values]
```

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

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

```
def test_nominal(values, encoder):
    m1 = list(zip(values, encoder(values)))
    random.shuffle(values)
    m2 = list(zip(values, encoder(values)))
    assert sorted(m1) == sorted(m2)
```

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

4.3 Data E

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

1. specify the fiber components F and connectivity K

2. have a that returns an atomic object that encapsulates τ

3. the view object must have that returns a fiber component

To support specifying the fiber bundle, we define an optional `FiberBundle` class

```
class FiberBundle:
    def __init__(self, base, fiber):
        """
        base:  {'tables': ['vertex', 'edge', 'face']}
        fiber: {'component name': {'type':, 'monoid':,  'range':}}
        """
        self.base = base
        self.fiber = fiber
```

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

To generate the scatter plot in figure 25, we fully specify a dataset with random keys and values in a section chosen at random form the corresponding fiber component. The fiberbundle `FB` is a class level attribute since all instances of `code`VertexSimplex come from the same fiberbundle.

```
class VertexSimplex: #maybe change name to something else
    """Fiberbundle is consistent across all sections
    """
    FB = FiberBundle({'tables': ['vertex']},
            {'v1': {'type': float,'monoid':'interval','range': [0,1]},
             'v2': {'type': str, 'monoid':'nominal', 'range':['true', 'false']},
             'v3': {'type': float, 'monoid':'interval', 'range':[2,3]}})

    def __init__(self, sid = 45, size=1000, max_key=10**10):
        # create random list of keys
    def tau(self, k):
        # e1 is sampled from F1, e2 from F2, etc...
        return (k, (e1, e2, e3, e4))

    def view(self):
        table = defaultdict(list)
```

```
        for k in self.keys:
            table['index'] = k
            # on each iteration, add one (name, value) pair per component
            for (name, value) in zip(self.FB.fiber.keys(), self.tau(k)[1]):
                table[name].append(value)
        return table
```

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

```
class EdgeSimplex:
    # assign a class level FB attribute
    def __init__(self, num_edges=4, num_samples=1000):
        self.keys = range(num_edge) #edge id
        # distance along edge
        self.distances = np.linspace(0,1, num_samples)
        # half generlized representation of arcs on a circle
        self.angle_samples = np.linspace(0, 2*np.pi, len(self.keys)+1)

    @staticmethod
    def _color(edge):
        colors = ['red','orange', 'green','blue']
        return colors[edge%len(colors)]

    @staticmethod
    def _xy(edge, distances, start=0, end=2*np.pi):
        # start and end are parameterizations b/c really there is
        angles = (distances *(end-start)) + start
        return np.cos(angles), np.sin(angles)

    def tau(self, k): #will fix location on page on revision
        x, y = self._xy(k, self.distances,
                        self.angle_samples[k], self.angle_samples[k+1])
        color = self._color(k)
        return (k, (x, y, color))

    def view(self, simplex):
        table = defaultdict(list)
```

```
        for k in self.keys:
            table['index'].append(k)
            # (name, value) pair, value is [x0, ..., xn] for x, y
            for (name, value) in zip(self.FB.fiber.keys(), self.tau(k, simplex)[1]):
                table[name].append(value)
```

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

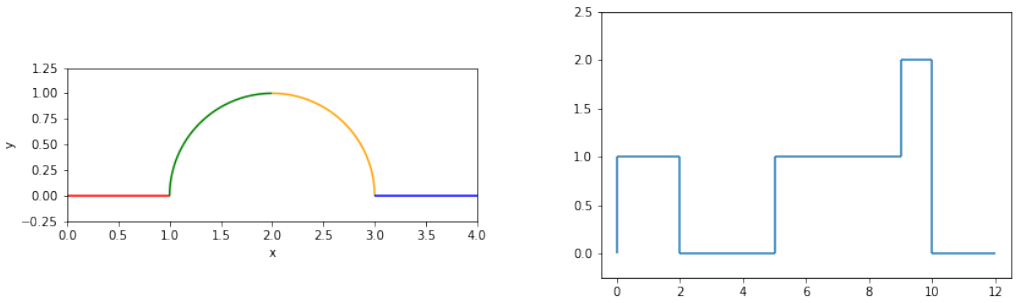


Figure 26: Continuous and discontinuous lines as defined by different data models, but generated with the same A'=`artist`

The graphics in figure 26 are made using the `Line` artist and the `Graphline` data source

```
class GraphLine:
    def __init__(self, FB, edge_table, vertex_table, num_samples=1000, connect=False):
        #s set args as attributes and generate distance
        if connect: # test connectivity if edges are continuous
            assert edge_table.keys() == self.FB.F.keys()
            assert is_continuous(vertex_table)

    def tau(self, k, simplex='edge'):
        # evaluates functions defined in edge table
        return(k, (self.edges[c][k](self.distances) for c in self.FB.F.keys()))

    def view(self, simplex='edge'):
        """walk the edge_vertex table to return the edge function
```

```
        """
        table = defaultdict(list)
        #sort since intervals lie along number line and are ordered pair neighbors
        for (i, (start, end)) in sorted(zip(self.ids, self.vertices), key=lambda v:v[1][0]):
            table['index'].append(i)
            # same as view for line, returns nested list
            for (name, value) in zip(self.FB.F.keys(), self.tau(i, simplex)[1]):
                table[name].append(value)
        return table
```

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

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

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

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

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

4.4 Case Study: Penguins

For this case study, we use the Palmer Penguins dataset[26, 32] since it is multivariate and has a varying number of penguins. We use a version of the data packaged as a pandas dataframe[51, 58] since that is a very commonly used Python labled data structure. The wrapper is very thin since here there is explicitly only one section.

```
class DataFrameSection:
    def __init__(self, dataframe):
        self._tau = dataframe.iloc
        self._view = dataframe
    def view(self):
        return self._view
```

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

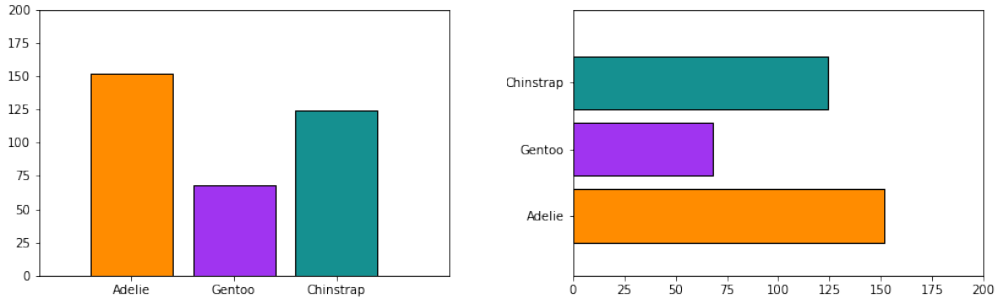


Figure 27: Frequency of Penguin types visualized as discrete bars.

The bar charts in figure 27 are generated with a `Bar` artist. The have the same required P components of (position, length). In of `Bar` an additional parameter is set, `orientation` which only applies holistically to the graphic and never to individual data parameters. Explicitly differentiate between parameters in V and ones that are only in $\hat{Q}$ is another way this model allows for cleaner separation of roles in the code.

```
class Bar(mcollections.Collection):
    def __init__(self, data, transforms, *args, **kwargs):
        # parameter of the graphic
        self.orientation = kwargs.pop('orientation', 'v')

        super().__init__(*args, **kwargs)
        self.data = data
        self.transforms = transforms

    @staticmethod
    def _make_bars(orientation, position, width, floor, length):
        if orientation in {'vertical', 'v'}:
            xval, xoff, yval, yoff = position, width, floor, length
        elif orientation in {'horizontal', 'h'}:
            xval, xoff, yval, yoff = floor, length, position, width
        return [[(x, y), (x, y+yo), (x+xo, y+yo), (x+xo, y), (x, y)]
                for (x, xo, y, yo) in zip(xval, xoff, yval, yoff)]


    def assemble(self, visual):
        #set some defaults
        visual['width'] = visual.get('width', itertools.repeat(0.8))
        visual['floor'] = visual.get('floor', itertools.repeat(0))
```

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

    def draw(self, renderer, *args, **kwargs):
        view = self.data.view()
        visual = utils.convert_transforms(view, self.transforms)
        self.assemble(visual)
        super().draw(renderer, *args, **kwargs)
        return
```

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

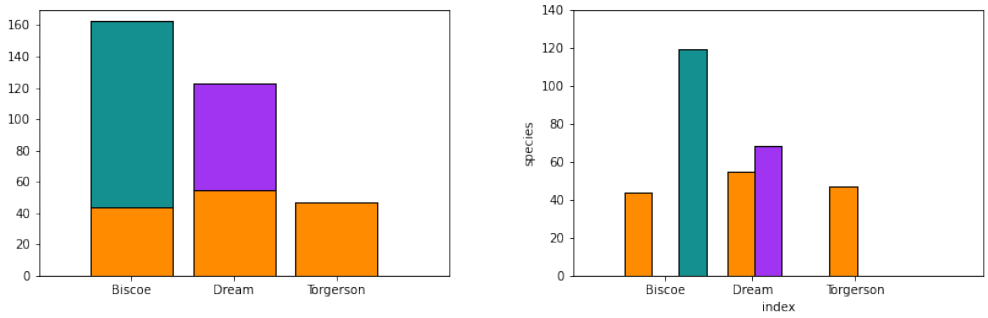


Figure 28: Penguin count disaggregated by island and species

For example, the `MultiBar` artist that makes figure 28 reuses `_make_bars` but does not reuse the assemble function because the composition of elements forces fundemental differences in glyph construction. As demonstrated in the `init`, the composite bar chart has orientation and whether it is stacked or not. While the stacked bar chart and the grouped bar chart could be seperate artists, as demonstrated they share so much overlapping code that it is far less redundant to implement them together. looking at the mess that is this code, I'm a) not convinced these should be combined b) no longer convinced this provides anything over just bar if it isn't rewritten to use bar more

```
class MultiBar(mcollections.Collection):
    def __init__(self, data, transforms, *args, **kwargs):
        #set the orientation of the graphic
        self.orientation = kwargs.pop('orientation', 'v')
        # set how the bar glyphs are put together to create the graphic
        self.stacked = kwargs.pop('stacked', False)
        # rest is same as other artist __init__s

        #this needs to be factored out but just want to finish now
        self.width = kwargs.pop('width', .8)

    def assemble(self, visual, view):
        (groups, gencoder) = self.transforms['length']
        ngroups = len(np.atleast_1d(groups))
        visual['floor'] = visual.get('floor', np.empty(len(view[groups[0]])))
        visual['facecolors'] = visual.get('facecolors', 'C0')
        # make equal width stacked columns
        if 'width' not in visual and self.stacked:
            visual['width'] = itertools.repeat(self.width)

        # make equal with groups
        if not self.stacked:
            visual['width'] = itertools.repeat(self.width/ngroups)
            offset = (np.arange(ngroups) /ngroups) * self.width
        else:
            offset = itertools.repeat(0)

        # make the bars and arrange them
        verts = []
        for group, off in zip(groups, offset):
            verts.extend(Bar._make_bars(self.orientation, visual['position'] + off,
                         visual['width'], visual['floor'], view[group]))
            if self.stacked: #add stacked bar to previous bar
                visual['floor'] += view[group]

        # convert lengths after all calculations are made and reorient if needed
        # here or in transform machinery?
        if self.orientation in {'v', 'vertical'}:
            tverts = [[(x, gencoder(y)) for (x, y) in vert]
```

```
                            for vert in verts]
        elif self.orientation in {'h', 'horizontal'}:
            tverts = [[(gencoder(x), y) for (x, y) in vert]
                            for vert in verts]
        self._paths = [mpath.Path(xy, closed=True) for xy in tverts]
        #flatted columns of colors to match list of bars
        self.set_facecolor(list(itertools.chain.from_iterable(visual['facecolors'])))
        self.set_edgecolors('k')

    def draw(self, renderer, *args, **kwargs):
        view = self.data.view()
        #exclude converting the group visual length, special cased in assemble
        visual = utils.convert_transforms(view, self.transforms, exclude=['length'])
        # pass in view because nu is not distributable so may need to apply it
        # after visual assembly
        self.assemble(visual, view)
        super().draw(renderer, *args, **kwargs)
        return
```

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

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

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

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

where `trans` is the same dictionary for both stacked and grouped version, as is the `DataFrameSection` object `table`. The only difference between the two versions is the `stacked` flag, and the only difference between figures 27 is the `orientation` argument. By decomposing the architecture into data, visual encoding, and assembly steps, we are able to build components that are more flexible and also more self contained than the existing code base.

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

5 Discussion

This work contributes a mathematical description of the transformation from data to visual representation. Combining Butler's fiber bundle model of data with Spivaks formalism of data schemas provides a way of decoupling topology from variability such that the model can support a very large variety of datasets, including discrete relational tables, multivariate high resolution spatio temporal datasets, and complex networks. Modeling the graphic as a fiber bundle provides a way to separate the target display space from the topology of the graphic. By decomposing the mapping from data to visual representation as encoding ν, assembly Q, and a mapping between data and graphic topologies $\xi$and formalizing what equivariance each stage needs to preserved, this work derives constraints that visualization library authors could embed in their code to guarantee visualizations that are equivariant transforms of the input data.

This work generalizes previous research constraining visual encodings from data components to graphic components as equivariant maps to components that are N-dimensional. Furthermore, it precisely defines the glyph as the visual element constructed from data on a simplex in a simplacial complex where the simplex is discrete or continuous. This is a restatement of for example Bertin's definition of a line that encapsulates all points on the continuous line. By modeling the data topology along with its variablity, this work also provides a generalization of topology preservation as a deformation retraction from graphic space to data space. By using a functional paradigm, we can deconstruct the graphic to the glyph associated with each data point or even to pieces of a glyph; therefore the renderer has full flexibility in how to to generate the image, while the graphic to data topology maps provide a way to keep track of which part of the image belongs to which data point.

The toy prototype built using this model validates that is usable for a general purpose visualization tool since it can be iteratively integrated into the existing architecture rather than starting from scratch/ Factoring out glyph formation into assembly functions allows for much more clarity in how the glyphs differ. This prototype demonstrates that this framework can generate the fundemental marks, point (scatter plot), line (line chart), and area (bar chart). Furthermore, the grouped and stacked bar examples demonstrate that this model supports composition of glyphs into more complex graphics. These composite examples also rely on the fiber bundles section base book keeping to keep track of which components contribute to the attributes of the glyph. Implementing this example

using a Pandas dataframe demonstrates the ease of incorporating existing widely used data containers rather than requiring users to conform to one stands.

5.1 Limitations

So far this model has only been worked out for a single data set tied to a primitive mark, but it should be extensible to compositing datasets and complex glyphs. The examples and prototype have so far only been implemented for the static 2D case, but nothing in the math limits to 2D and expansion to the animated case should be possible because the model is formalized in terms of the sheaf. While this model supports equivariance of figurative glyphs generated form parameters of the data[5, 14], it currently does not have a way to evaluate the semantic accuracy of the figurative representation. This model also does not currently factor in effectiveness, but potentially effectiveness criteria could be incorporated into a scheme for assigning encoders and assembling glyphs. 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[16]. 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[40], 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 [30]. 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 continous topology and an artist that can consume data with a 2D topology to visualize the image[27, 28, 71] and also encoding a separate heatmap[39, 79] artist that consumes 1D discrete data. A second important proof of concept artist is a boxplot[77] 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[6, 10] and verifying constraints in multiple view systems [54]. 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 potential tasks for future work is implementing a data object for a non-trivial fiber bundle and exploiting the models section level formalism to build distributed data source models and concurrent artists. This could be pushed further to integrate with topological[31] and functional [57] data analysis methods. While this paper formulates visualization in terms of monoidal action homomorphisms between fiberbundles, the model lends itself to a categorical formulation[23, 44] that could be further explored.

6 Conclusion

Despite the alternative formalism to Mackinlay's APT framework, it is indebted to APT for providing a guideline for issues the model needs to address. As with APT, this works aims to develop a framework on which to build visualization tools. Unlike APT, our model is geared towards balancing the needs of the the scientific and information visualization communities; therefore we present a framework that can describe both how the components of the data are encoded and how the continuity is preserved in the graphic. We hope that this framework can distill the constraints on the data to graphic mapping such that it can be used to improve the architecture of a heavily used visualization tool and allow developers to more easily build domain specific tools.

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

References

[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.

[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.

[3] *Action in nLab.* https://ncatlab.org/nlab/show/action#actions_of_a_monoid.

[4] Professor Denis Auroux. "Math 131: Introduction to Topology". en. In: (), p. 113.

[5] F. Beck. "Software Feathers Figurative Visualization of Software Metrics". In: *2014 International Conference on Information Visualization Theory and Applications (IVAPP)*. Jan. 2014, pp. 5–16.

[6] Richard A. Becker and William S. Cleveland. "Brushing Scatterplots". In: *Technometrics* 29.2 (May 1987), pp. 127–142. ISSN: 0040-1706. DOI: 10.1080/00401706.1987.10488204.

[7] 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.

[8] M. Bostock and J. Heer. "Protovis: A Graphical Toolkit for Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1121–1128. ISSN: 1941-0506. DOI: 10.1109/TVCG.2009.174.

[9] M. Bostock, V. Ogievetsky, and J. Heer. "D^3 Data-Driven Documents". In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pp. 2301–2309. ISSN: 1941-0506. DOI: 10.1109/TVCG.2011.185.

[10] Andreas Buja et al. "Interactive Data Visualization Using Focusing and Linking". In: *Proceedings of the 2nd Conference on Visualization '91*. VIS '91. Washington, DC, USA: IEEE Computer Society Press, 1991, pp. 156–163. ISBN: 0-8186-2245-8.

[11] D. M. Butler and M. H. Pendley. "A Visualization Model Based on the Mathematics of Fiber Bundles". en. In: *Computers in Physics* 3.5 (1989), p. 45. ISSN: 08941866. DOI: 10.1063/1.168345.

[12] David M. Butler and Steve Bryson. "Vector-Bundle Classes Form Powerful Tool for Scientific Visualization". en. In: *Computers in Physics* 6.6 (1992), p. 576. ISSN: 08941866. DOI: 10.1063/1.4823118.

[13] L. Byrne, D. Angus, and J. Wiles. "Acquired Codes of Meaning in Data Visualization and Infographics: Beyond Perceptual Primitives". In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (Jan. 2016), pp. 509–518. ISSN: 1077-2626. DOI: 10.1109/TVCG.2015.2467321.

[14] Lydia Byrne, Daniel Angus, and Janet Wiles. "Figurative Frames: A Critical Vocabulary for Images in Information Visualization". In: *Information Visualization* 18.1 (Aug. 2017), pp. 45–67. ISSN: 1473-8716. DOI: 10.1177/1473871617724212.

[15] Sheelagh Carpendale. *Visual Representation from Semiology of Graphics by J. Bertin*. en.

[16] George S. Carson. "Standards Pipeline: The OpenGL Specification". In: *SIGGRAPH Comput. Graph.* 31.2 (May 1997), pp. 17–18. ISSN: 0097-8930. DOI: 10.1145/271283.271292.

[17] John M Chambers et al. *Graphical Methods for Data Analysis*. Vol. 5. Wadsworth Belmont, CA, 1983.

[18] William S. Cleveland. "Research in Statistical Graphics". In: *Journal of the American Statistical Association* 82.398 (June 1987), p. 419. ISSN: 01621459. DOI: 10.2307/2289443.

[19] William S. Cleveland and Robert McGill. "Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods". In: *Journal of the American Statistical Association* 79.387 (Sept. 1984), pp. 531–554. ISSN: 0162-1459. DOI: 10.1080/01621459.1984.10478080.

[20] "Connected Space". en. In: *Wikipedia* (Dec. 2020).

[21] *Data Representation in Mayavi — Mayavi 4.7.2 Documentation*. https://docs.enthought.com/mayavi/mayavi/da

[22] 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.

[23] 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.

[24] 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.

[25] Berk Geveci et al. "VTK". In: *The Architecture of Open Source Applications* 1 (2012), pp. 387–402.

[26] 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.

[27] 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.

[28] Charles D Hansen and Chris R Johnson. *Visualization Handbook*. Elsevier, 2011.

[29] 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.

[30] 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.

[31] 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.

[32] Allison Marie Horst, Alison Presmanes Hill, and Kristen B Gorman. *Palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. Manual. 2020. DOI: 10.5281/zenodo.3960218.

[33] 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.

[34] John Hunter and Michael Droettboom. *The Architecture of Open Source Applications (Volume 2): Matplotlib*. https://www.aosabook.org/en/matplotlib.html.

[35] "Jet Bundle". en. In: *Wikipedia* (Dec. 2020).

[36] John Krygier and Denis Wood. *Making Maps: A Visual Guide to Map Design for GIS*. English. 1 edition. New York: The Guilford Press, Aug. 2005. ISBN: 978-1-59385-200-9.

[37] W A Lea. "A Formalization of Measurement Scale Forms". en. In: (), p. 44.

[38] *Locally Trivial Fibre Bundle - Encyclopedia of Mathematics*. https://encyclopediaofmath.org/wiki/Locally_trivial

[39] Toussaint Loua. *Atlas Statistique de La Population de Paris*. J. Dejey & cie, 1873.

[40] Kenneth C. Louden. *Programming Languages : Principles and Practice*. English. Pacific Grove, Calif: Brooks/Cole, 2010. ISBN: 978-0-534-95341-6 0-534-95341-7.

[41] Jock Mackinlay. “Automating the Design of Graphical Presentations of Relational Information”. In: *ACM Transactions on Graphics* 5.2 (Apr. 1986), pp. 110–141. ISSN: 0730-0301. DOI: 10.1145/22949.22950.

[42] JOCK D. MACKINLAY. “AUTOMATIC DESIGN OF GRAPHICAL PRESENTATIONS (DATABASE, USER INTERFACE, ARTIFICIAL INTELLIGENCE, INFORMATION TECHNOLOGY)”. English. PhD Thesis. 1987.

[43] Connie Malamed. *Information Display Tips*. https://understandinggraphics.com/visualizations/information-display-tips/. Blog. Jan. 2010.

[44] Bartosz Milewski. “Category Theory for Programmers”. en. In: (), p. 498.

[45] *Möbius Strip in nLab*. https://ncatlab.org/nlab/show/M%C3%B6bius+strip.

[46] “Monoid”. en. In: *Wikipedia* (Jan. 2021).

[47] T Munzner. “Marks and Channels”. In: *Visualization Analysis and Design*, pp. 94–114.

[48] Tamara Munzner. “Ch 2: Data Abstraction”. In: *CPSC547: Information Visualization, Fall 2015-2016* ().

[49] Tamara Munzner. *Visualization Analysis and Design*. AK Peters Visualization Series. CRC press, Oct. 2014. ISBN: 978-1-4665-0891-0.

[50] Jana Musilová and Stanislav Hronek. “The Calculus of Variations on Jet Bundles as a Universal Approach for a Variational Formulation of Fundamental Physical Theories”. In: *Communications in Mathematics* 24.2 (Dec. 2016), pp. 173–193. ISSN: 2336-1298. DOI: 10.1515/cm-2016-0012.

[51] Muhammad Chenariyan Nakhaee. *Mcnakhaee/Palmerpenguins*. Jan. 2021.

[52] 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: 0-201-62695-0.

[53] Z. Pousman, J. Stasko, and M. Mateas. “Casual Information Visualization: Depictions of Data in Everyday Life”. In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (Nov. 2007), pp. 1145–1152. ISSN: 1941-0506. DOI: 10.1109/TVCG.2007.70541.

[54] Z. Qu and J. Hullman. “Keeping Multiple Views Consistent: Constraints, Validations, and Exceptions in Visualization Authoring”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (Jan. 2018), pp. 468–477. ISSN: 1941-0506. DOI: 10.1109/TVCG.2017.2744198.

[55] “Quotient Space (Topology)”. en. In: *Wikipedia* (Nov. 2020).

[56] P. Ramachandran and G. Varoquaux. “Mayavi: 3D Visualization of Scientific Data”. In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 40–51. ISSN: 1558-366X. DOI: 10.1109/MCSE.2011.35.

[57] James O Ramsay. *Functional Data Analysis*. Wiley Online Library, 2006.

[58] Jeff Reback et al. *Pandas-Dev/Pandas: Pandas 1.0.3*. Zenodo. Mar. 2020. DOI: 10.5281/zenodo.3715232.

[59] “Retraction (Topology)”. en. In: *Wikipedia* (July 2020).

[60] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. "Declarative Interaction Design for Data Visualization". en. In: *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*. Honolulu Hawaii USA: ACM, Oct. 2014, pp. 669–678. ISBN: 978-1-4503-3069-5. DOI: 10.1145/2642918.2647360.

[61] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. "NIH Image to ImageJ: 25 Years of Image Analysis". In: *Nature Methods* 9.7 (July 2012), pp. 671–675. ISSN: 1548-7105. DOI: 10.1038/nmeth.2089.

[62] "Semigroup Action". en. In: *Wikipedia* (Jan. 2021).

[63] E.H. Spanier. *Algebraic Topology*. McGraw-Hill Series in Higher Mathematics. Springer, 1989. ISBN: 978-0-387-94426-5.

[64] David I Spivak. *Databases Are Categories*. en. Slides. June 2010.

[65] David I Spivak. "SIMPLICIAL DATABASES". en. In: (), p. 35.

[66] "Stalk (Sheaf)". en. In: *Wikipedia* (Oct. 2019).

[67] S. S. Stevens. "On the Theory of Scales of Measurement". In: *Science* 103.2684 (1946), pp. 677–680. ISSN: 00368075, 10959203.

[68] Software Studies. *Culturevis/Imageplot*. Jan. 2021.

[69] *[The Georgia Negro] City and Rural Population. 1890*. eng. https://www.loc.gov/item/2013650430/. Image. 1900.

[70] *[The Georgia Negro] Negro Property in Two Cities of Georgia*. eng. https://www.loc.gov/item/2013650443/. Image. 1900.

[71] M. Tory and T. Moller. "Rethinking Visualization: A High-Level Taxonomy". In: *IEEE Symposium on Information Visualization*. Oct. 2004, pp. 151–158. DOI: 10.1109/INFVIS.2004.59.

[72] Edward R. Tufte. *The Visual Display of Quantitative Information*. English. Cheshire, Conn.: Graphics Press, 2001. ISBN: 0-9613921-4-2 978-0-9613921-4-7 978-1-930824-13-3 1-930824-13-0.

[73] Jacob VanderPlas et al. "Altair: Interactive Statistical Visualizations for Python". en. In: *Journal of Open Source Software* 3.32 (Dec. 2018), p. 1057. ISSN: 2475-9066. DOI: 10.21105/joss.01057.

[74] C. Ware. *Information Visualization: Perception for Design*. Interactive Technologies. Elsevier Science, 2019. ISBN: 978-0-12-812876-3.

[75] Eric W. Weisstein. *Similarity Transformation*. en. https://mathworld.wolfram.com/SimilarityTransformation.htm Text.

[76] Hadley Wickham. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN: 978-3-319-24277-4.

[77] Hadley Wickham and Lisa Stryjewski. "40 Years of Boxplots". In: *The American Statistician* (2011).

[78] Leland Wilkinson. *The Grammar of Graphics*. en. 2nd ed. Statistics and Computing. New York: Springer-Verlag New York, Inc., 2005. ISBN: 978-0-387-24544-7.

[79] Leland Wilkinson and Michael Friendly. "The History of the Cluster Heat Map". In: *The American Statistician* 63.2 (May 2009), pp. 179–184. ISSN: 0003-1305. DOI: 10.1198/tas.2009.0033.

[80] *Writing Plugins*. en. https://imagej.net/Writing_plugins.

[81] Brent A Yorgey. "Monoids: Theme and Variations (Functional Pearl)". en. In: (), p. 12.