

Delta: A DSL for Enlivening Typeset Formulas with Interactive Explanations

Eric Dai
Computer and Information Science
University of Pennsylvania
Philadelphia, Pennsylvania, USA
ericdai@seas.upenn.edu

Zain Khan
Computer and Information Science
University of Pennsylvania
Philadelphia, Pennsylvania, USA
zainkhan@seas.upenn.edu

Jeffrey Tao
Computer and Information Science
University of Pennsylvania
Philadelphia, Pennsylvania, USA
jeffttao@seas.upenn.edu

Marti A. Hearst
School of Information
UC Berkeley
Berkeley, California, USA
hearst@berkeley.edu

Andrew Head
Computer and Information Science
University of Pennsylvania
Philadelphia, Pennsylvania, USA
head@seas.upenn.edu

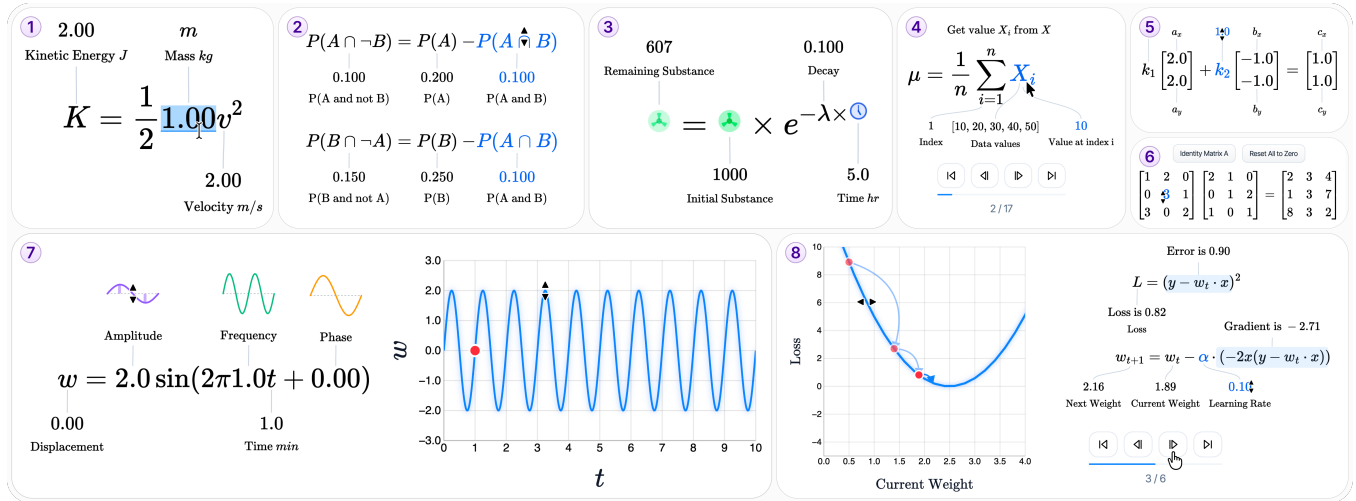


Figure 1: Delta helps authors turn static formulas into interactive explanations where readers can manipulate variables, step through computations, and explore meaning through linked visualizations. The examples above demonstrate the range of interactions one can implement with Delta, from basic variable scrubbing (1-3, 5-8) to systems of formulas (2, 8), graphical equations (3, 7), and linked interactive visuals (7, 8). See Section 4.3 for detailed descriptions of each example.

Abstract

Many important mathematical ideas are described with formulas, but formulas can be difficult to understand. This has prompted some authors to embed formulas in interactive explanations that relate symbols to text, computation, and visualizations. However, implementing such designs with standard tools is difficult and time-consuming. To lower the threshold and raise the ceiling of formula explanations, we design and implement Delta, a TypeScript-embedded domain-specific language for creating interactive formula explanations. Delta gives authors the ability to turn formulas

into calculators, explain them with step-by-step walkthroughs, and link symbols to nearby text and visual descriptions. It is designed to integrate with LaTeX math in web-based writing contexts, with a React-based runtime that handles necessary state management. In a controlled study, participants edited a simple interactive formula spec more quickly and easily with Delta than with a custom React baseline. In an observational study, participants largely found the language’s constructs intuitive after an hour of continuous use.

CCS Concepts

• Human-centered computing → Interactive systems and tools.

Keywords

interactive formulas, synchronized visualizations, walkthroughs, domain-specific languages, interactive documents, typesetting



This work is licensed under a Creative Commons Attribution 4.0 International License.
UIST '26, Detroit, MI, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2856-3/26/11

<https://doi.org/10.1145/3830398.3830616>

ACM Reference Format:

Eric Dai, Zain Khan, Jeffrey Tao, Marti A. Hearst, and Andrew Head. 2026. Delta: A DSL for Enlivening Typeset Formulas with Interactive Explanations. In *The 39th Annual ACM Symposium on User Interface Software and Technology (UIST '26)*, November 02–05, 2026, Detroit, MI, USA. ACM, New York, NY, USA, 21 pages. <https://doi.org/10.1145/3830398.3830616>

1 Introduction

Many important ideas in science are conveyed with formulas. For instance, in computer science, formulas define the statistical models we use to test our hypotheses, provide the backbone of proofs of correctness, and formally describe new techniques in systems and AI models. However, formulas are not always easy to understand. Professionals have described notation as a barrier to learning and reading [5, 25]. Teachers have described students struggling to learn to wield notation well [11]. When formulas use unfamiliar notation or require specialized symbolic reading skills, they can hinder rather than support comprehension [19, 27].

Perhaps to address some of these issues, math communicators have experimented with various ways to present formulas to make their meaning more self-evident. These experiments go beyond the graphs a student might see on the chalkboard of an algebra class or the screen of their graphing calculator. In interactive textbooks [23, 30] and articles [9], formulas have become part of narrative texts where they are inter-linked with sentences, simulations, and visualizations. Some authors have indicated a desire for tools to make their formula descriptions dynamic [15].

One challenge to making these kinds of explanations is implementation overhead. Consider an author wishing to turn a formula in a web article into a calculator where a user can plug in example values (e.g., Figure 1.1). The author must recover the web elements for relevant variables, attach event handlers, define computations, and re-render the formula as values change. If the author wants to do something more complex—involving multiple formulas, multi-step walkthroughs, or linked visualizations—they must shoulder the work of collecting and propagating state, computation, transitions, and selections across formulas, text, and visuals.

To reduce this overhead and promote better explanations of formulas, we developed *Delta*. Delta is a TypeScript-embedded domain-specific language (DSL) for enlivening formulas. Delta formulas respond to reader interaction. They show computed values and visuals in place of symbols, support step-by-step walkthrough explanations, and change in response to user interaction with nearby formulas, text, and visuals (Figure 1). Delta provides a compact specification language for defining these kinds of designs and handles much of the overhead of setting them up and managing them.

This paper makes two contributions. The first is the design of Delta, comprising a set of constructs for authoring dynamic formulas in interactive documents. The second contribution is evidence of its fit for the envisioned task. Across 2 studies, 19 participants augmented formulas using Delta. In the first study, they were able to make changes to a spec for an interactive formula more quickly and easily than with a React baseline that was streamlined to support the task. The second study examined the usability of the constructs after a bounded training session, with participants successfully adding calculation and walkthroughs to formulas. Altogether, this work demonstrates that a purpose-built language can make richly

interactive formulas more practical to authors, laying a foundation for their use in future scientific authoring environments.

2 Background and Related Work

2.1 Reading and augmenting notation

Reading notation can be difficult. Learners have described it as an impediment to self-driven learning in domains like machine learning [5]. Österholm [27] found that adding notation to a text could result in lower performance in a comprehension assessment. When readers review notation, they sometimes do so in concert with reading nearby text: Kohlhase et al. [17] observed readers shifting attention between the two in a math reading task. Changing notation can influence how it is read. Yung and Paas [43] found that the addition of visual cues was associated with lower perceived cognitive load, and Landy and Goldstone [18] found that space in a formula can influence how it is interpreted.

Some authors have experimented with augmenting notation to change how it is read. Head et al. [15] reviewed the work of many such authors and interviewed some of them. This led to the documentation of four patterns of augmentation: embedding visualizations in formulas, styling the formula’s text, adding annotations, and making formulas interactive. Our paper’s contribution is a DSL that helps authors make their formulas interactive. Additionally, it lets authors embed visuals and describe formulas with labels.

We are not aware of studies that demonstrate an effect of interactivity on the understanding of formulas. However, there is reason to believe from educational theory that interactivity may be helpful in the right circumstances. For instance, the DeFT framework [1] identifies three functions by which multiple external representations support learning: providing complementary information or processes, constraining interpretation, and supporting construction of deeper understanding. The feature that we call “walkthroughs” can be viewed as segmented multimedia presentations. These too can be helpful to learners, breaking complex material into chunks that learners can step through at a pace that feels comfortable to them [28].

2.2 Tools to augment notation

If an author were to attempt to augment their notation, one of the tools they might use is $\text{\LaTeX}$ (LaTeX), which offers simple commands for coloring [6] and labeling formulas [22] in its standard libraries. Some community packages offer more aesthetic options (e.g., `annotate-equations` [34]). Authors have reported some of the default LaTeX commands as clunky and difficult to use [15]. Experimental languages and toolkits have been developed to make it easier to augment formulas. Nota [8], Heartdown [20], and FFL [40] are DSLs that help authors define their symbols. The tools render documents to put definitions next to symbols, while letting authors separate definition specifications from the formula markup. Some tools offer more augmentation possibilities. For instance, Heartdown can automatically graph formulas. FFL lets authors write coloring rules that apply to all formulas in a document.

Some DSLs let authors make their formulas dynamic. *manim* [33] lets authors create animations that build up formulas step by step. Its output is video rather than an interactive formula embedded in a typeset document. *FigureOne* [2] lets authors create web-based

math explanations including animated and interactive formulas. The authoring experience is different from Delta: in a walkthrough, each step redefines a formula, and formulas are not atomic LaTeX strings but lists of expressions combined to form each formula. Delta lets authors define their formulas with LaTeX, define walkthroughs using steps in a semantics function, and connect computation to formulas using a data store indexed by a variable's LaTeX.

Beyond DSLs, some tools let authors add augmentation with graphical editing. Freeform [35] and i-LaTeX [12] are two research tools that let authors edit and annotate LaTeX with direct selections, and numerous products provide similar features [24, 26, 37]. Delta could provide a backend representation that enables tools like these to be used to add interactivity to formulas.

2.3 Tools to create explorable explanations

Delta is one tool in an ecosystem of tools for authoring technical explanations. It could be a building block for making explorable explanations, or documents that help readers learn by tinkering with assumptions and observing consequences [38]. Delta's support for interactive calculation follows a pattern common in DSLs for explorable explanations. A central store holds variable values, and controls in the text let readers edit those values and propagate changes elsewhere in the document. This pattern appears in the Potluck [21], Idyll [7], Tangle.js [39], and Living Papers [16] toolkits. Many of these frameworks can support interactive formulas. For instance, in Living Papers [16], an author could bind an input to a reactive variable, interpolate that variable in place of a LaTeX symbol, and use the variable to drive a generated visualization (their paper has an example demonstrating this capability). Versus these frameworks, Delta's contribution is in its accelerators that make common augmentations simple and reduce the cost of a broad set of useful interactive explanation patterns, including constructs for walkthroughs and linked graphs. Some of its most obvious optimizations are in the data store indexed by LaTeX variable expression, symbol input modes, and numeric precision properties.

Delta joins other DSLs for providing better visual explanations of math and science including Penrose [42], Bloom [36], Bluefish [29] and Chisel [14]. These other DSLs help authors create conceptual diagrams to mimic expert outputs and to make beautiful layouts follow from simple specs. Delta aims to do the same for the task of diagramming and explaining formulas.

3 Formative review of artifacts

Delta is designed to reduce the implementation overhead of creating interactive formula explanations. In this section, we describe that overhead more completely.

We acquired 11 documents containing interactive mathematical explanations involving formulas. We seeded our document set with items from Head et al. [15]'s prior content analysis. Then we supplemented the set with entries from community-sourced document lists of explorable explanations at <https://explorabl.es/> and <https://github.com/ubavic/awesome-interactive-math> and results from our own web searches. Documents came from sources including Distill.pub, Setosa.io, Explorable Explanations, complex-analysis.com, Georgia Tech's Interactive Linear Algebra textbook, and personal websites. The analysis was conducted by 2 authors

and covered 46 formulas of interest. Each interactive explanation included either a formula that was responsive to interaction or one that was displayed alongside a visualization or passage of prose with interactive elements.

Augmentation patterns. Of the 46 formulas, 24 were dynamic, changing in response to direct interaction with the formula, one of its elements, or a nearby interactive element in prose or a visualization. 22 featured some kind of link between formula elements and visualization elements. Readers could control formulas using sliders (8), draggable graph points (17), radio buttons (8), scrubbing (5), or direct editing of terms within the formula (1). Walkthroughs—step-by-step navigation through staged derivations—were used within 1 dynamic formula. In 2 other examples, a static formula was displayed alongside a walkthrough presented in nearby prose or visualizations.

Our review also showed some evidence of implementation overhead. We examined the source code for the interactive explanations. This revealed signs of several implementation patterns that could have posed friction to authors (see also appendix Table 1):

- *F1: Diffuseness and tangling (11 articles).* Interactions were implemented across multiple files; and rendering, layout, and behavior could be tangled within a single function.
- *F2: Indirect identifiers (11 articles).* Variables were referred to in the code using identifiers or array indexing distinct from how the variable appeared in the formula.
- *F3: Fussing the DOM and LaTeX (9 articles).* Code sometimes constructed LaTeX strings programmatically, traversed rendered formula DOM, or both. The two exceptions bypassed LaTeX with their own formula rendering engines.
- *F4: Repetition (10 articles).* Interaction logic appeared duplicated across instances of related math formulas.
- *F5: State management (10 articles).* Articles implemented custom state management and wired up drag handlers, click listeners, and other callbacks to propagate changes across formulas, variables, and other representations.

We do not claim these frictions as themselves novel. F1 and F3–5 are general web-development concerns. The use of indirect identifiers for variables (F2) is specific to our domain. Delta was designed to remove much of this overhead and instead allow authors to enliven formulas with a compact language. Section 4.2 relates these frictions to the Delta constructs that address them.

4 Delta

Delta is designed to provide a low threshold and high ceiling for adding interactivity to formulas in web documents. We illustrate the low threshold with a walkthrough of the process of creating simple augmentations in Section 4.1, describe the full set of constructs and our design philosophy in Section 4.2, and hint at the high ceiling with a gallery of examples we created with Delta in Section 4.3.¹

4.1 Basics

To author interactive formulas in Delta, an author starts by defining a `Config`. In this config, the author defines *formulas* with LaTeX, selects *variables* and makes them interactive, and attaches *semantics*

¹The Delta language is hosted at <https://penn-hci.github.io/delta/>.

via a JavaScript function. Optionally, the author specifies *connections* to other visuals and breaks down explanations into *steps*. We introduce each of these parts of the language in the examples below.

4.1.1 Simple calculation. Imagine an author making an interactive calculator from the physics formula, $K = \frac{1}{2}mv^2$. They would like to describe the variables with labels, show readers how different values of mass (m) and velocity (v) correspond to different values of kinetic energy (K), and demonstrate the relationship of K to m and v in a graph (Figure 2.4 and 2.5). The author first defines the base *formula* in standard LaTeX:

```
const kineticConfig: Config = {
  formulas: [
    { id: "kinetic-energy", latex: "K = \\frac{1}{2}mv^2" },
  ]
}
```

Because the formula's LaTeX is written as a JavaScript string, the author needs to escape backslashes in the macro commands ("\\"). The 'id' will serve as an identifier when the author later inserts this formula into a Delta React component. The Config will house the entire specification of the formula's behavior. Next, the author sketches out the basics of interaction by selecting key *variables* and giving them descriptive labels (Figure 2.1) and default values:

```
const kineticConfig: Config = { ...
  variables: {
    K: { name: "Kinetic Energy" },
    m: { name: "Mass", default: 5 },
    v: { name: "Velocity", default: 2 },
  },
}
```

To select a variable, the author uses the same LaTeX string that represents it in the original formula. Variables can be multi-letter expressions, though in this example, they are just single letters.

Then, to render the formula in a React document, the author wraps part of the document in a *provider* component that takes the config as an argument: `<Provider config={kineticConfig}>`. The author can place the formula inside the provider using the React component `<Formula id="kinetic-energy"/>`. Then, Delta renders the formula. It typesets the formula, adds the descriptive labels in the margins, and places the default values next to the labels. The labels are visually linked to symbols: hovering over a value highlights the corresponding term in the formula (Figure 2.2).

To turn the formula into a calculator, the author defines the *semantics* of the formula:

```
semantics: function ({ vars }) {
  vars.K = 0.5 * vars.m * Math.pow(vars.v, 2);
},
```

Semantics are defined using JavaScript functions. Semantics functions serve three roles: they *compute* derived variable values based on inputs, they can *sample* data points to populate linked graphs, and they can generate *steps* in a formula walkthrough including the labels and descriptions that should appear at each step.

In this example we focus on the simplest role: computation. The semantics function uses standard JavaScript to map input variables to outputs, reading m and v off of the `vars` object to compute and then set the variable K . Keys for getting and setting variable values are the same as the variable keys in the `variables` spec.

The formula is now interactive (Figure 2.3): a reader can click and drag m and v to change values and see K update live. The author can further fine-tune the interaction with a few simple properties—for instance, allowing free-text input for mass and defining minimum and maximum values for mass and velocity (for instance, to prevent negative mass values) (Figure 2.4):

```
m: { name: "Mass", default: 5,
      input: "inline", range: [0, 20] },
v: { name: "Velocity", default: 2,
      input: "drag", range: [0, 10] },
```

4.1.2 Linked visuals. The author can also *connect* the formula to text or visualizations elsewhere in the document. One of the accelerators Delta provides is for creating interlinked interactive 2D and 3D plots. To create a plot coupled to the formula so that interacting with either updates the other, the author writes the following:

```
graph2d: [{
  id: "energy-graph",
  xAxisVar: "v", yAxisVar: "K",
  lines: [{
    sampleId: "energy", parameter: "v",
    interaction: ["vertical-drag", "m"],
  }],
  points: [{
    sampleId: "energy",
    interaction: ["horizontal-drag", "v"],
  }],
}],
```

This creates a 2D graph. Its axes correspond to formula variables, with the x-axis representing v and the y-axis representing K . The graph plots data points from a data sample called "energy." The author generates these points with an updated semantics function:

```
semantics: function ({ vars, sample }) {
  vars.K = 0.5 * vars.m * Math.pow(vars.v, 2);
  sample("energy", { x: vars.v, y: vars.K });
},
```

This graph is interactive, with interactions on the plot triggering changes in the formula (Figure 2.5). The optional interaction fields in the `graph2d` config specify how interactions should behave. Vertical drags change the curve by altering m , and horizontal drags move a point along the curve by changing v . Interaction is bidirectional. If a reader changes m or v by interacting with the formula, the plot updates. Conversely, when a reader drags the curve or the point on the curve, the relevant variables in the bound formulas update. The author inserts the graph in the document by adding a Delta Graph component to the provider, passing the component the ID of the graph from the `graph2d` spec.

Simple walkthrough. The author may want to introduce the formula piece by piece, directing attention to parts of the formula one at a time. For example, they may wish to introduce an arithmetic mean $\mu = \frac{1}{n} \sum_{i=1}^n X_i$ formula by adding values to a running sum one at a time and updating a running mean after each addition. To support this style of explanation, Delta lets authors define explanations as a series of *steps*. They do this in the semantics function. The semantics function permits ordering and repetition of steps by nature of being a JavaScript function. Consider the following semantics function for an arithmetic mean:

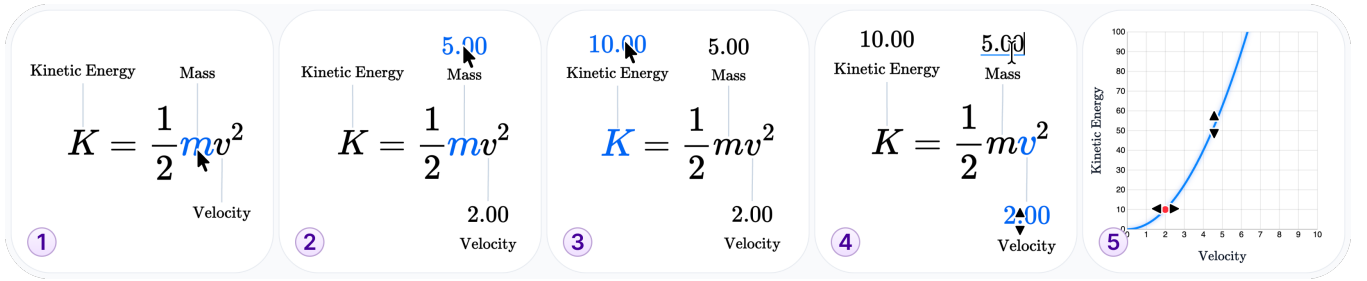


Figure 2: Stages of authoring a simple interactive calculator. (1) Adding labels. (2) Adding default values. (3) Defining semantics for computing derived variables. (4) Changing input modes (dragging and text editing). (5) Enabling bidirectional interactions with a plot.

```

semantics: function ({ vars }) { ...
  vars.n = vars.X.length;
  var sum = 0;
  vars["\\mu"] = 0;
  for (var i = 0; i < vars.n; i++) {
    vars.i = i;
    sum += vars.X[i];
    vars["\\mu"] = sum / (i + 1); // running average
  } ...
},

```

To present a formula using a walkthrough, the author can add descriptive “steps” within the loop:

```

for (var i = 0; i < vars.n; i++) {
  // ...
  step({
    description: "Running average is " + vars["\\mu"],
    labels: {
      "\\sum_{i=1}^{n} X_i": "sum is now " + sum,
      n: vars.n,
      i: i,
      X_i: vars.X[i],
      X: vars.X,
    },
  });
}

```

The resulting walkthrough shows one step per number in the input list. At each step, it shows the index of iteration, the sampled data value, and a running sum and average. Some labels include calculated values. Most labels are anchored to expressions; the description floats above the formula as a global label. The walkthrough can be navigated forward and backward (Figure 3). An arbitrary number of steps can be added. Steps appear in the same order in which they were generated during execution.

4.2 Language Description

Delta’s capabilities extend quite a bit further than this. Appendix Figure 8 provides a full listing of the language. This section provides a more comprehensive tour of the capabilities, options, and design rationale behind the language. It covers Delta’s three main capabilities in enlivening formulas: *substituting* computed values and visuals into formulas (Section 4.2.1), *walkthroughs* that present formulas step by step (Section 4.2.2), and *synchronization* of changes across formulas, text, and visuals (Section 4.2.3). Throughout, we italicize each capability that has corresponding language features.

4.2.1 Substitution. Delta lets authors make static mathematical expressions more expressive and interactive. The language lets authors select *variables* from a formula using the variables’ literal LaTeX and specify how those variables should be augmented. Variables can be *labeled*. They can also be *replaced*, where a concrete value or visual is substituted in place of the variable.

When a value of a variable is shown, either in a label or in a formula, it can be configured with a fixed *precision* or a fixed number of *significant digits*. A value can be made to change upon a user’s *scrubbing* or *text entry*. For the former, an author can bound the space of exploration by setting *defaults*, *range*, and *step* size of value changes. Values can be *numbers* or *flat lists* whose entries are numbers or strings. Each of the above options is accessible through a single property in a variable’s spec. Our intent was to make it trivially possible to define and edit these essential aspects of a variable’s interactivity.

Computed variable values are generated from *semantics* functions that the author defines. These are JavaScript functions that have access to all variables from all formulas declared in a Delta Config. An author does not have to mark specific variables as “inputs” and “outputs” in a formula. Rather, they mark some variables as configurable with the properties above, and then the semantics implicitly drive changes in any output variables set in the semantics function. Prior versions of Delta let authors write semantics as symbolic algebra expressions, but we ultimately opted for JavaScript-based semantics functions due to their expressivity.

Delta lets authors *represent expressions graphically*. The author can provide a JavaScript function that generates an SVG from the available variable values (e.g., clock hands rotating based on time). The SVG is substituted into the variable’s position in the formula.

4.2.2 Walkthroughs. Delta lets authors describe formulas with walkthroughs that comprise *steps*. Steps are specified within the semantics function. As we developed examples for Delta, we realized that step-by-step explanations often involved plugging into complex logic—for instance, to explain a summation, one might want to show example summands, but only for the first few terms. An explanation like this requires repetition and filtering, which can be easily implemented using control-flow constructs like loops and conditionals. We decided to leverage the built-in control flow of JavaScript to allow authors to arrange walkthroughs like this.

In Delta, a step in a walkthrough is a breakpoint in the semantics function. It is written as a function call interleaved with the computation code. Each call to the step function can define labels to be

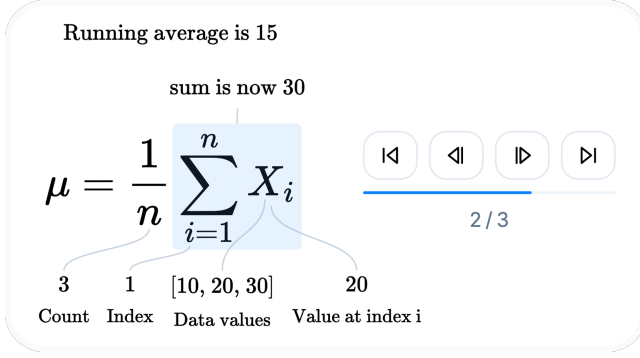


Figure 3: A step of a walkthrough for computing the arithmetic mean. Labels show current values of terms. Controls on the right let readers step forward and backward in the computation. The annotations update to reflect the current step. Variable names are defined with commands not shown in Section 4.

displayed at that point in the walkthrough, including an optional *description* to appear above the formula and *labels* for explaining variables and expressions. Consistent with the rest of the language, variables and expressions are selected for labeling by using their LaTeX. If an author wants to show multiple consecutive steps without advancing computation (e.g., to call attention to one expression and then another), they just call step multiple times, each time identifying a different set of expressions to label.

4.2.3 Synchronization. Delta helps authors weave formulas into interactive explanations containing reactive text and visualizations. It does so by providing components for reactive text and visualizations that communicate with formulas and by exposing reactive data stores to be shared by all components.

If an author wants their formula to communicate with other formulas, text, or visualizations, they must add all relevant components to one Delta *context provider*. All contained components can read from and change variables in the provider’s shared store. For instance, an *inline variable* component “*m*” could be added to prose “The mass is defined by *m*” and told to track with the variable “*m*” defined in a Delta config. The inline variable would now be connected to a variable “*m*” in a Delta formula (e.g., “ $K = \frac{1}{2}mv^2$ ”). Hovering one *m* highlights the other, and scrubbing changes the value of both. Within prose, variables can be rendered as symbols (e.g., “*m*”) or concrete values (“2 kg”).

To create *systems of formulas*, where multiple related formulas share values and can be played with together, an author places multiple formulas in the same context. Delta links repeated appearances of the same variable. Variable specs match variables in all in-context formulas. For example, “ $P(A \cap \neg B) = P(A) - P(A \cap B)$ ” and “ $P(B \cap \neg A) = P(B) - P(A \cap B)$ ” share the variable “ $P(A \cap B)$ ”. Changing its value in either formula triggers both to update.

Delta helps authors generate some of the most basic kinds of formula visualizations. Authors can define *2D* and *3D graphs* that plot formula variables. To populate these graphs, an author *samples* values already in the semantics function. For the kinetic energy formula, for example, an author might plot a kinetic energy curve by sampling values of (v, K) . Samples that include two input variables

can be used to define a surface. Interactions with variables within a formula trigger re-rendering of plots by forcing re-execution of the semantics function with new input values.

Graphs can drive changes to variable values by defining *graph interactions* associated with variables. When such graph interactions are defined, drag actions on lines and points trigger changes to variable values. Graph axes are constrained to the *ranges* of variables from the variables spec. Points and vectors can be added to graphs to visualize specific values from walkthrough steps. Assign an ID to a step and reference that ID in the point or vector definition in the graph config. When the walkthrough reaches the matching step, the corresponding point or vector appears. If configured as *persistent*, points and vectors remain visible and accumulate.

Authors might wish to drive their formulas using custom components and visualizations defined outside of Delta. In such cases, they can wrap their components in Delta’s Custom component. The component receives the context provider’s vars object, which is a reference to a shared variable store that can be read and set.

Across all components, authors read from and set variables by referring to the same LaTeX they are defined with in the formula. An example of an explanation that relates many variables across multiple formulas, inline variables, walkthrough steps, and graphs is shown in Figure 4.

Altogether, this design addresses much of the implementation overhead described in Section 3. Authors are spared from thinking about DOM traversal (friction F3) and custom state wiring (F5). Relevant code is brought into one place (F1) and repetitive interaction logic hidden (F4). Authors are able to refer to variables directly by their LaTeX (F2).

4.3 Example Gallery

Delta’s functionality was built up through a series of increasingly rich examples (Figure 1). In this section, we review the examples to demonstrate Delta’s expressivity. Each example includes a description of the intent behind an interactive explanation and pointers to language features exercised.

Kinetic Energy (1). This is our basic running example. It demonstrates how a formula can serve as a calculator, with input variables controlled with scrubbing and text entry. This example requires a LaTeX *formula* to be defined and *variable specifications* that define *labels* to assign to variables, values to be substituted *in place*, *drag* and *text entry* interactions, numeric *precision* in rendered values, and a *semantics* function driving calculation.

Probability Operations (2). Two formulas—“ $P(A \cap \neg B) = P(A) - P(A \cap B)$ ” and “ $P(B \cap \neg A) = P(B) - P(A \cap B)$ ”—form a system of formulas. They share a common synchronized variable $P(A \cap B)$ that can be changed from either formula. Scrubbing it triggers simultaneous updates to both equations. Synchronization is coordinated by an implicit *shared variable ID* that connects the formulas.

Exponential Decay (3). In this augmented exponential decay formula, dynamic icons visualize key quantities in an application of the formula to radioactive half-life. A radiation sign represents remaining substance *N* and glows or dims as the quantity changes. A clock for elapsed time *t* ticks continuously, faster or slower depending on the value of *t*. This example involves *graphical expressions* that are generated as SVGs parameterized by variable values.

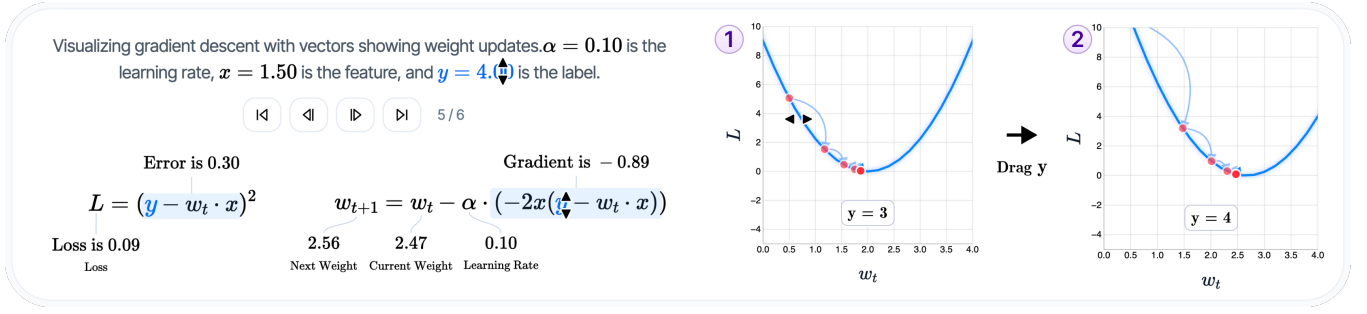


Figure 4: Reactive propagation across walkthrough steps and visualizations. Pictured is an interactive explanation of gradient descent. Dragging y in the prose, in the formula or on the graph’s line (1) triggers re-execution of the semantics function, updates all intermediate step values and adjusts the visualization (2) to reflect a new loss landscape. See Figure 12 for the full Delta configuration.

Arithmetic Mean (4). This basic walkthrough shows the accumulation of a sum in the process of computing a mean. The walkthrough is defined as a series of *steps* in a semantics function, where each step defines *descriptions* to appear above the formula and *labels* that describe and show values of expressions at each step.

Linear Combination of Vectors (5). This formula demonstrates a calculation involving vectors rather than just scalars. Scrubbing over scalar coefficients k_1 and k_2 immediately updates the result vector. It uses the same language features as (1).

Matrix Multiplication (6). In this calculator for a matrix product, a reader can scrub items in 3×3 input matrices and observe changes in the product. This formula uses the same language features as (1), along with *custom components*—external buttons for “Identity Matrix A” and “Reset All to Zero”—that provide accelerators for setting groups of variables at once.

Harmonic Motion (7). This formula is an augmented wave equation with a synchronized time-series plot. Dynamic glyphs appear in the labels above each parameter to visualize them, including a stretched wave for amplitude, compressed oscillations for frequency, and a phase-shifted curve for ϕ . Changes to variable values can be triggered by dragging the label glyphs. In addition, dragging the point on the graph updates t , and dragging the curve updates the amplitude variable in the formula. This example exercises *graphical expressions* in labels, a linked *2D graph*, and *interaction controls* mapping point and curve drag events to variable values.

Gradient Descent (8). Our final example exercises nearly all of Delta’s language features, with the addition of steps in the semantics function adding points to the graph synchronized with walkthrough points. The concept of gradient descent is demonstrated with a linked loss function “ $L = (y - w_t \cdot x)^2$,” weight update rule “ $w_{t+1} = w_t - \alpha \cdot (-2x(y - w_t \cdot x))$,” and loss landscape visualization. As readers step through iterations, points and arrows accumulate on the graph, tracing the optimization path. Dragging the learning rate α or the target value y triggers re-execution of all steps, updating formula annotations and graph elements.

5 Implementation

Delta is a TypeScript-embedded DSL, interpreted by a React-based runtime. The runtime parses Delta specifications, renders interactive formulas and other components, and coordinates shared reactive state changes across components. One of Delta’s goals

is to remove the overhead of making formulas interactive so that authors can instead focus on presentation. The runtime removes overhead in three important ways.

Parsing and rendering. Delta takes care of the work of rendering formulas with computed values and margin labels. Delta parses and augments formulas in a manner largely consistent with known formula augmentation tools [35, 40]. It leverages the parser from KaTeX [10], a widely-used LaTeX web formula renderer, to produce an AST of a formula’s LaTeX. On this AST, it matches variables named in the Config and annotates them with macros that will wrap those variables in CSS IDs at render time. If a spec asks to replace a variable with the value, Delta replaces the variable node with a value node. Delta serializes the parse tree into LaTeX and then passes it to MathJax to render it to HTML.² Each formula subscribes only to the subset of the store containing its relevant variables. Updating a substituted value in this subset triggers the formula to be re-parsed and re-rendered, whereas unrelated changes, such as hover-state updates, do not. Labels are rendered as elements on a ReactFlow canvas and spaced using a simple custom collision resolution algorithm.

Interaction handling. After MathJax renders the augmented LaTeX into the DOM, Delta uses the CSS IDs generated in the earlier step to locate rendered occurrences of variables named in the Config and attach interaction handlers to them. To support continuity of interaction, event handlers are attached to transparent overlay nodes over each variable. When variable values change, Delta removes the previous MathJax-generated DOM content, inserts the updated LaTeX, and asks MathJax to render it. When the formula re-renders, Delta keeps overlay nodes mounted and repositions them over the moving variables to provide continuity in drag actions.

Data collection and updates. State is saved in a shared store, comprising current values of all variables, the hover status of each variable, and steps and samples saved from executing the semantics function. Delta uses MobX, a reactive state management library, to coordinate updates across components within a context provider. When a change is triggered in a variable’s value, MobX triggers

²MathJax’s CommonHTML output exposes a MathML-derived element hierarchy and preserves custom IDs on corresponding DOM elements. We found this structure more convenient for locating and measuring annotated formula elements than KaTeX’s span-based visual markup, although KaTeX supports equivalent HTML annotations.

re-execution of the semantics function and then re-rendering of any affected formula components. Other components, such as inline variables embedded in prose and custom components, access the shared variable store through Delta’s Provider, which uses React Context, and are wrapped with MobX’s observer to re-render automatically when the values they use change.

6 Controlled Study

To evaluate Delta’s fit for making formulas interactive, we conducted two studies.³ In the first study (this section), we compared participants’ experience altering a formula augmentation with Delta and a baseline custom React component library. In the second study (Section 7), we evaluated the usability of a substantial subset of its language features after bounded training, with participants implementing formula calculators and walkthroughs from scratch.

In this first study, we sought to assess how rapidly participants could understand and edit an existing formula specification and whether Delta reduced the effort involved. The choice to have participants alter formulas was made to allow comparison to a baseline; any baselines we attempted required dozens or hundreds of lines of boilerplate to implement simple interactive formulas, so we reduced the complexity of the task until the participants could accomplish it with the baseline during a one-hour study session.

6.1 Methods

Participants. Delta is intended for authors of interactive math documents with proficiency in LaTeX and web development, a subset of the author population studied by Head et al. [15]. This study compared Delta with a custom React library, so eligibility required competency with LaTeX formulas, JavaScript, and basic React. We recruited 19 graduate and undergraduate students from a private university’s engineering school across computer science, data science, computer graphics, and robotics, screening for these competencies. Participants’ median comfort ratings on a 5-point Likert scale were 4 for LaTeX formulas and 4 for JavaScript; all had prior experience with React, ranging from small personal or academic projects to substantial professional use.⁴

Setup. Sessions took place remotely over Zoom and were recorded. Participants used their own computers. Tasks were run in GitHub Codespaces—an online IDE—pre-loaded with template code.

Tasks. The study was within-subjects: each participant completed the same formula editing task twice, once with each library (baseline and Delta). We chose to have participants complete the same task with both tools to enable direct comparison of implementation efficiency. Tool order was counterbalanced (9 used baseline first, 10 used Delta first) to mitigate learning effects. The challenge of the task primarily stemmed from identifying and using the correct language features in each library to achieve the desired functionality. Because language features varied by condition, we expected pronounced differences in performance even though each task required the same changes to the stimulus formula.

The task exercised essential aspects distinguishing Delta’s config-based structure from a baseline approach: selecting variables and defining interactions, labels, and computation. Participants received a starter implementation of a kinetic energy formula $K = \frac{1}{2}mv^2$, where K and v were interactive—dragging v updated the value of K —but m was static. They were asked to:

- make m draggable, with values ranging from 0 to 10 in increments of 1 and defaulting to 1;
- add a label displaying m ’s current value and description;
- ensure K updated correctly based on both m and v .

A reference video demonstrated the target behavior. Participants were given 20 minutes per task, a limit determined as being likely to allow all users to finish based on pilot testing. Instead of a demonstration or practice session, participants were given written API tutorial materials they could reference freely throughout. These materials were of similar length and quality across conditions.

Baseline. We compared Delta to a baseline implementation designed to resemble the best case of what a React implementation would look like that achieves Delta-style outputs.⁵ At an abstract level, the baseline is Delta without its integration of formulas, interactivity, and semantics into a coherent specification. The baseline exposed three React components specific to formulas: a formula renderer, invisible overlays for capturing variable interaction events, and variable labels, all used within a required formula container component. Succeeding with the baseline required seven steps: (1) creating state attributes for the target variable, (2) wrapping the variable’s LaTeX with `\cssId{}`, (3) placing an invisible overlay over the variable, (4) passing the ID to the variable’s label component, (5) adding the label, (6) configuring the variable’s bounds, default value, and step size, and (7) updating the computation function. These steps reflect work that would be required in any JavaScript-based setup. To the extent possible, we moved unnecessary complexity and boilerplate out of the React starter code into auxiliary files invisible to the participant; the constructs remaining included minimal logic for defining React state variables for the formula, a LaTeX string in which participants had to wrap variables in `\cssId{}` macros, and basic components with props for managing hover, drag, and display values. The resulting starter code was 61 lines in the baseline compared to 40 lines for Delta.

Measures. To assess success, we collected measures of speed and effort. We collected behavioral measures from screen recordings including completion time and instances of irrelevant edits, which indicated obstacles participants encountered when using Delta or the baseline. To measure completion time, we had participants tell us when they believed they were finished and then we checked their work; participants were asked to continue if they had not actually completed the task. Those who reached the 20-minute limit were stopped and their time recorded as 20 minutes. We also collected self-report measures of ease and other experience factors. On a 7-point Likert scale (1 = baseline, 4 = about the same, 7 = Delta), they rated which condition was (1) easier to complete the task with, (2) better suited for the task, and (3) a better match for how

³Supplemental material includes all study instruments: our screening questionnaire, starter code for all conditions and tasks, API tutorial documents, the reference video, task sheets, written tutorials, and post-task questionnaires.

⁴Further information about participants’ backgrounds is in the appendix.

⁵The supplemental materials provide the complete baseline repository, including its starter implementation (`src/App.tsx`) and completed solution (`src/solution.tsx`).

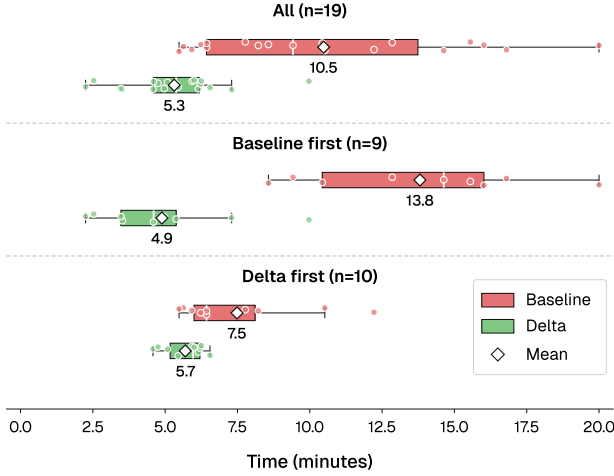


Figure 5: Task completion times by tool. Participants completed the task significantly faster with Delta than with the baseline, regardless of which tool they used first.

they would naturally describe the task. Participants also provided free-text explanations for each rating.

Analysis. Because our data are paired and not assumed to be normally distributed, we use the two-tailed Wilcoxon signed-rank test for all behavioral comparisons. For subjective Likert ratings, we test whether responses differ significantly from the neutral midpoint (4) using two approaches: a two-tailed sign test as our primary analysis, which considers only the direction of differences, and a two-tailed one-sample Wilcoxon signed-rank test that considers the ranked magnitudes of differences. We report Cohen’s d for paired samples as an estimate of effect size where applicable.

6.2 Results

Effect on time. Nearly all tasks (37 of 38, 97%) were completed within the 20-minute time limit (participant P30 did not complete the baseline task). As shown in Figure 5, participants completed the task significantly faster with Delta (Mdn = 5.4 min, $M = 5.3$ min, $SD = 1.8$ min) than with the baseline (Mdn = 9.4 min, $M = 10.5$ min, $SD = 4.4$ min); Wilcoxon $W = 6.0$, $p < .001$, Cohen’s $d = 1.54$. Of 19 participants, 16 were faster with Delta. The median speedup was $1.87\times$. This effect was robust to the ordering of the conditions ($p = .027$ for Delta first; $p = .004$ for baseline first), with a larger mean difference in task times when Delta was experienced second (more details in Appendix E.0.2).

Effect on effort. Participants rated Delta significantly above the neutral midpoint on all three scales: ease (Mdn = 7, $M = 6.42$; 18 of 19 above neutral, sign test $p < .001$; Wilcoxon $W = 1.5$, $p < .001$), suitability (Mdn = 6, $M = 5.84$; 15 of 17 non-tied above neutral, sign test $p = .002$; Wilcoxon $W = 9.0$, $p = .001$), and language match (Mdn = 7, $M = 5.53$; 14 of 17 non-tied above neutral, sign test $p = .013$; Wilcoxon $W = 26.5$, $p = .014$). Of the 19 participants, 18 rated Delta as easier (scores of 5–7); the only exception was P30, who wrote, “Library A felt easier to understand, especially making hover and drag behaviors easy to implement.” Of the 19 participants,

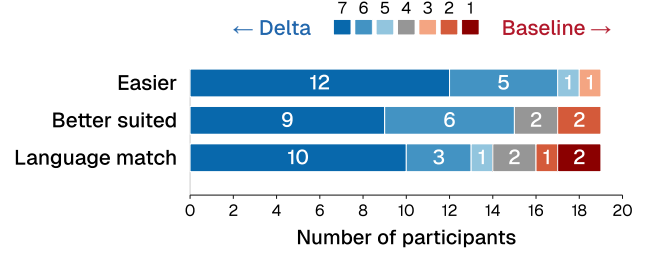


Figure 6: Post-task subjective ratings. Participants rated which library was easier, better suited, and a better language match on a 7-point scale (1 = baseline, 4 = about the same, 7 = Delta; $n = 19$). All 3 measures favored Delta significantly ($p \leq .014$).

6 (32%) attempted edits in irrelevant parts of the codebase with the baseline, compared to only 1 (5%) with Delta. A further analysis of what in the baseline caused friction, and of natural and unnatural parts of Delta, appears in Appendix E.0.3.

7 Observational Study

In our second study, we examined Delta’s usability after a bounded training session by asking participants to build two kinds of augmentations where they had freedom in the precise outcome. We sought to understand (1) whether participants could successfully apply the Delta language features introduced in the tutorials and (2) the appropriateness of those language features.

7.1 Methods

Participants. Of the 19 participants from the controlled study (Section 6), 10 took part in the observational study. We sampled in this way for convenience; we already knew these participants could pass our inclusion criteria. A consequence of this choice is that participants entered the observational study with some prior exposure to Delta. In aggregate, participant backgrounds were very similar to those in the prior study: participants’ median comfort ratings on a 5-point Likert scale were 4 for writing LaTeX formulas ($\sigma = 1.03$) and 3.5 for JavaScript ($\sigma = 1.43$). All had some prior experience with React. Below, we refer to participants by anonymized identifiers assigned to them during the study (e.g., P21, P48).

Setup. As in the controlled study, these study sessions were held on Zoom with coding taking place in a GitHub Codespace.

Tasks. Each participant completed two open-ended formula augmentation tasks, each lasting up to 30 minutes. In each task, participants were given an empty Delta config as a starting point, with placeholders for formulas, variables, and semantics. They were given a task sheet that provided the formula’s LaTeX, background about the formula, a real-world example with specific numerical values, and suggestions that could motivate potential designs. Participants were instructed to augment the formula in whatever way they thought would best help a reader understand the underlying concept; the task sheets stated “there is no single right answer” and encouraged participants to try alternatives.

Tasks were chosen to test essential parts of the Delta language. Task 1 (T1) assessed the basic features for enabling calculation and

making variables interactive, while Task 2 (T2) evaluated the walk-through constructs. Graphing and SVG constructs were excluded to keep session length feasible (already 2 hours); since they would have depended on features from T1 and T2, we prioritized those features instead. T1 required participants to augment a formula for radioactive decay, $N(t) = N_0 \cdot e^{-\lambda t}$; T2 required them to create a walkthrough for an expected-value formula, $E = \sum_{x \in X} x P(x)$, that included at least one step. Prior to each task, participants followed a self-paced written tutorial introducing all language features they might need for the task on an example formula. They were given 20 minutes to complete each tutorial.

Duration. Each task lasted up to 30 minutes, after which the facilitator stopped the participant. Although tasks were open-ended with no fixed completion criteria, all 10 participants produced a working interactive formula with correct computation and semantics and had time to iterate on its design. Including tutorials, questionnaires, and transitions, sessions lasted approximately 2 hours.

Measurement and Analysis. 9 of 10 sessions were facilitated by the same researcher; 1 by a second researcher following the same protocol. Participants were told to think aloud, and the facilitator could interrupt with questions. After each task, participants used Likert scales to rate the intuitiveness and difficulty of different kinds of edits to the Delta spec and elaborated on their ratings in free-text responses. Because each task introduced different features, questionnaires asked only about features relevant to that task. Each researcher reviewed recordings from the sessions they facilitated and compiled timestamped observation logs documenting usage patterns, errors, recovery strategies, and notable utterances. One researcher then performed qualitative coding across all data, identifying common workflows, design successes, and gaps.

7.2 Results

All 10 participants completed both tasks within the allotted 30 minutes. We first describe design successes and then gaps. Design successes are indexed by relevant dimensions from the Cognitive Dimensions of Notations framework [4], which provides a vocabulary for different aspects of usability in notations like the Delta DSL. We foreground two dimensions that were most important to us in designing the DSL: *closeness of mapping* (how naturally the notation maps to the problem domain) and *viscosity* (the effort involved in making small changes).

7.2.1 Closeness of mapping. Per-feature intuitiveness ratings were our proxy for closeness of mapping; distributions of ratings appear in Figure 7. Difficulty followed similar distributions and is shown in the appendix (Figure 13). For nearly all features, a majority of participants selected either “very intuitive” or “somewhat intuitive,” with basic features (from T1) appearing particularly intuitive. Below, we name particular features that were sites of interesting behavior and participant commentary.

Variables as IDs. Our language uses LaTeX variable strings as indexes into the formula and calculation state. This seemed to work well, with 6 participants describing it as “very intuitive” and 4 as “somewhat intuitive.” Participants named variable IDs capably; 6

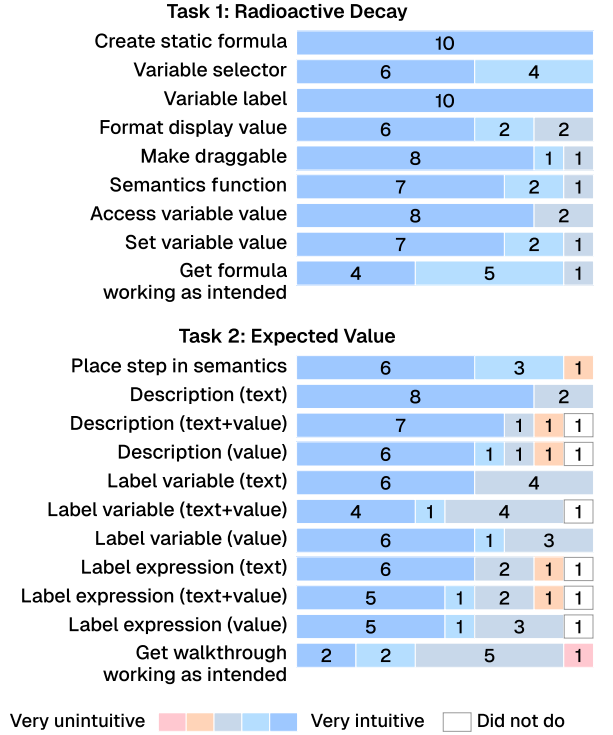


Figure 7: Self-reported intuitiveness ratings. Participants rated intuitiveness of Delta language features on a 5-point Likert scale with a “did not do” option. Task 1 included 8 language features, and Task 2 10 features. Each task also included one item assessing the overall authoring workflow.

were able to reference complex variables like “ $N(t)$ ” and “ λ ” correctly on their first try, and 7 were able to select multi-symbol expressions in walkthrough steps correctly on their first try.

Computation. The semantics function—plain JavaScript—was well received: accessing variable values within it was rated “very intuitive” by 8 of 10 participants, and expressing computation by 7 of 10. Participants incorporated standard JavaScript control flow into walkthroughs: 8 of 10 used the for-loop pattern introduced in the tutorial to generate repeated summation steps, with P57 articulating the rationale aloud: “Oh, I have to use a for loop—they all need to do the same thing.” P45 also used an if statement to vary walkthrough content across cases. Referring to adding step calls within the semantics function, P48 noted that “placing steps [is] pretty intuitive because [it] follows how your brain thinks about math expressions,” while P57 observed that “once the template of the formula [semantics] was set up, writing the steps was fast.”

Walkthrough labels. Delta’s formatting utilities for walkthrough labels—`sigfigs()`, `precision()`, and `latex()`—were adopted successfully on first use as 7 participants refined the presentation of label values. 3 participants combined `latex()` with `sigfigs()` or `precision()` to format step labels on their first attempts.

One further indicator of the features’ fit was that participants reached for them driven by higher-level goals about how best to

present the formula. Several participants (P11, P17, P19, P29, P48) paused to reason about pedagogical effectiveness. In T1, P48 decided to make the time variable interactive because “some questions relate to how much time has passed.” P19 chose not to make λ draggable, reasoning “lambda is a constant.” P11 reasoned about physical constraints: “Time should never be negative. Let’s have a range for that.” P29 said, “For the first step I can define a set of possible values X and on the second step I can decompose the sum.”

7.2.2 Viscosity. Delta was designed to let authors build up formulas one bit at a time, beginning with a static formula and incrementally adding and refining augmentations. We saw participants take advantage of this capability. 8 of 10 participants regularly alternated between editing code and checking rendered output. 4 reviewed a static formula before declaring any variables; 3 initialized empty variable declarations and verified their correctness by hovering over each variable’s location in the formula and confirming that it was highlighted; and 3 assigned and rendered labels before doubling back to define semantics and interactivity. Across both tasks, formatting of labels tended to follow a two-phase pattern: 8 of 10 participants first focused on getting computation correct—defining variables, writing semantics, and verifying values—before returning to adjust presentation with significant figures, precision, or LaTeX rendering. 5 participants demonstrated or expressed that the language helped them implement localized edits, modifying one aspect of the specification without ripple effects across others. Participants largely followed the default configuration structure—first providing LaTeX, then variables, then semantics (8 of 10 participants); the alternative was writing out the semantics first (P19, P57).

7.2.3 Gaps. We saw two recurring gaps. The first was trouble selecting the right occurrence of a variable when it appeared multiple times in a formula (5 participants). We initially deferred occurrence-specific selection because a known solution is already published in FFL [40]. We have since implemented FFL-inspired occurrence selection with optional filter and exclude rules. The second gap was that stepping requires setting a modal flag. 3 participants got stuck when they forgot to set the flag. One solution could be to enable stepping by default as soon as steps are added to semantics.

8 Discussion

Our first study showed greater speed and ease with Delta than a React baseline. In the second study, after bounded training, participants produced formula calculators and walkthroughs from scratch, largely reporting language features as somewhat or very intuitive. Participants successfully used features for referencing variables, defining semantics and steps, and configuring interactivity. These findings suggest the fit of Delta’s basic design for basic tasks.

8.1 Study limitations

Our study involved participants who were not experienced math document authors; studies with experienced authors may reveal additional gaps in Delta. Tasks were relatively short (maxing out at 30 minutes) and the possibilities of exploration were limited by the equations provided. Further studies of unconstrained use are necessary to determine if the language can support the formulas and goals that authors bring to the table.

8.2 Future Work

The Delta language and ideas from its development open up several avenues for interesting future research.

Interactions around complex expressions. How should a DSL expose mechanisms for readers to interactively modify complex expressions like vectors, matrices, or indexed sets? Currently, Delta requires authors to augment member elements (matrix entries, set members) individually. An alternative could be to let authors augment full structures at once (vectors, matrices). An interesting direction to explore is defining input control widgets for the various complex data types. In some places, it may make sense to provide compositional control widgets. For instance, a vector of strings might allow a reader to change the list of elements and expose widgets for modifying individual member strings.

Symbolic expansion and derivation. Delta is optimized for calculations involving *concrete* values. However, manipulation of formulas happens in contexts without concrete values. For instance, formulas are symbolically manipulated to derive new formulas. They are also expanded into finer terms, like when one lists out the terms of a Taylor series. It could be useful to allow authors to support interactive expansion and derivation. We conjecture that this might require integration of Delta-like DSLs with engines for symbolic manipulation (e.g., sympy), and constructs for pointing out what to expand and enabling recursive expansion of terms.

Role in agentic programming. The value of Delta is murkier if agents can generate the boilerplate it was designed to replace. However, our preliminary experiments suggest Delta may have a role to play in agentic coding workflows. In a comparison on a handful of tasks, an agent completed tasks more reliably and with less code using Delta than using a React baseline (see Appendix G). Future work should deepen investigations like these to sharpen the value proposition of Delta-like DSLs in agent workflows.

Broadening access. Delta might eventually help non-programmer authors as well if used as a target output format for AI-assisted document authoring and graphical editing tools.

9 Conclusion

This paper presents Delta, a TypeScript-embedded DSL for adding interactivity to typeset formulas in web-based math explanations. In our studies, Delta reduced the time and effort of editing a formula augmentation, and participants were able to use it to devise their own augmentations from scratch after a brief tutorial. Language features were largely described as intuitive. Our own explorations have shown Delta capable of producing rich augmented formulas with in-situ calculation, walkthrough explanations, and linked visualizations. Altogether, this paper develops an important kind of tool for realizing the expressive potential of interactive science texts.

Acknowledgments

This research was supported by funding from the National Science Foundation under grant number #2541577.

References

- [1] Shaaron Ainsworth. 2006. DeFT: A conceptual framework for considering learning with multiple representations. *Learn. Instr.* 16, 3 (June 2006), 183–198.
- [2] airladon. 2018. FigureOne: Draw, animate and interact with shapes, text, plots and equations. GitHub repository. Retrieved March 26, 2026 from <https://github.com/airladon/FigureOne>
- [3] airladon. 2020. Traveling Wave 02 – Sine Waves. FigureOne Examples. <https://airladon.github.io/FigureOne/examples/Traveling%20Wave%2002%20-%20Sine%20Waves/> Source: fig2.js, fig3.js.
- [4] Alan Blackwell and Thomas Green. 2003. Notational systems—the cognitive dimensions of notations framework. In *HCI Models, Theories, and Frameworks*. Elsevier, 103–133.
- [5] Carrie J Cai and Philip J Guo. 2019. Software developers learning machine learning: Motivations, hurdles, and desires. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE.
- [6] David P. Carlisle. 1997. The color package. Comprehensive TeX Archive Network (CTAN). <https://ctan.org/pkg/color>
- [7] Matthew Conlen and Jeffrey Heer. 2018. Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 977–989.
- [8] Will Crichton. 2021. A New Medium for Communicating Research on Programming Languages. In *Proceedings of the 3rd Workshop on Human Aspects of Types and Reasoning Assistants (HATRA) at SPLASH*.
- [9] Distill Editorial Team. 2017. Distill: A Journal for Machine Learning. Web journal. Retrieved March 26, 2026 from <https://distill.pub>
- [10] Emily Eisenberg and Sophie Alpert. *KaTeX*. Retrieved September 16, 2020 from <https://katex.org>
- [11] Brian Faulkner, Katherine Earl, and Geoffrey Herman. 2019. Mathematical maturity for engineering students. *International Journal of Research in Undergraduate Mathematics Education* 5, 1 (2019), 97–128.
- [12] Camille Gobert and Michel Beaudouin-Lafon. 2022. i-LaTeX: Manipulating Transitional Representations between LaTeX Code and Generated Documents. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, Paper 554.
- [13] Gabriel Goh. 2017. Why Momentum Really Works. *Distill* (2017). <https://doi.org/10.23915/distill.00006> Source: eigensum.js.
- [14] Devamardeep Hayatpur, Brian Hempel, Richard Lin, and Haijun Xia. 2025. The Shapes of Abstraction in Data Structure Diagrams. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. Article 883.
- [15] Andrew Head, Amber Xie, and Marti A Hearst. 2022. Math augmentation: How authors enhance the readability of formulas using novel visual design practices. In *CHI Conference on Human Factors in Computing Systems*. ACM, Article 491.
- [16] Jeffrey Heer, Matthew Conlen, Vishal Devireddy, Tu Nguyen, and Joshua Horowitz. 2023. Living Papers: A language toolkit for augmented scholarly communication. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. ACM, Article 42.
- [17] Andrea Kohlhasse, Michael Kohlhasse, and Taweechai Ouypornkochagorn. 2018. Discourse phenomena in mathematical documents. In *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 147–163.
- [18] David Landy and Robert L Goldstone. 2007. Formal notations are diagrams: evidence from a production task. *Mem. Cognit.* 35, 8 (Dec. 2007), 2033–2040.
- [19] Martin Leung, Renae Low, and John Sweller. 1997. Learning from equations or words. *Instructional Science* 25, 1 (1997), 37–70.
- [20] Yong Li, Shoaib Kamil, Alec Jacobson, and Yotam Gingold. 2022. H $\heartsuit$ rtDown: Document Processor for Executable Linear Algebra Papers. In *SIGGRAPH Asia 2022 Conference Papers*. Article 51.
- [21] Geoffrey Litt, Paul Shen, Max Schoenig, and Paul Sonnentag. 2022. Potluck: Dynamic Documents as Personal Software. Research project and essay. Retrieved March 26, 2026 from <https://www.inkandswitch.com/potluck/>
- [22] Lars Madsen, Will Robertson, and Joseph Wright. 2014. The mathtools package. Comprehensive TeX Archive Network (CTAN). <https://ctan.org/pkg/mathtools>
- [23] Dan Margalit and Joseph Rabinoff. 2019. *Interactive Linear Algebra*. Georgia Institute of Technology. <https://textbooks.math.gatech.edu/ila> Open-source interactive textbook.
- [24] Microsoft. 2025. Equation Editor. Microsoft Support article.
- [25] Sheshera Mysore, Mahmood Jasim, Haoru Song, Sarah Akbar, Andre Kenneth Chase Randall, and Narges Mahyar. 2023. How Data Scientists Review the Scholarly Literature. In *Proceedings of the Conference on Human Information Interaction and Retrieval*. ACM, 137–152.
- [26] Bui Duc Nha. Retrieved July 7, 2023. *Mathcha*. <https://www.mathcha.io/>
- [27] Magnus Österholm. 2006. Characterizing reading comprehension of mathematical texts. *Educ. Stud. Math.* 63, 3 (Oct. 2006), 325–346.
- [28] Jan L Plass, Bruce D Homer, and Elizabeth O Hayward. 2009. Design factors for educationally effective animations and simulations. *J. Comput. High. Educ.* 21, 1 (April 2009), 31–61.
- [29] Josh Pollock, Catherine Mei, Grace Huang, Elliot Evans, Daniel Jackson, and Arvind Satyanarayan. 2024. Bluefish: Composing diagrams with declarative relations. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. Article 23.
- [30] Juan Carlos Ponce Campuzano. 2019. *Complex Analysis: A Visual and Interactive Introduction*. Juan Carlos Ponce Campuzano. <https://complex-analysis.com/>
- [31] Victor Powell and Lewis Lehe. 2015. Eigenvectors and Eigenvalues. Setosa.io Explained Visually. <https://setosa.io/ev/eigenvectors-and-eigenvalues/> Source: script.js.
- [32] Victor Powell and Lewis Lehe. 2015. Ordinary Least Squares Regression. Setosa.io Explained Visually. <https://setosa.io/ev/ordinary-least-squares-regression/> Source: SLRParameters.react.js.
- [33] Grant Sanderson. 2015. Manim: A Mathematical Animation Framework. GitHub repository. <https://github.com/3b1b/manim>
- [34] ST John et al. Annotate equations in LaTeX using TikZ. GitHub repository. Retrieved March 31, 2026 from <https://github.com/st-/annotate-equations>
- [35] Jeffrey Tao, Litao Yan, Jessica Shi, Mia Ginsberg, and Andrew Head. 2025. FreeForm: Flexibly Augmenting Formulas with Synchronized Markup and Graphical Edits. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. Article 578.
- [36] Griffin Teller and Joshua Sunshine. 2025. Bloom: Simplifying interactive diagram development with optimization constraints. In *PLATEAU Workshop on the intersection of HCI and PL*.
- [37] Paul Topping. 1999. Using MathType to Create T ϵ X and MathML Equations. In *TUGboat, Volume 20, No. 3 — Proceedings of the 1999 T ϵ X Users Group Annual Meeting*. T ϵ X Users Group, 184–188.
- [38] Bret Victor. 2011. Explorable Explanations. Retrieved March 26, 2026 from <https://worrydream.com/ExplorableExplanations/>
- [39] Bret Victor. 2011. Tangle: A JavaScript Library for Reactive Documents. Web page. Retrieved March 26, 2026 from <https://worrydream.com/Tangle/>
- [40] Zhiyuan Wu, Jiening Li, Kevin Ma, Hita Kambhamettu, and Andrew Head. 2023. FFL: A language and live runtime for styling and labeling typeset math formulas. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. ACM, Article 90.
- [41] xnought. 2020. Backprop Explainer. GitHub Pages. <https://xnought.github.io/backprop-explainer/> Source: AutoTool.js.
- [42] Katherine Ye, Wode Ni, Max Krieger, Dor Ma’ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: from mathematical notation to beautiful diagrams. *ACM Transactions on Graphics* 39, Article 144 (2020).
- [43] Hsin I. Yung and Fred Paas. 2015. Effects of Computer-Based Visual Representation on Mathematics Learning and Cognitive Load. *Journal of Educational Technology & Society* 18, 4 (2015), 70–77.

A Delta Config Grammar

```

<Config> ::= { formulas : <Formulas>, variables : <Vars>, semantics : <Sem>,
               [graph2d : <G2D>*, [graph3d : <G3D>*],
               [stepping : bool], }

<Formulas> ::= [<Formula>]*      <Formula> ::= { id : str, latex : str }

<Vars> ::= { (sel : str | <VarSpec>)* }
<VarSpec> ::= { [name : str], [default : num], [input : "drag" | "inline"],
                 [range : [min, max]],
                 [step : num], [precision : num], [sigFigs : num],
                 [latexDisplay : <Disp>], [labelDisplay : <Disp>],
                 [svgContent : <SVGFn>], [svgSize : {w, h}] }

<Disp> ::= "name" | "value" | "svg" | "none"

<Sem> ::= function ({vars, sample, step, latex}) <Body>
<Body> ::= { <Stmt>* }
<Stmt> ::= vars.id = <Expr> | sample(id, {x, y, [z]})
           | step(<StepSpec>[, stepId : str]) | <JSStmt>
<StepSpec> ::= { [description : str], [labels : {(sel : val)*}] }

<G2D> ::= { id : str, [xAxisLabel], [yAxisLabel], [xRange], [yRange],
             [lines : <Line2D>*, [points : <Pt2D>*], [vectors : <Vec>*] }

<Line2D> ::= { sampleId : str, parameter : sel, [range], [samples], [interaction : <Drag>] }
<Pt2D> ::= { sampleId : str, [interaction : <Drag>], [stepId], [persistence] }
<Vec> ::= { startSampleId : str, endSampleId : str, [label],
            [interaction : [xVar, yVar]], [curved], [stepId], [persistence] }
<Drag> ::= ["horizontal-drag" | "vertical-drag", sel]

<G3D> ::= { id : str, [xRange], [yRange], [zRange],
             [surfaces : <Surf3D>*, [lines : <Line3D>*, [points : <Pt3D>*] ] }
<Surf3D> ::= { sampleId : str, parameters : [sel1, sel2], [ranges], [samples], }
<Line3D> ::= { sampleId : str, parameter : sel, [range], [samples] }
<Pt3D> ::= { sampleId : str }

```

Figure 8: Grammar for Delta config. Styling properties (e.g., color, fontSize, labelFontSize, lineWidth, opacity) and the React component library are omitted. A *sel* (selector) is a literal LaTeX string or expression (e.g., "v", "N(t)", "\mu") identifying a variable. Abbreviations: *str*=string, *num*=number, *bool*=boolean. *<Expr>* and *<JSStmt>* are JavaScript expressions/statements. The *latex* helper produces string-valued expressions, such as *latex(n).precision(d)* and *latex(n).sigfigs(d)*, for use in step descriptions and labels. *<SVGFn>* receives the current variable's value and all other variables (environment) from the runtime, returning an SVG string for dynamic variable icons. Square brackets enclose arrays or, when surrounding an entire property, indicate that the property is optional.

B Patterns in code from formative analysis

Article	Source	F1	F2	F3	F4	F5
Backprop Explainer	Individual developer	•	•	•	•	•
Traveling Wave – Sine Waves	FigureOne	•	◦	•	•	•
OLS Regression	Setosa.io	•	•	•	•	•
Eigenvectors & Eigenvalues	Setosa.io	•	•	•	•	•
Why Momentum Really Works	Distill.pub	•	•	•	•	•
Understanding GNNs	Distill.pub	•	•	•	•	•
ILA: Least Squares	Georgia Tech	•	•	•	•	•
ILA: Systems of Equations	Georgia Tech	•	•	•	•	•
ILA: Vectors	Georgia Tech	•	•	•	•	•
Roots of Complex Numbers	complex-analysis.com	•	◦	◦	◦	•
The Complex Power Function	complex-analysis.com	•	•	•	•	•
Total		11/11	9/11	10/11	10/11	11/11

Table 1: Formative analysis thematic coding results. Each cell indicates whether the pattern was observed (•) or not (◦) in the article’s source repository. Codes for frictions: **F1** Diffuseness and tangling, **F2** Indirect identifiers, **F3** Fussing the DOM and LaTeX, **F4** Repetition, **F5** State management. See Section 3 for definitions.

C Examples from formative analysis

fig2.js: 123–139 (amplitude control)

```
let offset = 0.5;
mover.notifications.add('setTransform', () => {
  offset += mover.getPosition().y * 2;
  mover.transform.updateTranslation([0, 0]);
  if (offset > 1) { offset = 1; }
  if (offset < -1) { offset = -1; }
  const newA = offset * 2;
  trace.update(sine(newA));
  eqn.updateElementText({
    A: `${Math.abs(newA).toFixed(1)}`,
    sign
  });
});
```

fig3.js: 129–142 (wavelength control)

```
let offset = 0;
mover.notifications.add('setTransform', () => {
  offset += mover.getPosition().x;
  mover.transform.updateTranslation([0, 0.15]);
  if (offset > 1) { offset = 1; }
  if (offset < -1) { offset = -1; }
  const newR = offset * 3 + 5;
  trace.update(sine(newR));
  eqn.updateElementText({
    value: `${newR.toFixed(1)}`
  });
});
```

Figure 9: Schema duplication in the Traveling Wave article [3]. Two figures implement the same interaction schema—a draggable parameter controlling a sine wave—with structurally identical code. Both blocks maintain local `offset` state, accumulate drag input, clamp to bounds, transform to a parameter value, update the trace, and refresh the equation label. The only differences are which axis is read (`.y` vs `.x`), which parameter is computed (`newA` vs `newR`), and which label field is updated. This pattern, where the same mathematical interaction schema is copy-pasted and lightly edited per instance, appeared in 10 of 11 audited articles.

OLS Regression: DOM geometry for highlight positioning

```
// After render, measure tspan elements
var svgBB = this.getDOMNode().getBoundingClientRect();
var beta1Text = this.refs.beta1Text.getDOMNode();
var beta1TextBB = beta1Text.getClientRects()[0];
var beta2Text = this.refs.beta2Text.getDOMNode();
var beta2TextBB = beta2Text.getClientRects()[0];

// Manually reposition highlight overlays
d3.select(this.refs.beta1Highlight.getDOMNode())
  .attr('transform',
    `translate(${highlight1Pos.x}, ${highlight1Pos.y})`);
d3.select(this.refs.beta2Highlight.getDOMNode())
  .attr('transform',
    `translate(${highlight2Pos.x}, ${highlight2Pos.y})`);
```

Why Momentum Really Works: HTML string replacement

```
// Cache rendered MathJax in hidden DOM
function MathCache(id) {
  return document.querySelector(
    "#math-cache ." + id).innerHTML;
}

// String-replace inside rendered formula HTML
function renderEquation(hat, value, number) {
  var html = hat ? MathCache("phat")
    : MathCache("p");
  html = html.replace(
    '<span class="mord mathrm">0</span>', value);
  html = html.replace(
    '<span class="mord mathrm mtight">1</span>', number);
  return html;
}
```

Figure 10: Fussing the DOM and LaTeX (F3). Left: the OLS Regression article [32] measures rendered `tspan` bounding boxes after each React render, then manually repositions SVG highlight groups with `translate()` transforms. Right: the Momentum article [13] caches rendered MathJax fragments in a hidden DOM element, retrieves them via `querySelector`, and performs string replacement on the internal `<span>` markup to swap in dynamic values. Both patterns require authors to understand renderer internals and coordinate between mathematical intent and low-level DOM structure.

Backprop Explainer: deeply nested state object

```
// Bespoke nested state for regression
this.state = {
  yhat: [],
  playButton: true,
  linreg: {
    data: { X: [], y: [] },
    hyperparams: {
      learningRate: 0.03,
      epochs: 0,
      loss: null,
      speed: 200
    },
    tunableparams: { m: 0, b: 0 }
  }
};

// Manual event wiring to state transitions
<Fab onClick={async () => {
  await this.setState({
    playButton: !this.state.playButton });
  await this.a();
}}>
```

Eigenvectors: manual derived-value propagation

```
// Mutable state object
var opt = $scope.opt = {};
opt.pos0 = [2, 3];
opt.basis1 = [1, 0.5];
opt.basis2 = [0.5, 1];
opt.eigenVectors = [[1, 0], [0, 1]];
opt.eigenValues = [0, 0];

// Deep watch triggers manual recomputation
$scope.$watch('opt', function() {
  derived($scope)
}, true);

// Low-level drag wiring
nobs.call(d3.behavior.drag()
  .on('drag', function(d) {
    scope.$apply(function() {
      d.set(scope, d3.mouse(svg.node()))
    })
  }));
```

Figure 11: State management (F5). Left: the Backprop Explainer [41] maintains a deeply nested state object mixing UI flags (playButton), data arrays, hyperparameters, and tunable parameters, then wires click handlers to async state transitions. Right: the Eigenvectors article [31] stores mutable state in an Angular scope object, uses a deep \$watch to trigger manual recomputation of derived quantities, and wires D3 drag events through framework-specific \$apply calls. Both patterns require authors to manually orchestrate state flow, derived-value updates, and framework integration.

D Example Delta config

```

1  const config = {
2    formulas: [
3      { id: "loss-function", latex: "L = (y - w_t \\\cdot x)^2" },
4      { id: "update-weight", latex: "w_{t+1} = w_t - \\\alpha \\\cdot (-2x(y - w_t \\\cdot x))" },
5    ],
6    variables: {
7      L: { name: "Loss" },
8      w_t: { default: 0.5, name: "Current Weight" },
9      "w_{t+1}": { default: 0.5, name: "Next Weight" },
10     "\\\alpha": { default: 0.1, input: "drag", range: [0, 1], name: "Learning Rate" },
11     x: { default: 1.5, input: "drag", range: [0, 4], name: "Feature" },
12     y: { default: 3.0, input: "drag", range: [0, 4], name: "Label" },
13   },
14   stepping: true,
15   semantics: function ({ vars, sample, step, latex }) {
16     var w = vars.w_t;
17     for (var t = 0; t < 6; t++) {
18       var error = vars.y - w * vars.x;
19       var loss = error * error;
20       var gradient = -2 * vars.x * error;
21       sample("initial-loss", { x: w, y: loss });
22       step({
23         "loss-function": {
24           labels: {
25             "L": "Loss is " + latex(loss).precision(2),
26             "y - w_t \\\cdot x": "Error is " + latex(error).precision(2),
27           },
28         },
29         "update-weight": {
30           labels: {
31             "w_t": w,
32             "\\\alpha": vars["\\alpha"],
33             "w_{t+1}": w - vars["\\alpha"] * gradient,
34             "(-2x(y - w_t \\\cdot x))": "Gradient is " + latex(gradient).precision(2),
35           },
36         },
37       }, "1");
38       w -= vars["\\alpha"] * gradient;
39       vars.L = (vars.y - w * vars.x) * (vars.y - w * vars.x);
40       sample("final-loss", { x: w, y: vars.L });
41     }
42     vars["w_{t+1}"] = w;
43   },
44   graph2d: [
45     {
46       id: "gradient-descent",
47       xAxisVar: "w_t", yAxisVar: "L",
48       yRange: [-5, 10], xRange: [0, 4],
49       lines: [{
50         sampleId: "initial-loss",
51         parameter: "w_t",
52         interaction: ["horizontal-drag", "y"],
53       }],
54       points: [{
55         sampleId: "initial-loss",
56         stepId: "1", persistence: true,
57       }],
58       vectors: [{
59         startSampleId: "initial-loss", endSampleId: "final-loss",
60         stepId: "1", persistence: true,
61         curved: -1,
62       }],
63     },
64   ],
65 };

```

Figure 12: Delta configuration for gradient descent walkthrough (Figure 4). The specification declares two formulas, five interactive variables, and a semantics function that loops through training iterations, emitting `step()` calls with labels keyed by formula id.

E Study 1 Details

E.0.1 Participant backgrounds. On a 5-point Likert scale, participants' median comfort rating for writing LaTeX formulas was 4 ($\sigma = 0.91$); 11 wrote LaTeX weekly, 5 monthly, and 3 less than monthly. On the same scale, their median comfort rating for writing JavaScript was 4 ($\sigma = 1.12$); 9 wrote JS weekly, 8 monthly, 1 daily, and 1 less than monthly. Regarding React experience, 6 had written substantial amounts of React in a professional context, 6 had written substantial amounts in personal or academic projects, and 7 had written small amounts in personal or academic projects.

E.0.2 Task time by order of interface assignment. We observed a significant order effect on the magnitude of the time difference between conditions (Mann-Whitney $U = 88.0$, $p < .001$). Participants who used the baseline first showed a larger advantage for Delta (mean difference = 8.9 min) than those who used Delta first (mean difference = 1.8 min). This suggests that task familiarity compressed completion times in the second condition regardless of library. Critically, even within the more conservative subgroup that used Delta *first*, Delta was still significantly faster (Wilcoxon $W = 6.0$, $p = .027$, $n = 10$), with 7 of 10 participants completing the task faster with Delta.

E.0.3 Analysis of participant free-text responses. Participants' free-text responses revealed qualitative differences in *where* each tool imposed cognitive load. We organize these findings using the Cognitive Dimensions of Notations framework [4]. Three dimensions emerged most clearly.

Viscosity. The most frequently cited baseline pain point was the effort required to make even a small change. The baseline was described by 7 participants (P45, P55, P79, P80, P87, P88, P99) as requiring coordinated edits across disconnected locations—state declarations, component props, and the LaTeX string (P55: “making the small change to make `m` modifiable with a drag motion required changes in several parts of the code”). The `\cssId{}` mechanism was a specific source of friction: 7 participants (P11, P14, P20, P79, P87, P88, P99) reported confusion or errors matching CSS identifiers across component props and LaTeX strings (e.g., P88: “I forgot to add the `cssId` on the LaTeX element ‘`m`’”).

Closeness of mapping. During the study, 4 participants (P11, P20, P79, P88) independently described Delta's structure—formulas, variables, and semantics—in terms that closely mirror the library's API, without being prompted. P11 explained: “there's the semantics of a formula, the looks of a formula [LaTeX], and then different components of the formula. [Delta] naturally gave in to this description.” That participants independently reconstructed the library's decomposition suggests it maps closely to how authors naturally reason about interactive formula authoring.

Visibility. Delta's configuration also improved *visibility*—the ease with which required parts of the notation can be identified. Of the 19 participants, 10 cited co-location as an advantage: P29 found it “way faster to just look at the code and immediately see all the moving parts at a glance”; P21 praised “clear partitions for variables, so could find and update easily”; and P80 described Delta as grouping “the variables/components (labels, dragging functionality, displayed value) together in an intuitive way.” By contrast, the

baseline scattered relevant parts across disconnected locations (P79: “a lot of moving components—in the html, use state, two types of wrappers”).

Among the participants, 2 (P14, P29) preferred the baseline on suitability and language dimensions despite completing the task faster with Delta, citing familiarity with React idioms—pointing to a tension between task-specific efficiency and integration with existing workflows.

F Study 2 Details

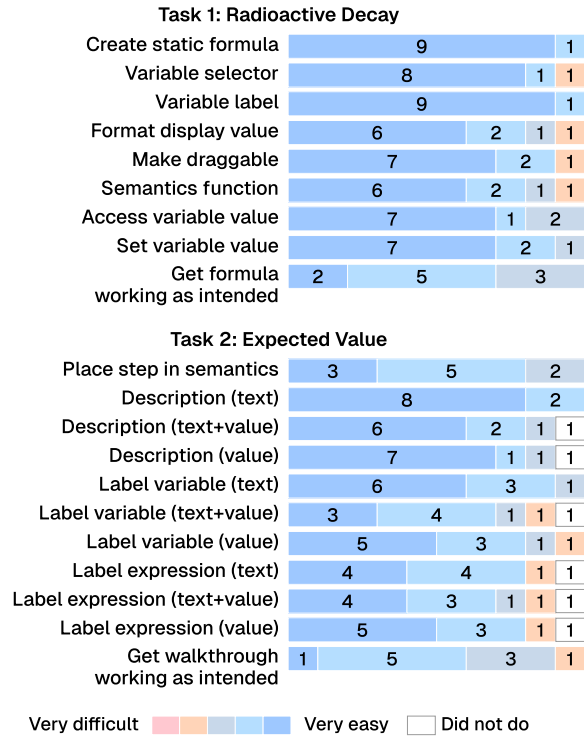


Figure 13: Self-reported difficulty ratings. Participants rated the difficulty of language features on a 5-point Likert scale ($n = 10$), with a separate “did not do” option. The distributions largely mirrored the intuitiveness ratings (Figure 7), with most actions rated as easy or very easy.

G Programming assistant probe

In a preliminary experiment to investigate how Delta might integrate with agentic coding tools, we asked Cursor Composer 2.5 to create interactive explanations for three formulas, and we tested how it performed on this task across two conditions. In the Delta condition, the agent was required to use the Delta DSL package and was directed to Delta’s API reference. In the React baseline, Delta was unavailable, while conventional notation and plotting libraries like `katex`, `better-react-mathjax`, `d3`, `plotly.js-dist`, and `mathjs` were available. Each task focused on a different capability supported by Delta. The kinetic-energy example tested basic labeling and interaction, the arithmetic-mean example tested walk-throughs, and the wave-equation example tested dynamic graphical labels and a linked plot.

For each formula task across the conditions, the agent received the same design instructions. All authored code had to remain in a

single implementation file and render without runtime errors. We measure LOC as the total number of physical lines in each source file; no filtering was applied. After each attempt, the study facilitator made a human judgment call about whether the predetermined rubric criteria were satisfied. The unsatisfied criteria were presented to the agent again in a templated prompt asking it to try to satisfy the listed criteria. Each formula creation task ended either when all criteria were satisfied or after the fifth attempt.

Results. See tables on the next page.

Limitations. This is a preliminary experiment. The probe used a single programming assistant and included one attempt sequence per task-condition pair across only three tasks. Criterion satisfaction was assessed by one facilitator. The probe does not measure reliability across repeated independent generations or evaluators.

Table 2: Results of the agentic coding probe. Attempts count the initial generation as Attempt 1. Each LOC sequence reports the recorded line count for successive attempts; the source files do not define the LOC-counting convention. Pass indicates that all task criteria were satisfied. Incomplete indicates that criteria remained unmet after five attempts.

Task (# of criteria)	Condition	Attempts	Outcome	Recorded LOC by attempt	Criterion remaining at stop
Wave equation (16)	Delta	1	Pass	183	—
	React baseline	5	Incomplete	378, 570, 606, 589, 614	t was not draggable within the required 0–10 bounds.
Arithmetic mean (12)	Delta	2	Pass	128, 121	—
	React baseline	2	Pass	397, 333	—
Kinetic energy (10)	Delta	1	Pass	51	—
	React baseline	5	Incomplete	275, 212, 210, 225, 254	The formula was not properly typeset: the fraction was not rendered as a built-up fraction, and the exponent was not rendered as a superscript.

Table 3: Attempt-level failure progression. Only unsuccessful attempts are shown. The final column summarizes the criteria recorded as unmet after each attempt.

Task	Condition	Attempt(s)	Criteria recorded as unmet
Wave equation	React baseline	1	Mathematical typesetting, interaction with t , and a curve marker.
		2	Interaction with t .
		3	Mathematical typesetting and interaction with t .
		4–5	Interaction with t .
Arithmetic mean	Delta	1	Visibility of all variable labels.
	React baseline	1	Mathematical typesetting.
Kinetic energy	React baseline	1	Mathematical typesetting, labels for m and v , and drag interactions for m and v .
		2–5	Mathematical typesetting.