Topological Equivariant Artist Model

Hannah Aizenman, Thomas Caswell, Michael Grossberg

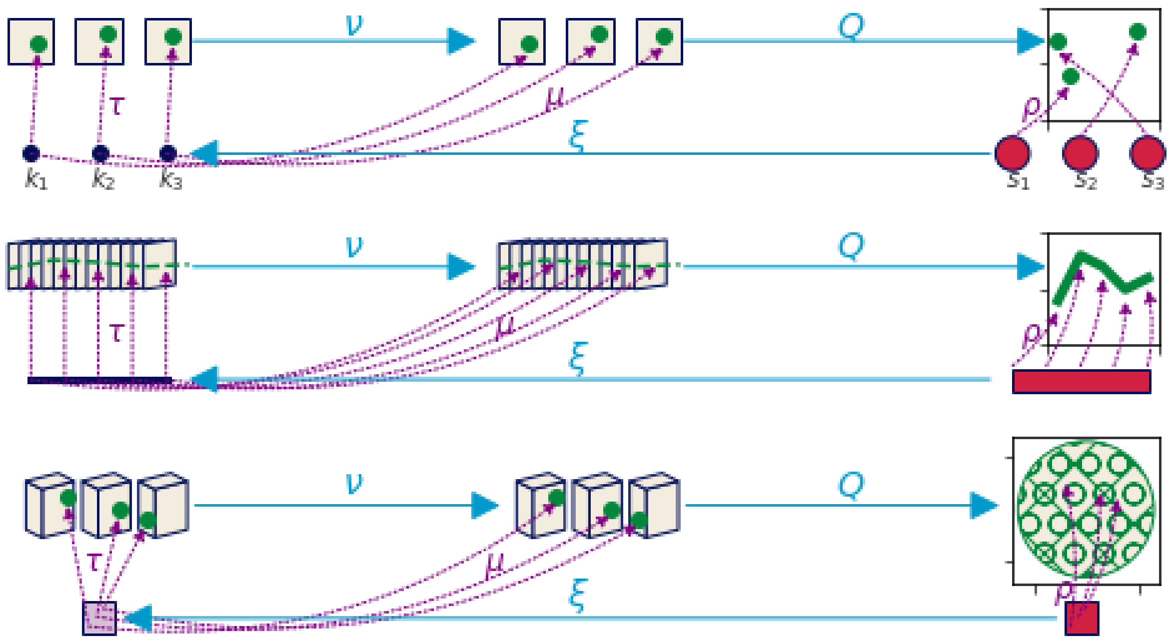


Fig. 1: Visualizations consist of topologically equivariant maps. There is a set of monoid action equivariant maps from data components to visual components ν that are then reduced via Q into a single graphic, and there is a deformation retraction ξ from graphic continuity to data continuity.

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

Index Terms—Taxonomy, Models, Frameworks, Theory

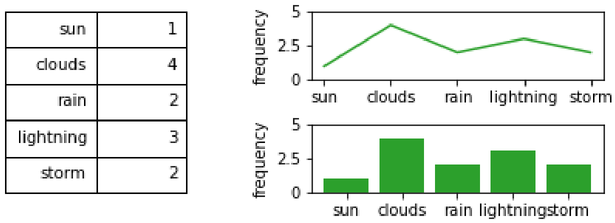
1 Introduction

General purpose visualization libraries are often tasked with supporting many domains, leading the libraries to grow organically in ways that can lead to incoherent interfaces and brittle components. To adapt to modern library development needs, this paper presents a functional model of the how components provided by lower level libraries can be guaranteed to be structure preserving maps from data to graphic. The target for this model are the libraries developers use to build automated and

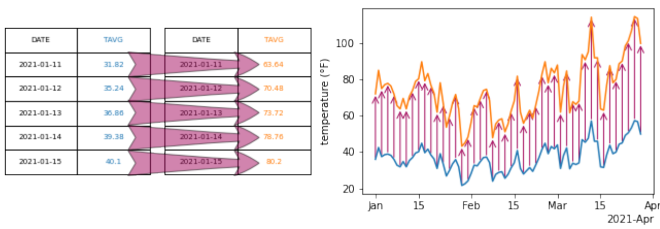
- *Hannah Aizenman and Michael Grossberg are with the department of Computer Science, City College of New York. E-mail: haizenman@ccny.cuny.edu, mgrossberg@ccny.cuny.edu.*
- *Thomas Caswell is with National Synchrotron Light Source II, Brookhaven National Lab E-mail: tcaswell@bnl.gov.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

user facing tools rather than those tools. The model presented here is agnostic towards what makes for a good visualization, so long as the specified constraints for building those visualizations are satisfied.



(a)



(b)

Fig. 2: In Fig. 2a, the line plot does not preserve *continuity* because it implies that the discrete records are connected to each other, while the bar plot is *continuity* preserving because it visually represents the records as independent data points. In Fig. 2b, the data in blue is scaled by a factor of two, yielding the data in orange. To preserve *equivariance*, the blue line plot representation of the unscaled data is also scaled by a factor of two, yielding the orange line plot that is equivalent to the scaled data.

The structure that components of a visualization library must preserve is *continuity* and *equivariance*, as illustrated in Fig. 2. *Continuity* is the way in which the records in the dataset are connected to each other and *equivariance* means that if any action is applied to the data or the visual, for example a rotation, permutation, translation, or rescaling, an equivalent action is applied on the other side. In Fig. 2a, the line plot does not preserve continuity because the line connecting the discrete categories implies that the frequency of weather events is sampled from a continuous interval and the categories are points on that interval. In Fig. 2b, the data and visual representation are scaled by equivalent factors of two, resulting in the change illustrated in the shift from blue to orange data and lines.

To specify the constraints components must satisfy to be equivariant continuity preserving mappings from data to visual representation, we developed a functional model, which we call the Topological Equivariant Artist Model (TEAM). Motivated by the challenge of re-architecturing the Python visualization library Matplotlib, a functional architecture derived from our model would gain a generalized data abstraction and well specified reusable components with minimal side effects [44], in turn yielding improvements in maintainability, extendability, scaling, and support for concurrency. The contribution of this work is

1. formalization of the topology preserving relationship between data and graphic via continuous maps Sect. 3.2.1
2. formalization of property preservation from data component to visual representation as monoid action equivariant maps Sect. 3.3
3. functional oriented visualization architecture built on the mathematical model to demonstrate the utility of the model Sect. 3.3.2
4. prototype of the architecture built on Matplotlib's infrastructure to demonstrate the feasibility of the model. Sect. 4

2 Related Work

The notion that visualization is equivariant continuity preserving maps from data to visual representation is neither a new formalism nor a new implementation goal; this work bridges the formalism and implementation in a functional manner with a topological approach at a building blocks library [71] level. When introducing the retinal variables, Bertin informally specifies that continuity is preserved in the mark and defines equivariance constraints in terms of data and visual variables being selective, associative, ordered, or quantitative [14]. In the *A Presentation Tool*(APT) model, Mackinlay embeds the continuity constraint in the choice of visualization type and generalizes the equivariance constraint to preserving a binary operator from one domain to another. The algebraic model of visualization [41], proposed by Kindlmann and Scheidegger, restricts equivariance to invertible transformations. Our model defines equivariance in terms of monoid actions, which are more restrictive than all binary operations but more general than invertible transformations. The Kindlmann and Scheidegger algebraic model also describes visualization as commutative mappings from data space to representation space to graphic space. Our model has the same notion of commutative mappings across analogous spaces, but uses topology to explicitly include continuity.

We emphasize continuity because visualization tools tend to either be architectured around a core data structure [38] or explicitly implement the visual algorithm in terms of the data continuity the algorithm expects [64]. For example, the relational database is core to tools influenced by APT, such as Tableau [35,46,62] and the Grammar of Graphics [69] inspired ggplot [67], Vega [54] and Altair [66]. Images underpin scientific visualization tools such as Napari [57] and ImageJ [55] and the digital humanities oriented ImagePlot [63] macro; the need to visualize and manipulate graphs has spawned tools like Gephi [11], Graphviz [29], and Networkx [34]. Neither the table nor image nor graph model on its own supports all the data types a typical general purpose visualization library needs to support; instead libraries such as Matplotlib [40] and Vtk [**?**,37] and D3 [16] explicitly carry around different data representations for all the different types of visualizations they support. Where libraries with a single core data structure have very consistent APIs, VTK, D3 and Matplotlib APIs can seem incoherent as every visualization has a different notion of how the data is structured. Our model facilitates unifying these APIs via an abstraction of data general enough to encompass most continuities, the clear separation of data, visual, and graphic transformations into functional components to mitigate side effects, and by specifying the constraints these components must satisfy to map data to visualizations in a structure preserving manner.

3 Topological Equivariant Artist Model

We introduce the notion of an artist $\mathscr{A}$ as an equivariant map

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

from data $\mathscr{E}$ (Sect. 3.1) to graphic $\mathscr{H}$ (Sect. 3.2) fiber bundles. We decompose the artist into an indexing map from graphic to data (Sect. 3.2.1), a map from data components to visual components (Sect. 3.3.1), and map from visual components to graphic (Sect. 3.3.2).

3.1 Data Bundle

Building on Butler's proposal of using fiber bundles as a common data representation structure for visualization data [19,20], a fiber bundle is a tuple (E, K, π, F) defined by the projection map π

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

that binds the components of the data in F to the continuity represented in K. By definition fiber bundles are locally trivial [2,58], meaning that over a localized neighborhood U the total space is the cartesian product $K \times F$.

3.1.1 Fiber Space: Variables

To formalize the structure of the data components, we use notation introduced by Spivak [59,60] that binds the components of the fiber to variable names. Spivak constructs a set $\mathbb{U}$ that is the disjoint union of all possible objects of types $\{T_0, \dots, T_m\} \in \mathbf{DT}$, where **DT** are the data types of the variables in the dataset. He then defines the single variable set $\mathbb{U}_\sigma$, which is $\mathbb{U}$ restricted to objects of type T bound to variable name c. The $\mathbb{U}_\sigma$ lookup is by name to specify that every component is distinct, since multiple components can have the same type T. Given σ, the fiber for a one variable dataset is

$$F = \mathbb{U}_{\sigma(c)} = \mathbb{U}_T \tag{3}$$

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:

$$F = \mathbb{U}_{\sigma(c_1)} \times \dots \mathbb{U}_{\sigma(c_i)} \dots \times \mathbb{U}_{\sigma(c_n)} \tag{4}$$

which is equivalent to

$$F = F_0 \times \dots \times F_i \times \dots \times F_n \tag{5}$$

which allows us to decouple F into components F_i. Each component of F is a dimension of the topological fiber space and is specified by a tuple of the form (c, T, $\mathbb{U}_{\sigma(c)}$).

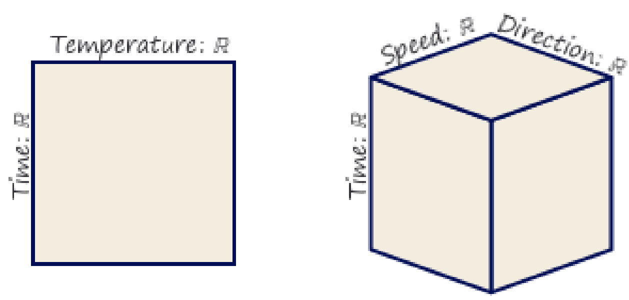


Fig. 3: These two datasets have the same base space K. The plane is a representation of the fiber $F = \mathbb{R} \times \mathbb{R}$ for the variables (time, temperature) from Fig. 2b, while the cube is the fiber $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ associated with (time, wind=(speed, direction))

In Fig. 3 the plane fiber has components (*time*, `datetime`, $\mathbb{R}$) and (*temperature*, `float`, $\mathbb{R}$), while the cube fiber has components (*time*, `datetime`, $\mathbb{R}$) and (*wind*, `wind`, $\mathbb{R} \times \mathbb{R}$) which encodes (*speed*, *direction*).

3.1.2 Equivariant Variable Properties: Monoid Actions

While structure on a set of values is often described algebraically as operations or through the actions of a group, for example Steven's measurement scales [42,61], we generalize to monoids to support partial orderings. A partial ordering allows for multiple measurement values to have the same rank [30], which is useful for visualizing many types of multi indicator systems [17].

A monoid [7] M is a set with an associative binary operator $* : M \times M \to M$. A monoid has an identity element $e \in M$ such that $e * a = a * e = a$ for all $a \in M$. As defined on a component of F, a left monoid action [8,51] of M_i is a set F_i with an action $\bullet : M \times F_i \to F_i$ with the properties of associativity and identity. As with the fiber F the total monoid space M is the cartesian product

$$M = M_0 \times \dots \times M_i \times \dots \times \dots M_n \tag{6}$$

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

Defining the monoid actions on the components serves as the basis for identifying the invariance [41] that must be preserved in the visual representation of the component. Monoids are commonly found in functional programming because a property of monoids is that the components can be composed into complex transformation pipelines [72].

3.1.3 Base Space: Continuity

The base space K acts as an indexing space, as emphasized by Butler [19,20], to express how the records in E are connected to each other. This is similar the notion of structural *keys* with associated *values* proposed by Munzner [48], but our model treats keys as a pure reference to topology. Decoupling the keys from their semantics allows the metadata to be altered facilitates encoding of data where the independent variable may not be clear; for example the growth of a plant is dependent on both time and sunlight, and changing the changing the coordinate system or time resolution should have no effect on how the records are connected to each other.

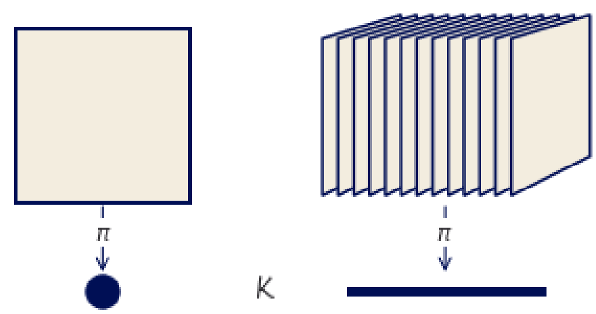


Fig. 4: These two datasets have the same fiber space $F = \mathbb{R} \times \mathbb{R}$. For example, the left dataset is a set of discrete (*time*, *temperature*) records while the right is a 1D continuous function over the interval $[0,1]$ with return values of the form (*time*, *temperature*).

As illustrated in Fig. 4, K can have any number of dimensions, can be continuous or discrete, and is somewhat independent of the dimensions of the fiber. Every $k \in K$ has a corresponding fiber F_k because K is the quotient space [5,10] of E. As with Equation 5 and Equation 6, we can decompose the total space into component bundles

$$\pi : E_1 \oplus \dots \oplus E_i \oplus \dots \oplus E_n \to K \tag{7}$$

such that M_i acts on component bundle E_i. The K remains the same because the connectivity of records does not change just because there are fewer components in each record. By encoding this continuity in the model as K the data model now explicitly carries information about its structure such that the implicit assumptions of the visualization algorithms are now explicit. The explicit topology is a concise way of distinguishing visualizations that appear identical, for example heatmaps and images.

3.1.4 Section: Values

While the projection function $\pi : E \to K$ ties together the base space K with the fiber F, a section $\tau : K \to E$ encodes a dataset. A section function takes as input location $k \in K$ and returns a

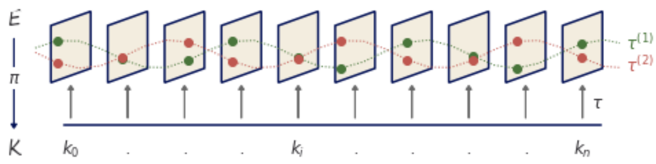


Fig. 5: Each section in the fiber bundle is a unique continuous map from base space to fiber encoding a set of records in the dataset. For example, the two sections $\tau^{(1)}$ and $\tau^{(2)}$ each encode a timeseries from a different weather station.

record $r \in E$. For any fiber bundle, there exists a map

$$\begin{tikzcd} F \arrow[r, hook] & E \arrow[d, "\pi"'] \\ & K \arrow[u, bend right, "\tau"'] \end{tikzcd} \tag{8}$$

such that $\pi(\tau(k)) = k$. The set of all global sections is denoted as $\Gamma(E)$. As illustrated in Fig. 5, the section is a continuous mapping from a location $k \in K$ on the base space to a record $r \in F$ in the fiber. Assuming a trivial fiber bundle $E = K \times F$, a section

$$\tau(k) = (k, (g_{F_0}(k), \dots, g_{F_n}(k))) \tag{9}$$

returns a record for each k. The index function $g : K \to F$ into each fiber component returns a value in the component. This formulation of the section also holds on locally trivial sections of a non-trivial fiber bundle. As with Equation 5 and Equation 7, τ can be decomposed into components

$$\tau = (\tau_0, \dots, \tau_i, \dots, \tau_n) \tag{10}$$

where each section τ_i maps into a record on a component $F_i \in F$. This allows for accessing the data component wise in addition to accessing the data in terms of its location over K.

3.1.5 Sheafs

A sheaf is a mathematical structure for defining collections of objects [31,32,65] on mathematical spaces. On the fiber bundle E, we can describe a sheaf as the collection of local sections $\iota^*\tau$

$$\begin{tikzcd} \iota^*E \arrow[r, hook, "\iota^*"] \arrow[d, "\pi"'] & E \arrow[d, "\pi"'] \\ U \arrow[u, bend right, "\iota^*\tau"'] \arrow[r, hook, "\iota"] & K \arrow[u, bend right, "\tau"'] \end{tikzcd} \tag{11}$$

which are sections of E pulled back over local neighborhood $U \subset E$ via the inclusion map $\iota : E \to U$. The collation of sections enabled by sheafs is necessary for navigation techniques such as pan and zoom [50] and dynamically updated visualizations such as sliding windows [27,28]

3.2 Graphic Bundle

We introduce a graphic bundle to hold the essential information necessary to render a graphical design constructed by the artist. As with the data, we can represent the target graphic as a section ρ of a bundle (H, S, π, D)

$$\begin{tikzcd} D \arrow[r, hook] & H \arrow[d, "\pi"'] \\ & S \arrow[u, bend right, "\rho"'] \end{tikzcd} \tag{12}$$

where ρ is a fully specified graphic such that it is an abstraction of rendering. To fully specify the visual characteristics of the image, we construct a fiber D that is an infinite resolution version of the target space. Typically H is trivial and therefore sections can be thought of as mappings into D. In this work, we assume a 2D opaque image $D = \mathbb{R}^5$ with elements $(x, y, r, g, b) \in D$ such that a rendered graphic only consists of 2D position and color. By abstracting the target display space as D, the model can support different targets, such as a 2D screen or 3D printer.

3.2.1 Equivalant Continuity: Graphic Base Space

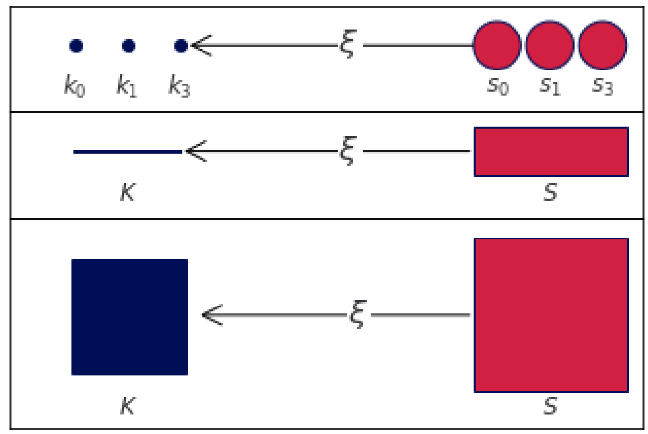


Fig. 6: The 0D scatter k and 1D line k are thickened into S with coordinates $s = (\alpha, \beta)$ that are a region in an idealized 2D screen. The image has the same dimension in S as in K.

Just as the K encodes the connectivity of the records in the data, we propose an equivalent S that encodes the connectivity of the rendered elements of the graphic. Formally, we require that K be a deformation retract [6] of S so that K and S have the same homotopy. The surjective map $\xi : S \to K$

$$\begin{tikzcd} E \arrow[d, "\pi"'] & H \arrow[d, "\pi"'] \\ K & S \arrow[l, "\xi"'] \end{tikzcd} \tag{13}$$

goes from region $s \in S_k$ to its associated point s. While S must have the same continuity as K it is sometimes the thickened version shown in Fig. 6. This thickening is necessary when the dimensionality of K is less than the dimensionality of the target display. For example, a k that is a point in 0D K cannot be represented on screen unless it is thickened to 2D to encode the connectivity of the points in D that visually represent the record at k. The ξ mapping is critical to interactive visualizations as it is the map from a region on screen to the data associated with that region. One example is to fill in details in a hover tooltip, another is to convert region selection on S to a query on the data to access the corresponding record components on K.

3.3 Artist

The topological artist A is a map from the sheaf on a data bundle E which is $\mathcal{O}(E)$ to the sheaf on the graphic bundle H, $\mathcal{O}(H)$.

$$A : \mathcal{O}(E) \to \mathcal{O}(H) \tag{14}$$

that carries a homomorphism of monoid actions $\varphi : M \to M'$ [25]. Given M on data $\mathscr{E}$ and M' on graphic $\mathscr{H}$, we propose that artists $\mathscr{A}$ are equivariant maps

$$A(m \cdot r) = \varphi(m) \cdot A(r) \tag{15}$$

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

The monoid equivariant map has two stages: the encoders $\nu : E' \to V$ convert the data components to visual components, and the assembly function $Q : \xi^*V \to H$ composites the fiber components of ξ^*V into a graphic in H.

$$\begin{tikzcd} E \arrow[r, "\nu"] \arrow[dr, "\pi"'] & V \arrow[d, "\pi"] & \xi^*V \arrow[l, "\xi^*"'] \arrow[r, "Q"] \arrow[d, "\xi^*\pi"] & H \arrow[dl, "\pi"] \\ & K & S \arrow[l, "\xi"] & \end{tikzcd} \tag{16}$$

ξ^*V is the visual bundle V pulled back over S via the equivariant continuity map $\xi : S \to K$ introduced in Sect. 3.2.1. The visual bundle (V, K, π, P) is the space of possible parameters of a visualization type, such as a scatter or line plot. As with the data and graphic bundles, the visual bundle is defined by the projection map π

$$\begin{tikzcd} P \arrow[r, hook] & V \arrow[d, "\pi"'] \\ & K \arrow[u, bend right, "\mu"'] \end{tikzcd} \tag{17}$$

where μ is the visual variable encoding, as described by Bertin [14], of the data section τ. The visual fiber P is defined in terms of the input parameters of the visualization library's plotting functions; by making these parameters explicit components of the fiber, we can build consistent definitions and expectations of how these parameters behave. The functional decomposition of the visualization artist facilitates building reusable components at each stage of the transformation because the equivariance constraints are defined on ν, Q, and ξ. We name this map the artist as that is the analogous part of the Matplotlib [39] architecture that builds visual elements.

3.3.1 Visual Component Maps

We define the visual transformers ν

$$\{\nu_0, \dots, \nu_n\} : \{\tau_0, \dots, \tau_n\} \mapsto \{\mu_0, \dots, \mu_n\} \tag{18}$$

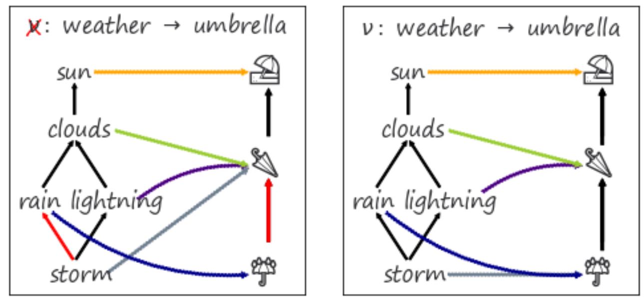
as the set of equivariant maps $\nu_i : \tau_i \mapsto \mu_i$. Given M_i is the monoid action on E_i and that there is a monoid $M_i{}'$ on V_i, then there is a monoid homomorphism from $\varphi : M_i \to M_i{}'$ that ν must preserve. As mentioned in Sect. 3.1.2, monoid actions define the structure on the fiber components and are therefore the basis for equivariance. A validly constructed ν is one where the diagram of the monoid transform m commutes

$$\begin{tikzcd} E_i \arrow[r, "\nu_i"] \arrow[d, "m_r"'] & V_i \arrow[d, "m_\nu"] \\ E_i \arrow[r, "\nu_i"] & V_i \end{tikzcd} \tag{19}$$

such that applying equivariant monoid actions to E_i and V_i preserves the map $\nu_i : E_i \to V_i$. In general, the data fiber F_i cannot be assumed to be of the same type as the visual fiber P_i and the actions of M on F_i cannot be assumed to be the same as the actions of M' on P; therefore an equivariant ν_i must satisfy the constraint

$$\nu_i(m_r(E_i)) = \varphi(m_r)(\nu_i(E_i)) \tag{20}$$

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



(a) (b)

Fig. 7: As highlighted by the red arrows, the equivariance constraint is not met because $\mathrm{rain} \geqslant \mathrm{storm}$, but $\nu(\mathrm{rain}) \leqslant \nu(\mathrm{storm})$. Instead, mapping *storm* to the same element as *rain* yields an equivariant ν such that $\nu(\mathrm{rain}) \geqslant \nu(\mathrm{storm})$.

scale	group	constraint
nominal	permutation	if $r_1 \neq r_2$ then $\nu(r_1) \neq \nu(r_2)$
ordinal	monotonic	if $r_1 \leqslant r_2$ then $\nu(r_1) \leqslant \nu(r_2)$
interval	translation	$\nu(x + c) = \nu(x) + c$
ratio	scaling	$\nu(xc) = \nu(x) * c$

Table 1

We can state the conditions on ν such that they cover the Stevens measurement scales [61], as defined in Table 1. This is because Stevens' defined the measurement scales in terms of their mathematical group structure and a group is a monoid with inverses [53]. An example of monoid action equivariance is the preservation of partial ordering illustrated in Fig. 7.

3.3.2 Visualization Assembly

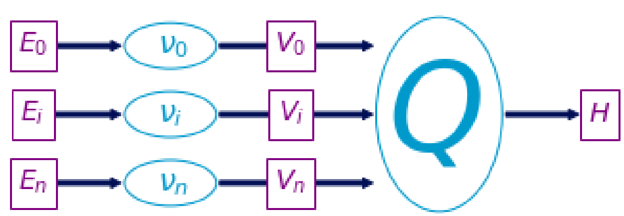


Fig. 8: The transform functions ν_i convert data $\tau_i \in E$ to visual characteristics $\mu_i \in V$, then Q assembles μ_i into a graphic $\rho \in H$.

As shown in Fig. 8, ν and Q are analogous to a map-reduce operation: data components E_i are mapped into visual components V_i that are reduced into a graphic in H. The space of all graphics that Q can generate is the subset of graphics reachable via applying the reduction function $Q(\Gamma(V)) \in \Gamma(H)$ to the visual section $\mu \in \Gamma(V)$. We formalize the expectation that visualization generation functions parameterized in the same way should generate the same functions as the equivariant map $Q : \mu \mapsto \rho$. We then define the constraint on Q such that if Q is applied to two visual sections μ and μ' that generate the same ρ then the output of μ and μ' acted on by the same monoid m must be the same. We do not define monoid actions on all of $\Gamma(H)$ because there may be graphics $\rho \in \Gamma(H)$ for which we cannot construct a valid mapping from V. Lets call the visual representations of the

Fig. 9: These two glyphs are generated by the same annulus Q function. The monoid action m_i on edge thickness μ_i of the first glyph yields the thicker edge ${\mu_i}'$ in the second glyph.

components $\Gamma(V) = X$ and the graphic $Q(\Gamma(V)) = Y$

Proposition 1 *If for elements of the monoid* $m \in M$ *and for all* $\mu, \mu' \in X$, *we define the monoid action on* X *so that it is by definition equivariant*

$$Q(\mu) = Q(\mu') \implies Q(m \circ \mu) = Q(m \circ \mu') \tag{21}$$

then a monoid action on Y *can be defined as* $m \circ \rho = \rho'$. *If and only if* Q *satistfies Equation 21, we can state that the transformed graphic* $\rho' = Q(m \circ \mu)$ *is equivariant to a monoid action applied on* Q *with input* $\mu \in Q^{-1}(\rho)$ *that must generate valid* ρ.

For example, given fiber $P = (xpos, ypos, color, thickness)$, then sections $\mu = (0,0,0,1)$ and $Q(\mu) = \rho$ generates a piece of the thin hollow circle. The action $m = (e, e, e, x+2)$, where e is identity, translates μ to $\mu' = (e, e, e, 3)$ and the corresponding action on ρ causes $Q(\mu')$ to be the thicker circle in Fig. 9.

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

$$J = \{j \in K \text{ s. t. } \exists \gamma \text{ s.t. } \gamma(0) = k \text{ and } \gamma(1) = j\} \tag{22}$$

where the path [3] γ from k to j is a continuous function from the interval [0,1]. We define the glyph as the graphic generated by $Q(S_j)$

$$H \underset{\rho(S_j)}{\overset{}{\rightleftarrows}} S_j \underset{\xi^{-1}(J)}{\overset{\xi(s)}{\rightleftarrows}} J_k \tag{23}$$

such that for every glyph there is at least one corresponding region on K, in keeping with the definition of glyph as any differentiable element put forth by Ziemkiewicz and Kosara [73]. The primitive point, line, and area marks [14,23] are specially cased glyphs.

In Fig. 1, we illustrate the output of a minimal Q that will generate distinguishable graphical marks: non-overlapping scatter points, a non-infinitely thin line, and an image. The scatter plot can be defined as

$$Q(xpos, ypos)(\alpha, \beta) \tag{24}$$

with a constant *size* and color $\rho_{RGB} = (0,0,0)$ that are defined as part of Q. The position of this swatch of color can be computed relative to the location on the disc $(\alpha, \beta) \in S_k$ as shown in Fig. 10

$$x = size * \alpha \cos(\beta) + xpos$$
$$y = size * \alpha \sin(\beta) + ypos$$

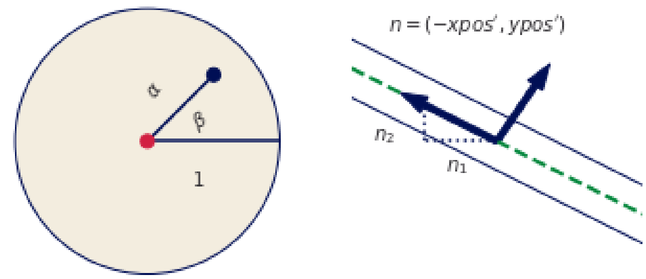


Fig. 10: The coordinates $s = (\alpha, \beta)$ dictate the color of the region in prerender space S. When Q is applied over the whole disk S_k, it generates the graphical point mark. The line fiber is thickened with the derivative because the tangent the line needs to be pushed perpendicular to the tangent of (xpos, ypos) in order to have visible thickness.

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

$$Q(xpos, \hat{n}_1, ypos, \hat{n}_2)(\alpha, \beta) \tag{25}$$

in Fig. 1 has a ξ function that is not only parameterized on k but also on the α distance along k and corresponding region in S. As shown in Fig. 10, 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; therefore the artist takes as input the jet bundle [4,49] $\mathcal{J}^2(E)$ which is the data E and the first and second derivatives of E. The magnitude of the slope is $|n| = \sqrt{{n_1}^2 + {n_2}^2}$ such that the normal is $\hat{n}_1 = \frac{n_1}{|n|}$, $\hat{n}_2 = \frac{n_2}{|n|}$ which yields components of ρ

$$x = xpos(\xi(\alpha)) + width * \beta \hat{n}_1(\xi(\alpha))$$
$$y = ypos(\xi(\alpha)) + width * \beta \hat{n}_2(\xi(\alpha))$$

where (x,y) look up the position $\xi(\alpha)$ on the data and the derivatives $\hat{n}_1, \hat{n}_2$. The derivatives are then multiplied by a $width$ parameter to specify the thickness. In Fig. 1, the image

$$Q(xpos, ypos, color) \tag{26}$$

is a direct lookup into $\xi : S \to K$. The indexing variables (α, β) define the distance along the space, which is then used by ξ to map into K to lookup the color values

$$R = R(\xi(\alpha, \beta)),\ G = G(\xi(\alpha, \beta)),\ B = B(\xi(\alpha, \beta))$$

In the case of an image, the indexing mapper 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.

3.3.3 Assembly Factory

The graphic base space S is not accessible in many architectures, including Matplotlib; instead we can construct a factory function $\hat{Q}$ over K that can build a Q. As shown in Equation 16, Q is a bundle map $Q : \xi^*V \to H$ where ξ^*V and H are both bundles over S.

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

$$\xi^*V\,|_{\xi^{-1}(k)} = \xi^{-1}(k) \times P \tag{27}$$

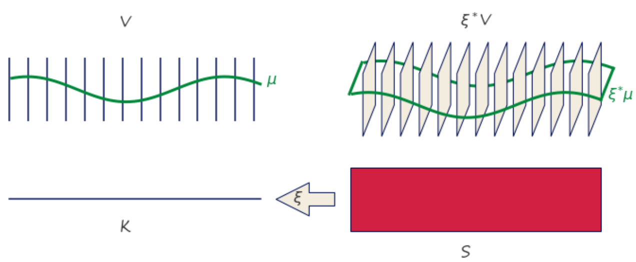


Fig. 11: Because the pullback of the visual bundle ξ^*V is the replication of a μ over all points s that map back to a single k, we can constract a $\hat{Q}$ on μ over k that will fabricate the Q for the equivalent region of s associated to that k

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

As shown in Fig. 11, given the section $\xi^*\mu$ pulled back from μ and the point $s \in \xi^{-1}(k)$, there is a direct map $(k, \mu(k)) \mapsto (s, \xi^*\mu(s))$ from μ over k to the section $\xi^*\mu$ over s. This means that the pulled back section $\xi^*\mu(s) = \xi^*(\mu(k))$ is the section μ copied over all ssuch that $\xi^*\mu$ is identical for all s where $\xi(s) = k$. In Fig. 11 each dot on P is equivalent to the line on $P^*\mu$.

Given the equivalence between μ and $\xi^*\mu$ defined above, the reliance on S can be factored out. When Q maps visual sections into graphics $Q : \Gamma(\xi^*V) \to \Gamma(H)$, if we restrict Q input to $\xi^*\mu$ then the graphic section ρ evaluated on a visual region s

$$\rho(s) := Q(\xi^*\mu)(s) \tag{28}$$

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

$$\hat{Q}(\mu(k))(s) := Q((\xi^*\mu)(s)) \tag{29}$$

where $\hat{Q}$ with input μ is defined to Q that takes as input the copied section $\xi^*\mu$ such that both functions are evaluated over the same location $\xi^{-1}(k) = s$ in the base space S. Factoring out s from Equation 29 yields

$$\hat{Q}(\mu(k)) = Q(\xi^*\mu) \tag{30}$$

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

3.3.4 Composite and Reusable Artists

Given the family of artists $(E_i : i \in I)$ on the same image, the $+$ operator

$$+ := \bigsqcup_{i \in I} E_i \tag{31}$$

defines a simple composition of artists. When artists share a base space $K_2 \hookrightarrow K_1$, a composition operator can be defined such that the artists are acting on different components of the same section. This type of composition is important for visualizations where elements update together in a consistent way, such as multiple views [9,52] and brush-linked views [13,18]. It is impractical to implement an artist for every single graphic; instead we implement an approximation of the equivalence class of artists

$$\{A \in A' : A_1 \equiv A_2\} \tag{32}$$

Roughly, two artists are equivalent if they have the same visual fiber P assembly function Q and continuity map ξ.

4 Prototype

To build a prototype, we make use of the Matplotlib figure and axes artists [39,40] so that we can initially focus on the data to graphic transformations and exploit the Matplotlib transform stack to convert data coordinates into screen coordinates. While the artist is specified in a fully functional manner in Equation 16, we implement our prototype in a heavily object oriented manner. We do so mostly to more easily manage function inputs, especially parameters that are passed through to the structurally functional transform and draw methods.

```
fig, ax = plt.subplots()
artist = Artist(E, V)
ax.add_artist(artist)
```

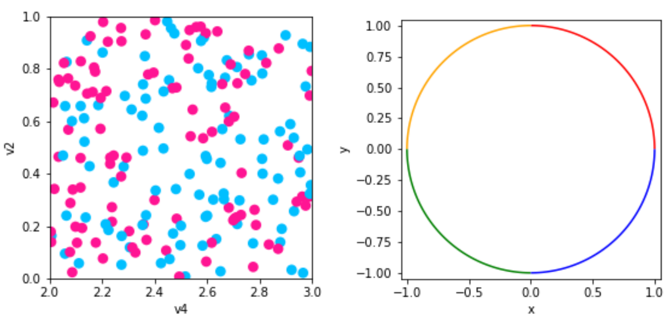
Building on the current Matplotlib artists which construct an internal representation of the graphic, `ArtistClass` acts as an equivalence class artist A' as described in Equation 32. The visual bundle V is specified as the `V` dictionary of the form `{parameter:(variable name, encoder)}` where parameter 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 `E` object. By binding data and transforms to A' inside `__init__`, the `draw` method is a fully specified artist A as defined in Equation 14.

```
class ArtistClass(matplotlib.artist.Artist): #A'
    def __init__(self, E, V, *args, **kwargs):
        # properties that are specific to the graphic
        self.E = E
        self.V = V
        super().__init__(*args, **kwargs)

    def hat_Q(self, **args):
        # set the properties of the graphic

    def draw(self, renderer):
        # returns K, indexed on fiber then key
        tau = self.E.view(self.axes)
        # visual channel encoding applied fiberwise
        mu = {p: nu(tau(c))
                for p, (c, nu) in self.V.items()}
        self.hat_q(**mu)
        # pass configurations off to the renderer
        super().draw(renderer)
```

The data is fetched in section τ via a `view` method on the data because the input to the artist is a section on E. The `view` method takes the `axes` attribute because it provides the region in graphic coordinates S that can be used to query back into data to select a subset as described in Sect. 3.1.5. To ensure the integrity of the section, `view` must be atomic, which means that the values cannot change after the method is called in draw until a new call to draw. We put this constraint on the return of the `view` method so that we do not risk race conditions.

The ν functions are then applied to the data, as describe in Equation 18, to generate the visual section μthat here is the object `V`. The conversion from data to visual space is simplified here to directly show that it is the encoding ν applied to the component. The `q_hat` function that is $\hat{Q}$, as defined in Equation 30, is responsible for generating a representation such that it could be serialized to recreate a static version of the graphic. This artist is not optimized because we prioritized demonstrating the separability of ν and $\hat{Q}$. The last step in the artist function is handing itself off to the renderer. The extra `*arg, **kwargs` arguments in `__init__,draw` are artifacts of how these objects are currently implemented.

4.1 Scatter and Line Artists



(a) (b)

Fig. 12: Scatter plot and line plot implemented using prototype artists and data models, building on Matplotlib rendering.

The figure in Fig. 12a is described by Equation 24. This is implemented via a `Line` object where the scatter marker shape is fixed as a circle, and the visual fiber components are x and y position and the facecolor and size of the marker. We only show the `q_hat` function here because the `__init__, draw` are identical the prototype artist.

The `view` method repackages the data as a fiber component indexed table of vertices. Even though the `view` is fiber indexed, each vertex at an index khas corresponding values in section $\tau(k_i)$. This means that all the data on one vertex maps to one glyph.

```
class Point(mcollections.Collection):
  def q_hat(self, x, y, s, facecolors): #\hat{Q}
    # construct geometries of circle glyphs
    self._paths = [mpath.Path.circle((xi,yi), radius=si)
                   for (xi, yi, si) in zip(x, y, s)]
    # set attributes of glyphs, these are vectorized
    # circles and facecolors are lists of the same size
    self.set_facecolors(facecolors)
```

In `q_hat`, the μ components are used to construct the vector path of each circular marker with center `(x,y)` and size `x` and set the colors of each circle. This is done via the `Path.circle` object.

```
class Line(mcollections.LineCollection):
  def q_hat(self, x, y, color): #\hat{Q}
    #assemble line marks as set of segments
    segments = [np.vstack((vx, vy)).T for vx, vy
                in zip(x, y)]
    self.set_segments(segments)
    self.set_color(color)
```

To generate Fig. 12b, the `Line` artist `view` method 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 `q_hat`, described by Equation 25, this visual representation is composed into a set of line segments, where each segment is the array generated by `np.vstack((vx, vy))`. 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.

4.1.1 Visual Encoders

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'})
# {P_i name:{'name':c_i, 'encoder:\nu_i}}
V = {'x': {'name': 'v4', 'encoder': lambda x: x},
     'y': {'name': 'v2', 'encoder': lambda x: x},
     'facecolors': {'name':'v3', 'encoder': cmap},
     's':{'name': None ,
          'encoder': lambda _: itertools.repeat(.02)}}
```

where `lambda x: x` is an identity ν, `{'name':None}` maps into P without corresponding τ to set a constant visual value, and `color.Categorical` is a custom ν implemented as a class for reusability.

```
#\nu_i(m_r(E_i)) = \varphi(m_r)(\nu_i((E_i))
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)
```

As described in Equation 20, a test for equivariance can be implemented trivially. It is currently factored out of the artist for clarity.

4.1.2 Data Model

The data input into the `Artist` will often be a wrapper class around an existing data structure. This wrapper object must specify the fiber components F and connectivity K and have a `view` method that returns an atomic object that encapsulates τ. To support specifying the fiber bundle, we define a `FiberBundle` data class [1]

```
@dataclass
class FiberBundle:
    K: dict #{'tables': []}
    F: dict #  {variable name: type}
```

that asks the user to specify the the properties of F and the K connectivity as either discrete vertices or continuous data along edges. To generate the scatter plot and the line plot, the distinction is in the `tau` method that is the section.

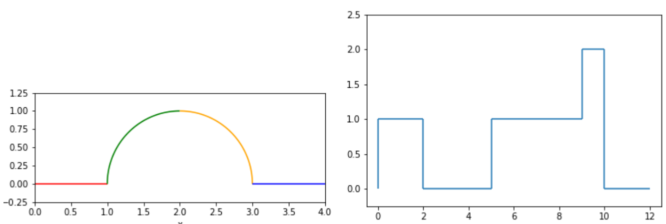
```
class PointData:
    def __init__(self):
      FB = FiberBundle({'tables': ['vertex']},
                       {'v1': float, 'v2': str, 'v3': float})
    def tau(self, k):
      return # tau evaluated at one point k
```

```python

class LineData:
  def __init__(self):
    FB = FiberBundle({'tables': ['edge']},
                     {'x': float, 'y': float, 'color': str})
  def tau(self, k):
    return # tau evaluated on interval k
```

The discrete tau method returns a record of discrete points, akin to a row in a table, while the linetau returns a sampling of points along an edge k.

```python
def view(self, axes):
    table = defaultdict(list)
    for k in self.keys:
        table['index'].append(k)
        for (name, value) in zip(self.FB.fiber.keys(),
                                 self.tau(k)[1]):
            table[name].append(value)
    return table
```

In both cases, the view method packages the data into a data structure that the artist can unpack via data component name, akin to a table with column names when K is 0 or 1 D.



(a) (b)

Fig. 13: Continuous and discontinuous lines as defined via the same data model, and generated with the same A'Line

The graphics in figure Fig. 13 are made using the Line artist and the GraphData data source 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,

```python
GraphData(FB, edges, verticies, num_samples=1000, connect=True)
```

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

```python
GraphData(FB, edges, verticies, 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.

5 Discussion

This work contributes a functional model of the structure-preserving maps from data to visual representation to guide the development of visualization libraries, thereby providing a means to express the constraints of preserving the data continuity in the graphic and faithfully translating the properties of the data variables into visual variables. Combining Butler's proposal of a fiber bundle model of visualization data with Spivak's formalism of schema lets this model support a variety of datasets, including discrete relational tables, multivariate high resolution spatio temporal datasets, and complex networks. Decomposing the artist into encoding ν, assembly Q, and reindexing ξ provides the specifications that the graphic must have continuity equivalent to the data, and that the visual characteristics of the graphics are equivariant to their corresponding components under monoid actions. This model defines these constraints on the transformation function such that they are not specific to any one type of encoding or visual characteristic. Encoding the graphic space as a fiber bundle provides a structure rich abstraction of the target graphical design in the target display space. 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 graphic formation into assembly functions allows for much more clarity in how they differ. This prototype demonstrates that this framework can generate the fundamental point (scatter plot) and line (line chart) marks.

5.1 Limitations

Our model and prototype are deeply tied to Matplotlib's existing architecture, so it has not yet been worked through how the model generalizes to libraries such as VTK or D3. Even though the model is designed to be backend and format independent, it has only been tested against PNGs rendered with AGG [56]. It is unknown how this framework interfaces with high performance rendering libraries such as openGL [24] that implement different models of ρ. While our model supports equivariance of figurative glyphs [21] generated from data components [12,22], it cannot evaluate the semantic accuracy of the figurative representation. Effectiveness criteria [26,45] are out of scope.

5.2 Future Work

More work is needed to formalize the composition operators and equivalence class A'. We also need to implement artists that demonstrate that the model can underpin a minimally viable library, foremost an image [33,36], a heatmap [43,70], and an inherently computational artist such as a boxplot [68]. Since this model formalizes notions of structure preservation, it can serve as a good base for tools that assess quality metrics [15] or invariance [41]. While this paper formulates visualization in terms of monoidal action homomorphisms between fiber bundles, the model lends itself to a categorical formulation [30,47] that could be further explored.

6 Conclusion

The data model and functional transform refactor presented here allows building block libraries to better support downstream, including domain specific, libraries without having to explicitly incorporate the specific data structure and visualization needs of those domains without having to fold domain specific assumptions back into the base library. Adopting this model would induce a separation of data representation and visual representation that, for example, in Matplotlib is so entangled that it has lead to a brittle and sometimes incoherent API and internal code base. A TEAM driven refactor would produce more maintainable, reusable, and extensible code, leading to better building blocks for the ecosystem of tools built on top of Matplotlib.

Acknowledgments

The authors wish to thank the Matplotlib development team for their invaluable feedback along the way, particulary Bruno Beltran, Eric Schles and Chana Tilevitz for idenfiying the confusing bits, Nicolas Kruchten for articulating the framing, and Marc Hanwell, Lev Manovich, Robert Haralick and Huy Vo for very helpful feedback on various iterations. This paper is indebted to Kindlmann and Scheidegger's [41] for serving as a model and Munzner's guide to writing visualization papers. This project has been made possible in part by grant number 2019-207333 from the Chan Zuckerberg Initiative DAF, an advised fund of Silicon Valley Community Foundation.

References

[1] Dataclasses — Data Classes — Python 3.9.2rc1 documentation. https://docs.python.org/3/library/dataclasses.html.

[2] Locally trivial fibre bundle - Encyclopedia of Mathematics. https://encyclopediaofmath.org/wiki/Locally_trivial_fibre_bundle.

[3] Connected space. *Wikipedia*, Dec. 2020.

[4] Jet bundle. *Wikipedia*, Dec. 2020.

[5] Quotient space (topology). *Wikipedia*, Nov. 2020.

[6] Retraction (topology). *Wikipedia*, July 2020.

[7] Monoid. *Wikipedia*, Jan. 2021.

[8] Semigroup action. *Wikipedia*, Jan. 2021.

[9] Y. Albo, J. Lanir, P. Bak, and S. Rafaeli. Off the Radar: Comparative Evaluation of Radial Visualization Solutions for Composite Indicators. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):569–578, Jan. 2016. doi: 10.1109/TVCG.2015.2467322

[10] P. D. Auroux. Math 131: Introduction to Topology. p. 113.

[11] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An Open Source Software for Exploring and Manipulating Networks. *Proceedings of the International AAAI Conference on Web and Social Media*, 3(1), Mar. 2009.

[12] F. Beck. Software Feathers figurative visualization of software metrics. In *2014 International Conference on Information Visualization Theory and Applications (IVAPP)*, pp. 5–16, Jan. 2014.

[13] R. A. Becker and W. S. Cleveland. Brushing Scatterplots. *Technometrics*, 29(2):127–142, May 1987. doi: 10.1080/00401706.1987.10488204

[14] J. Bertin. *Semiology of Graphics : Diagrams, Networks, Maps*. ESRI Press, Redlands, Calif., 2011.

[15] E. Bertini, A. Tatu, and D. Keim. Quality metrics in high-dimensional data visualization: An overview and systematization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2203–2212, 2011.

[16] M. Bostock, V. Ogievetsky, and J. Heer. D^3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec. 2011. doi: 10.1109/TVCG.2011.185

[17] R. Brüggemann and G. P. Patil. *Ranking and Prioritization for Multi-Indicator Systems: Introduction to Partial Order Applications*. Springer Science & Business Media, July 2011.

[18] A. Buja, J. A. McDonald, J. Michalak, and W. Stuetzle. Interactive data visualization using focusing and linking. In *Proceedings of the 2nd Conference on Visualization '91*, VIS '91, pp. 156–163. IEEE Computer Society Press, Washington, DC, USA, 1991.

[19] D. M. Butler and S. Bryson. Vector-Bundle Classes form Powerful Tool for Scientific Visualization. *Computers in Physics*, 6(6):576, 1992. doi: 10.1063/1.4823118

[20] D. M. Butler and M. H. Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3(5):45, 1989. doi: 10.1063/1.168345

[21] L. Byrne, D. Angus, and J. Wiles. Acquired Codes of Meaning in Data Visualization and Infographics: Beyond Perceptual Primitives. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):509–518, Jan. 2016. doi: 10.1109/TVCG.2015.2467321

[22] L. Byrne, D. Angus, and J. Wiles. Figurative frames: A critical vocabulary for images in information visualization. *Information Visualization*, 18(1):45–67, Aug. 2017. doi: 10.1177/1473871617724212

[23] S. Carpendale. Visual Representation from Semiology of Graphics by J. Bertin.

[24] G. S. Carson. Standards pipeline: The OpenGL specification. *SIGGRAPH Comput. Graph.*, 31(2):17–18, May 1997. doi: 10.1145/271283.271292

[25] A. M. Cegarra. Cohomology of monoids with operators. In *Semigroup Forum*, vol. 99, pp. 67–105. Springer, 2019.

[26] J. M. Chambers, W. S. Cleveland, B. Kleiner, and P. A. Tukey. *Graphical Methods for Data Analysis*, vol. 5. Wadsworth Belmont, CA, 1983.

[27] C.-S. J. Chu. Time series segmentation: A sliding window approach. *Information Sciences*, 85(1):147–173, July 1995. doi: 10.1016/0020-0255(95)00021-G

[28] M. S. Crouch, A. McGregor, and D. Stubbs. Dynamic graphs in the sliding-window model. In *European Symposium on Algorithms*, pp. 337–348. Springer, 2013.

[29] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphviz— Open Source Graph Drawing Tools. In P. Mutzel, M. Jünger, and S. Leipert, eds., *Graph Drawing*, pp. 483–484. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[30] B. Fong and D. I. Spivak. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. Cambridge University Press, first ed., July 2019. doi: 10.1017/9781108668804

[31] R. Ghrist. Homological algebra and data. *Math. Data*, 25:273, 2018.

[32] R. W. Ghrist. *Elementary Applied Topology*, vol. 1. Createspace Seattle, 2014.

[33] R. B. Haber and D. A. McNabb. Visualization idioms: A conceptual model for scientific visualization systems. *Visualization in scientific computing*, 74:93, 1990.

[34] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In G. Varoquaux, T. Vaught, and J. Millman, eds., *Proceedings of the 7th Python in Science Conference*, pp. 11–15. Pasadena, CA USA, 2008.

[35] P. Hanrahan. VizQL: A language for query, analysis and visualization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, p. 721. Association for Computing Machinery, New York, NY, USA, 2006. doi: 10.1145/1142473.1142560

[36] C. D. Hansen and C. R. Johnson. *Visualization Handbook*. Elsevier, 2011.

[37] M. D. Hanwell, K. M. Martin, A. Chaudhary, and L. S. Avila. The Visualization Toolkit (VTK): Rewriting the rendering code for modern graphics cards. *SoftwareX*, 1-2:9–12, Sept. 2015. doi: 10.1016/j.softx.2015.04.001

[38] J. Heer and M. Agrawala. Software design patterns for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):853–860, 2006. doi: 10.1109/TVCG.2006.178

[39] J. Hunter and M. Droettboom. The Architecture of Open Source Applications (Volume 2): Matplotlib. https://www.aosabook.org/en/matplotlib.html.

[40] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science Engineering*, 9(3):90–95, May 2007. doi: 10.1109/MCSE.2007.55

[41] G. Kindlmann and C. Scheidegger. An Algebraic Process for Visualization Design. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2181–2190, Dec. 2014. doi: 10.1109/TVCG.2014.2346325

[42] W. A. Lea. A formalization of measurement scale forms. p. 44.

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

[44] K. C. Louden. *Programming Languages : Principles and Practice*. Brooks/Cole, Pacific Grove, Calif, 2010.

[45] J. Mackinlay. *Automatic Design of Graphical Presentations*. PhD Thesis, Stanford, 1987.

[46] J. Mackinlay, P. Hanrahan, and C. Stolte. Show me: Automatic presentation for visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1137–1144, Nov. 2007. doi: 10.1109/TVCG.2007.70594

[47] B. Milewski. Category Theory for Programmers. p. 498.

[48] T. Munzner. *Visualization Analysis and Design*. AK Peters Visualization Series. CRC press, Oct. 2014.

[49] J. Musilová and S. Hronek. The calculus of variations on jet bundles as a universal approach for a variational formulation of fundamental physical theories. *Communications in Mathematics*, 24(2):173–193, Dec. 2016. doi: 10.1515/cm-2016-0012

[50] D. Nekrasovski, A. Bodnar, J. McGrenere, F. Guimbretière, and T. Munzner. An evaluation of pan & zoom and rubber sheet navigation with and without an overview. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pp. 11–20. Association for Computing Machinery, New York, NY, USA, 2006. doi: 10.1145/1124772.1124775

[51] nLab authors. Action. Mar. 2021.

[52] Z. Qu and J. Hullman. Keeping multiple views consistent: Con-

straints, validations, and exceptions in visualization authoring. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):468–477, Jan. 2018. doi: 10.1109/TVCG.2017.2744198

[53] C. Remling. Algebra (Math 5353/5363) Lecture Notes.

[54] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative interaction design for data visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, pp. 669–678. ACM, Honolulu Hawaii USA, Oct. 2014. doi: 10.1145/2642918.2647360

[55] C. A. Schneider, W. S. Rasband, and K. W. Eliceiri. NIH Image to ImageJ: 25 years of image analysis. *Nature Methods*, 9(7):671–675, July 2012. doi: 10.1038/nmeth.2089

[56] M. Shemanarev. Anti-Grain Geometry. https://antigrain.com/.

[57] N. Sofroniew, T. Lambert, K. Evans, P. Winston, J. Nunez-Iglesias, G. Bokota, K. Yamauchi, A. C. Solak, ziyangczi, G. Buckley, M. Bussonnier, D. D. Pop, T. Tung, V. Hilsenstein, Hector, J. Freeman, P. Boone, alisterburt, A. R. Lowe, C. Gohlke, L. Royer, H. Har-Gil, M. Kittisopikul, S. Axelrod, kir0ul, A. Patil, A. McGovern, A. Rokem, Bryant, and G. Peña-Castellanos. Napari/napari: 0.4.5rc1. Zenodo, Feb. 2021. doi: 10.5281/zenodo.4533308

[58] E. Spanier. *Algebraic Topology*. McGraw-Hill Series in Higher Mathematics. Springer, 1989.

[59] D. I. Spivak. SIMPLICIAL DATABASES. p. 35.

[60] D. I. Spivak. Databases are categories, June 2010.

[61] S. S. Stevens. On the Theory of Scales of Measurement. *Science*, 103(2684):677–680, 1946.

[62] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, Jan. 2002. doi: 10.1109/2945.981851

[63] S. Studies. Culturevis/imageplot, Jan. 2021.

[64] M. Tory and T. Moller. Rethinking visualization: A high-level taxonomy. In *IEEE Symposium on Information Visualization*, pp. 151–158, 2004. doi: 10.1109/INFVIS.2004.59

[65] D. Urbanik. A Brief Introduction to Schemes and Sheaves. p. 16.

[66] J. VanderPlas, B. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert. Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software*, 3(32):1057, Dec. 2018. doi: 10.21105/joss.01057

[67] H. Wickham. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016.

[68] H. Wickham and L. Stryjewski. 40 years of boxplots. *The American Statistician*, 2011.

[69] L. Wilkinson. *The Grammar of Graphics*. Statistics and Computing. Springer-Verlag New York, Inc., New York, 2nd ed ed., 2005.

[70] L. Wilkinson and M. Friendly. The History of the Cluster Heat Map. *The American Statistician*, 63(2):179–184, May 2009. doi: 10.1198/tas.2009.0033

[71] K. Wongsuphasawat. Navigating the Wide World of Data Visualization Libraries (on the web), 2021.

[72] B. A. Yorgey. Monoids: Theme and Variations (Functional Pearl). p. 12.

[73] C. Ziemkiewicz and R. Kosara. Embedding Information Visualization within Visual Representation. In Z. W. Ras and W. Ribarsky, eds., *Advances in Information and Intelligent Systems*, pp. 307–326. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi: 10.1007/978-3-642-04141-9_15