H❤RTDOWN: Document Processor for Executable Linear Algebra Papers

YONG LI, George Mason University, USA
SHOAIB KAMIL, Adobe Research, USA
ALEC JACOBSON, University of Toronto and Adobe Research, Canada
YOTAM GINGOLD, George Mason University, USA

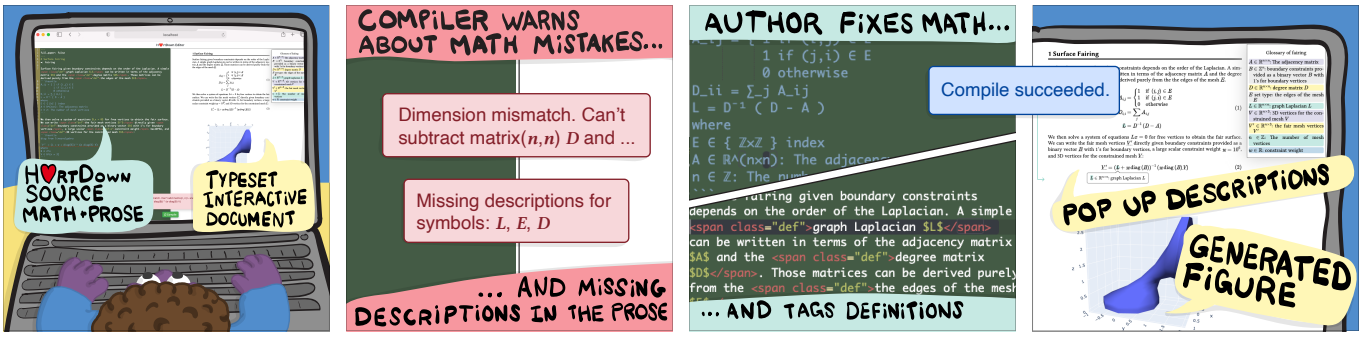

Fig. 1. An author writes a mathematical document in H❤RTDOWN. The document's source contains math written in I❤LA interspersed with prose. H❤RTDOWN compiles this into a typeset interactive document. Unlike LaTeX or other systems, the math is compiled. This means the author is alerted of static errors or if symbols are not described in the prose. The result is a rich document that allows readers to investigate math equations and interact with multimedia generated by running the compiled math.

Scientific documents describe a topic in a mix of prose and mathematical expressions. The prose refers to those expressions, which themselves must be encoded in, e.g., LATEX. The resulting documents are static, even though most documents are now read digitally. Moreover, formulas must be implemented or re-implemented separately in a programming language in order to create executable research artifacts. Literate environments allow executable code to be added *in addition* to the prose and math. The code is yet another encoding of the same mathematical expressions.

We introduce H❤RTDOWN, a document processor, authoring environment, and paper reading environment for scientific documents. Prose is written in Markdown, linear algebra formulas in an enhanced version of I❤LA, derivations in LATEX, and dynamic figures in Python. H❤RTDOWN is designed to support existing scientific writing practices: editing in plain text, using and defining symbols in prose-determined order, and context-dependent symbol re-use. H❤RTDOWN's authoring environment assists authors by identifying incorrect formulas and highlighting symbols not yet described in the prose. H❤RTDOWN outputs a dynamic paper reader with math augmentations to aid in comprehension, and code libraries for experimenting with the executable formulas. H❤RTDOWN supports dynamic figures generated by inline Python code. This enables a new approach to scientific experimentation, where editing the *mathematical formulas* directly updates the figures. We evaluate H❤RTDOWN with an expert study and by re-implementing *SIGGRAPH* papers.

CCS Concepts: • **Computing methodologies** → **Graphics systems and interfaces**; • **Software and its engineering** → **Domain specific languages**; • **Mathematics of computing** → *Mathematical software.*

SA '22 Conference Papers, December 6–9, 2022, Daegu, Republic of Korea

ACM ISBN 978-1-4503-9470-3/22/12.
https://doi.org/10.1145/3550469.3555395

Additional Key Words and Phrases: linear algebra, domain-specific language, compiler, Markdown, literate programming

ACM Reference Format:
Yong Li, Shoaib Kamil, Alec Jacobson, and Yotam Gingold. 2022. H❤RTDOWN: Document Processor for Executable Linear Algebra Papers. In *SIGGRAPH Asia 2022 Conference Papers (SA '22 Conference Papers), December 6–9, 2022, Daegu, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3550469.3555395

1 INTRODUCTION

Researching and disseminating scientific ideas relies on written communication. Authors interleave prose, mathematical expressions, and figures when writing on a chalkboard, an email, a blog post, or an article. For example, computer graphics research frequently uses linear algebra notation. *Reading* these static documents is held back by the need to keep track of a large number of symbol definitions, leading to various proposals for math augmentations [Head et al. 2022]. *Making use of* these documents, for trying out ideas, replicating the results, or follow-up experiments, is held back by the need to translate formulas to an executable programming language. Literate environments address the lack of executability, but require authors to manually *re-write* the mathematical formulas as executable code blocks [Knuth 1984]. Better reading environments for scientific papers have been proposed (e.g., ScholarPhi [Head et al. 2021]), but lack a canonical source for necessary metadata about mathematical symbols (connecting prose *about* a symbol to the symbol *itself*, and disambiguating symbol re-use in different contexts).

We introduce H❤️rtDown, an open-source document processor, authoring environment, and paper reader. H❤️rtDown supports reading with math augmentations in a dynamic reading environment. H❤️rtDown supports authoring by identifying incorrect math and undescribed symbols in an authoring environment. H❤️rtDown supports experimentation by automatically updating figures and generating executable code libraries whenever *formulas* are changed. H❤️rtDown documents are plain-text files based on Markdown [Gruber and Swartz 2004], I❤️LA [Li et al. 2021] for compilable formulas, LaTeX for derivations (non-executable math), and Python for dynamic figures. From a single plain-text source, H❤️rtDown outputs a dynamic paper reader, figures that automatically update as *formulas* change, and code libraries for accessing the formulas from all programming languages supported by I❤️LA.

We designed H❤️rtDown to support existing SIGGRAPH paper-writing practices (Section 3), based on a dataset of papers published at SIGGRAPH 2020. H❤️rtDown supports interleaving prose with formulas and derivations, out-of-order symbol definitions, symbol re-use in different contexts, and automatic matching of symbols used in I❤️LA formulas with those in LaTeX derivations. In doing so, we extended I❤️LA to support additional constructs, such as symbols with LaTeX formatting commands, local functions compatible with paper-writing conventions, conditional expressions, and *modules with parameters* as the unit of structure for executable code in papers. We evaluate H❤️rtDown with an expert study and by re-writing SIGGRAPH papers and replacing code in their implementations with our generated libraries.

2 RELATED WORK

Mathematical and scientific discourse relied heavily on prose and diagrams until the development of mathematical notation, which developed over the last 700 years into the one we know today [Cajori 1993; Wolfram 2000]. Contemporary scientific articles, such as those in computer graphics, are typically a mix of prose, figures, mathematical expressions, and sometimes pseudocode. The prose serializes the document and determines the order in which the non-textual elements (e.g., figures, expressions, and pseudocode) are presented. Literate programming environments were proposed by Knuth [1984] as a similar style of document to describe a computer program. The prose determines the order in which the pieces of the computer program are presented, rather than the order of execution required by the programming language. Knuth's original literate programming environment supported math in the prose, as the prose was compiled with TeX. Notebooks, such as Mathematica or Jupyter, can also be thought of as a form of literate programming [Arnon 1988; Kery et al. 2018; Rule et al. 2018; Wolfram 1988]. Authors write cells containing either prose and mathematical expressions or executable code. The executable code must appear in execution order. (Pluto [Plas 2020] and Observable [Bostock 2017] are notable exceptions.) Insofar as literate programming is modeled after mathematical papers, H❤️rtDown can also be thought of as a literate environment that automates the generation of executable code from formulas and supports prose-determined ordering of the mathematical expressions. Formulas can be directly evaluated when generating figures, eliminating the difficult and error-prone coding step.

Many approaches to reactive documents have been proposed (e.g. [Conlen and Heer 2018; Victor 2011]; see Conlen and Heer [2018] for a taxonomy). These approaches focus on creating "explorable explanations" as visualization-heavy interactive web documents. The underlying computation is hidden from readers. Some of these, such as Idyll [Conlen and Heer 2018], are also based on Markdown for ergonomic authoring. In contrast, H❤️rtDown is focused on helping users correctly author, read, and experiment with mathematical formulas in scientific documents.

The Distill journal [Team 2021] and Authorea [Goodman et al. 2017] focus on authoring and publishing scientific articles for the web with dynamic visualizations. Distill observed that the primary bottleneck is the effort in producing the content. Nota [Crichton 2021] is a document processor for scientific articles on the web that supports authoring dynamic reading functionality like symbol descriptions and visualizations. ScholarPhi [Head et al. 2021] proposed an improved reading interface for scientific paper PDFs, generating required metadata for math augmentations in a semi-automated manner from LaTeX. Suggestions were generated using natural-language processing techniques and verified by hand. This approach does not distinguish between re-uses of the same symbol in different contexts. H❤️rtDown obtains metadata from authors directly and generates ScholarPhi-like dynamic reader environments.

Various math augmentations have been proposed to facilitate understanding mathematical notation in papers [Alcock and Wilkinson 2011; Dragunov and Herlocker 2003; Head et al. 2021, 2022]. In [Head et al. 2022]'s taxonomy, H❤️rtDown's dynamic reader environment generates background, connector, and label annotations. Penrose [Ye et al. 2020] is a language for automatically generating mathematical diagrams from notation.

Bonneel et al. [2020] evaluated the state of replicability of computer graphics research. They found that most papers do not provide code, and many papers that do provide code required modifications to run. They did not investigate reproducibility, whether the algorithms could be independently re-created. We are also motivated to improve the replicability and reproducibility of scientific articles. H❤️rtDown verifies the executability of mathematical formulas via the I❤️LA compiler and can visualize correctness with dynamic figures that execute the formulas.

2.1 Compilable Math, Overview, & Limitations of I❤️LA

Several programming languages allow users to write executable code using syntax close to hand-written math, such as Fortran, Fortress [Allen et al. 2005], Lean [de Moura et al. 2015], and Julia [Bezanson et al. 2017], each with different intended uses. For example, Fortress is designed for distributed computations, and Lean is a proof assistant. Obtaining LaTeX or other formatted text suitable for including in papers is possible from Fortress, Lean, and Julia.

Although we could have adopted Fortess or Julia as the compilable math language for H❤rtDown, we would have had to modify their large compiler codebases to support automatic code ordering, to output latex for more kinds of expressions, to output metadata for our dynamic reading environment, and to generate modules with the appropriate execution model. Moreover, they would have encumbered our evaluation using in-the-wild code not written in those languages.

Instead, we chose to build on top of the I❤LA programming language [Li et al. 2021]. I❤LA allows researchers to write a plain-text source that visually looks like chalkboard math and compiles to both executable targets (e.g., Python, C++, Matlab) and typesetting instructions (e.g., LaTeX). I❤LA variables are type checked and may not be redefined. The following I❤LA example from Li et al. [2021] computes the closest point q to a set of lines in 3D:

```
given
p_i ∈ ℝ³: points on lines
d_i ∈ ℝ³: unit directions along lines

P_i = ( I₃ - d_i d_iᵀ )
q = ( ∑_i P_i )⁻¹ ( ∑_i P_i p_i )
```

I❤LA only supports linear sequences of expressions. It does not support out-of-order expressions, math mixed with prose, local functions, modules, or variable scoping. Nonetheless it is an ideal candidate for the foundation of H❤rtDown. We extend I❤LA to achieve our goals.

3 FORMATIVE STUDY

Our goal is to support authoring, reading, and making use of correct and reproducible scientific documents with minimal authoring overhead. We wish to do this with a plain-text document format that generates both executable code (as motivated by Li et al. [2021]) and a dynamic paper reading environment with math augmentations [Head et al. 2022] and figures that update in response to changes in the math. We aim to achieve this without changing *what* authors put in their papers (prose, math, figures, tables) and with minimal changes to *how* they write. We refer to this minimal impact on authors' preferred paper-writing practices as **ecological compatibility**.

To inform our design, we thoroughly analyzed 156 papers from the *SIGGRAPH North America* 2020 Technical Papers program[1], collecting both quantitative and qualitative observations. Our major qualitative findings are that:

(I) Prose organizes the document. Mathematical expressions appear between paragraphs of prose or inline.
(II) Math symbols are often used before they are defined, as determined by the prose.
(III) Symbols may be re-used, but the different context is clear to the reader.
(IV) A symbol may appear in both derivations and executable formulas.
(V) Symbols and functions may be defined via conditional assignment, a simple form of control flow.
(VI) Functions make use of a variety of implied semantics for parameters and pre-computed symbols.

All 156 papers appear to be written using LaTeX, that is, as plain-text source organized around prose (I) resulting in typeset math and a static document (a PDF). Without any requirement of generating valid, compilable code, or metadata for dynamic document viewing, the symbol (II–V) and function (V–VI) management of these papers falls onto the author, who must strive to maintain clarity and correctness via the informal context of the document's prose.

Pseudocode is sometimes present in the papers themselves, but compilable code almost never is. No paper was written as a literate program. We don't want to change what authors put in their papers, so we do not consider literate programming.

Our goal is to formalize and assist an author in creating a document where symbols and functions are coherently managed and deterministically result in both a compilable code library, metadata affording dynamic reading environments with math augmentations, and figures generated directly from the formulas. In particular, the non-linear ordering of expressions via prose (I), use of contexts (III), and local functions (VI) immediately precludes an attempt to trivially write an entire paper as "one giant I❤LA expression."

Indeed, local functions are ubiquitous in these papers. Quantitatively, we manually observed of the 916 function definitions across the 156 papers:

96% use parentheses for parameters,
91% rely on implicit parameters,
17% interpret the function's subscript as parameters,
15% have seemingly unused parameters,
6% are defined via conditional assignment
4% use square brackets for parameters,
2% interpret the function's superscript as parameters,
2% interpret parameter superscripts as additional parameters.

Examples and our tabulated data can be seen in the supplemental material. Based on these findings, we extend the grammar and implementation of I❤LA to include support for local functions (Section 4.2.1).

4 DOCUMENT DESIGN

As a concrete example, consider the H❤rtDown document for a mesh smoothing algorithm in Figure 2. This involves an energy minimization involving two terms defined in the prose. We describe the process of writing this example using H❤rtDown and consider a hypothetical reader's experience experimenting with H❤rtDown's outputs. The H❤rtDown document source is shown in Figure 3. In the appendix (Figures 7–9), we show three other examples: k-means clustering (with experiments into data weights and k-medians); image convolution with various filters by editing the filter function; and surface fairing with various Laplacian powers.

4.1 Authoring

Just as in LaTeX or many other Markdown formats, the prose is written as plain text with occasional markup commands (I). Optional metadata at the top of the file specifies the document title, authors, and abstract (not shown). In H❤rtDown, only non-executable

[1] There were 163 total papers, but we could not access 7 PDFs at the time of our analysis.

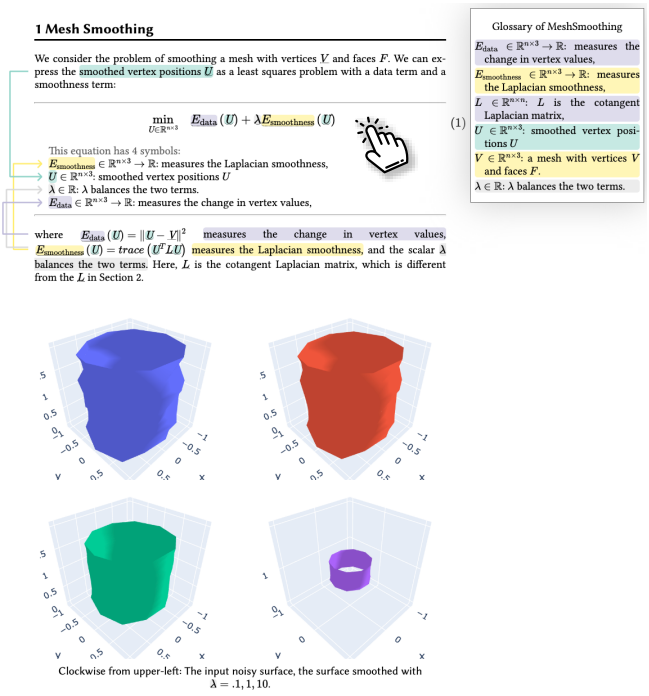

Fig. 2. The H♥RTDOWN reading environment for a mesh smoothing document. The reader has clicked on Equation 1 and is shown where the terms involved are defined. The source for this example can be seen in Figure 3.

mathematical expressions, e.g., derivations or symbol references, are written in LATEX format surrounded by $. For example,

```
We consider the problem of smoothing a mesh with vertices $V$
↪  and edges $S$.
```

Executable mathematical expressions are written in I♥LA format either in multi-line code blocks (surrounded by the Markdown convention ```) or inline surrounded by a pair of ♥'s. For example:

```
♥`E_\text{data}`(U) = ‖ U - V ‖² where U ∈ ℝ^(n×3)♥
```

Authors write I♥LA statements in any order (II). As I♥LA is a single-static assignment language, we modified the compiler to order I♥LA statements based on def-use analysis [Kennedy 1978].

The I♥LA compiler generates complete type information for all mathematical symbols that appear in the I♥LA code. This symbol list and type information are stored in a dictionary that the viewer uses to generate the glossary. H♥RTDOWN searches the LATEX-formatted mathematical expressions for the same symbols and automatically generates appropriate annotations for the viewer to locate appearances of symbols in prose and derivations (IV). H♥RTDOWN attempts to impose minimal overhead on authors and maintain ecological compatibility.

4.2 Overhead

H♥RTDOWN requires three kinds of additional effort from authors. First, one appearance of a symbol in the prose deserves special attention: the text *defining* the symbol. Detecting the span of this

```
# Mesh Smoothing
♥: MeshSmoothing

We consider the problem of smoothing <span class="def">a mesh with vertices $V$ and edges
$S$</span>. We can express the <span class="def">smoothed vertex positions $U$</span> as a least
squares problem with a data term and a smoothness term:

``` iheartla
trace from linearalgebra

min_( U ∈ ℝ^(n×3) ) `E_\text{data}`(U) + λ `E_\text{smoothness}`(U)
where
V ∈ ℝ^(n×3): The vertices of the mesh
L ∈ ℝ^(n×n) sparse: The Laplacian matrix
λ ∈ ℝ
```

where ♥`E_\text{data}`(U) = ‖ U - V ‖² where U ∈ ℝ^(n×3)♥ measures the change in vertex values, ♥`E_\text{smoothness}`(U)
= trace(Uᵀ L U) where U ∈ ℝ^(n×3)♥ measures the Laplacian
smoothness, and the scalar $λ$ balances the two terms. Here,
L is the cotangent Laplacian matrix, which is different from the
$\proselabel{ImageTools}{L}$ in Section 2.

<figure>
...
<figcaption>Clockwise from upper-left: The input noisy surface, the surface smoothed with $λ=.1,
1, 10$. </figcaption>
</figure>
```

Fig. 3. The H♥RTDOWN source for the mesh smoothing document in Figure 2. Source for the figure has been removed to save space and can be seen in the supplemental materials.

prose cannot be accurately automated, so we require authors to annotate such spans. In the example above:

```
We consider the problem of smoothing <span class="def">a mesh
↪  with vertices $V$ and edges $S$</span>.
```

The prose in this span tag is provided to reading environments as the definition of the symbols V and S. If symbols do not appear in the prose definition (as in the definition of $E_{\text{smoothness}}$), we require users to specify which symbol is being defined.

Second, I♥LA requires type declarations for all symbols not appearing on the left-hand side:

```
where
V ∈ ℝ^(n×3)
L ∈ ℝ^(n×n) sparse
λ ∈ ℝ
```

Third, authors must declare a context for their symbols:

```
♥: MeshSmoothing
```

Later context declarations override earlier declarations. The context disambiguates symbol reuse (III), e.g., "Here, L is the cotangent Laplacian matrix, which is different from the L in Section 2." and partitions the resulting executable code into modules. We provide a LATEX command (\proselabel) and syntax for our <span> tag to override the current context.

This extra author effort unlocks all the benefits of H♥RTDOWN (compilable code libraries, static analysis of paper math, and the enhanced reading environment).

4.2.1 Functions and Modules. Motivated by our formative study (Section 3), we extended I♥LA with a syntax for local functions (VI). In our example above,

```
`E_\text{smoothness}`(U) = trace( Uᵀ L U ) where U ∈ ℝ^(n×3)
```

Functions can make use of terms defined in the context, such as L in this example. Each context becomes an independent module in the resulting code library. In I♥LA output code, modules are implemented as structs or classes whose fields are all variables defined

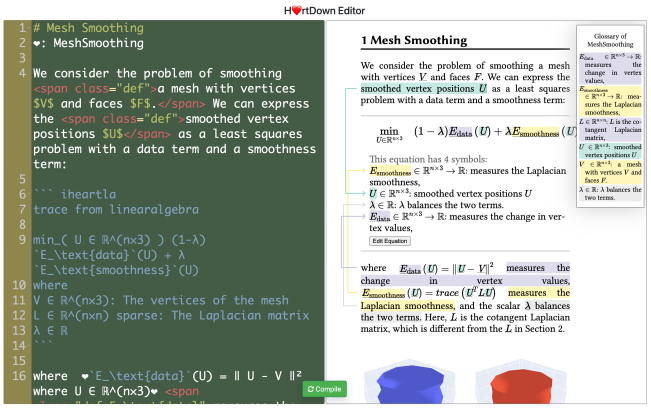

Fig. 4. The H♥rtDown editor for the mesh smoothing document. The example from Figure 2 has been modified with a $(1-\lambda)$ term in front of E_{data}. The figures update automatically.

in the context. The constructor computes the field values from the input parameters. Local functions become methods on the struct or class. We also extended I♥LA to support instantiating and importing symbols from other I♥LA modules.

Splitting a paper into contexts may be necessary due to intentional symbol re-use. Contexts can also organize the functionality of a paper into smaller units for users who only wish to access a subset of the paper's functionality. This prevents users from having to provide unnecessary parameters and avoids pre-computing ultimately unused values.

4.2.2 Figures. H♥rtDown executes Python code blocks, which allows authors to generate figures programmatically. Authors already rely on code execution to generate data for figures; in H♥rtDown, the Python code can access the compiled functionality of the document as a module. During experimentation, changes to the mathematical formulas are automatically reflected in the figure; authors can skip manually implementing the math in a programming language. Figures can be also added with any supported Markdown technique, such as directly referencing an image or inserting interactive HTML visualizations. In the smoothing example (Figure 2), the Plotly library generates interactive figures that allow viewers to inspect the output. In Figure 4, the author experiments with changing how the weight λ is used by adding a $(1-\lambda)$ term in front of E_{data}.

4.3 Author Support

H♥rtDown helps authors write correct math and complete prose. H♥rtDown provides a web-based visual editor (Figure 4) that displays the document source and output viewer side by side. Error messages appear whenever the user's formulas contain incompatible indices, dimensions, or types or erroneous syntax. The editor displays the I♥LA compiler error message and highlights the appropriate line in the source. When symbols are not described with prose, they appear with red underlines in the viewer. The output of Python code that fails to run is displayed inline. Authors can also edit I♥LA formulas and Python code for figures directly in the viewer-side of the authoring environment. Clicking on either generates an editing window containing only the relevant code block.

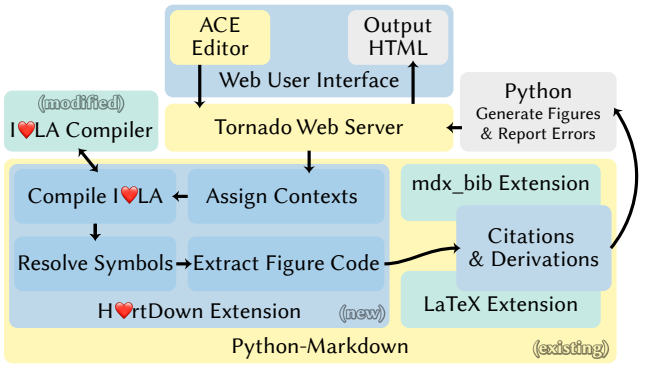

Fig. 5. H♥rtDown is implemented as a Python-Markdown extension, with additional modifications to existing extensions to adjust citation style and support inline LaTeX equation parsing.

4.4 Reading

H♥rtDown's dynamic reading environment makes use of the metadata generated by the compiler to add a glossary and math augmentations [Head et al. 2022]. Figure 2 shows our reader for the mesh smoothing example. The glossary contents dynamically match the on-screen context. Users can click on a symbol; color cues identify all uses of the same symbol and arrows guide readers to the symbol's definition in context. Users can click on an equation; an inline glossary appears and all symbols are highlighted and their definitions indicated with arrows. Colors are chosen so that no two symbols in the same equation will have the same highlight color. The functionality we support was inspired by ScholarPhi's reader [Head et al. 2021]. Some of our design choices leverage the dynamic nature of the web for e.g., re-flowing the text.

5 IMPLEMENTATION

Figure 5 shows the overall structure of the implementation of H♥rtDown. In this section, we describe how H♥rtDown makes use of existing Python libraries (with some modifications) and a modified version of the I♥LA compiler to produce output for our prototype viewer, which takes the form of an HTML document with interactivity implemented in JavaScript.

5.1 H♥rtDown Extension of Python Markdown

We developed H♥rtDown based on the open source Python Markdown library [pyt 2021], which provides easy-to-use APIs for writing Markdown extensions. We implemented H♥rtDown as an extension to Python Markdown, and slightly modified open source extensions for parsing LaTeX equations and bibliography files. We added support for inline mathematical expressions by enclosing them in $ (a single-line change). Meanwhile, the `mdx_bib` extension [Darakananda 2015] provides support for R Markdown-style citations; we modified it to resemble the SIGGRAPH bibliography style. As part of the H♥rtDown extension, we implement Pandoc-style YAML headers to specify authors, document title, and abstract.

Figure 5 shows how the H❤rtDown extension leverages Python Markdown, modified extensions, and a modified I❤LA compiler to produce output for our prototype viewer. Given an input Markdown source file, H❤rtDown first parses all the context declarations. These are used to infer each symbol's scope, so that the prose annotations in the MathJax LaTeX output, the span tags around symbol definitions in prose, and I❤LA can all omit specifying the context. Then H❤rtDown concatenates all I❤LA code from the same context and compiles it into both executable code for the libraries and MathJax.

H❤rtDown tracks the source location during this concatenation in order to place the MathJax in the output HTML. At the same time, H❤rtDown gathers metadata from I❤LA, such as precise type information for each symbol and the symbol list for each equation. H❤rtDown then searches for each symbol in the user's LaTeX derivations and applies the LaTeX command used by our MathJax extension so that derivation symbols can be queried uniformly. We explored writing a MathML backend for I❤LA, but discovered that browser support for MathML is uneven, so we would have still relied on MathJax to guarantee correct presentation of the MathML. H❤rtDown solves a graph coloring problem using a greedy technique [Liu et al. 2021] to ensure symbols in the same equation have different colors. These colors are output as JSON metadata in the HTML.

5.2 Modifications to I❤LA

The H❤rtDown extension leverages a modified version of the I❤LA compiler. Our modifications alter both the capabilities of the language and the output produced. Since I❤LA already supports MathJax LaTeX output, we modified it to include the metadata needed for our paper reading environment as LaTeX commands around symbols. This metadata includes specifying which symbols are used in each equation, as well as specifying the locations of symbol definitions.

We implement two major extensions to the I❤LA language. First, we no longer require that symbol definitions occur before their use; instead, we implement a simple symbol def-use analysis that reorders the code such that all symbol definitions occur before their uses. This allows us to compile arbitrarily-ordered snippets into a single I❤LA program. Second, we modify the compiler to support local functions, which are used frequently in papers. This modification required new syntax for specifying such functions, as well as modifications to the backends to ensure local functions are usable from clients of the output code.

In addition, we also extend I❤LA's support for modules. Previously, the language only supported modules for the purposes of importing standard library functions such as `sin`; we extend this syntax to allow importing from other I❤LA files. Our syntax extension allows users to specify inputs to an externally-defined module, which are then used to compute the outputs and make them accessible from the current module. Each context corresponds to a single file defining a single module.

5.3 Web-based Editor

The web-based authoring GUI displays the editable input source and output paper reading environment side by side, leveraging an embeddable code editor [Ajax.org 2022]. The GUI communicates via POST requests with a server running the Python-based Tornado web framework and asynchronous networking library to run the H❤rtDown document processing. To speed compilation, H❤rtDown caches I❤LA code and only re-compiles it when the I❤LA code has changed (determined via string comparison). When a figure's code is changed from the viewer, H❤rtDown only runs that Python code block.

5.4 Paper Reading Environment

We built our paper reading environment using Javascript for viewing in a browser. The paper reading environment uses JSON output by the H❤rtDown extension to visualize symbol relationships and enhance the paper reading experience. We leverage MathJax extensions to store information for symbols and equations in the HTML tags generated by MathJax when displaying LaTeX math. This allowed us to access the symbols in a structured way from JavaScript, which implements the dynamic, interactive aspects of our reading environment, and styles the symbols using CSS.

6 EVALUATION

We have extensively evaluated H❤rtDown with case studies rewriting SIGGRAPH papers in H❤rtDown and evaluating the generated code libraries, with an expert study to understand how professional researchers can make use of H❤rtDown, and with additional scenarios demonstrating how editable math facilitates experimentation. Details of our evaluations are provided in Appendices A and B and in the supplemental materials. We summarize our findings here.

6.1 Case Studies

With our case studies, we seek to answer (a) whether it is capable of being used to author papers, (b) how much overhead is required to rewrite a paper in H❤rtDown, (c) whether the resulting code library is correct and useful, and (d) what limitations we observed in practice. To do this, we converted a variety of SIGGRAPH papers and paper sections to H❤rtDown. Our criteria for selecting papers were that they use linear algebra implementable by I❤LA, while ensuring we cover a variety of topics in graphics. The papers are from the past five years (2017–2021) of SIGGRAPH and span geometry processing, image processing, visualization, simulation, and rendering. To evaluate (c), for papers with accessible existing implementations, we replaced manually-written code with calls to our automatically generated libraries. We describe this process case-by-case in Appendix A.

Replicability for research papers is not a new problem [Bonneel et al. 2020]. Out of the 14 papers we reimplemented (5 full papers and 9 papers for which we implemented single subsections), we only found code online for 7 of them. Among these, 4 are from the paper authors and 3 are third-party implementations. We use the library generated by H❤rtDown to replace functions in the original code for each of these 7 examples, and use input examples

to verify that the results match. The equations we wrote in H❤rtDown don't always have corresponding implementations in the original code; we only checked equations with implementations. Please refer to Appendix A for details about each case study and the supplemental materials for the original and the replaced source code; here we only summarize our results.

Overall, we verified 15 equations across the 7 case studies; for 5 of the studies, we generated C++ libraries, and for the other two, we generated MATLAB code. In several cases, we needed to slightly modify the way equations were written in the paper. For example, we separated out re-definitions into separate contexts (e.g., when the equation is defined with the same name for both 2D and 3D), or added variables to turn equations without assignments (i.e., without left hand sides) into callable I❤LA code. In one case, we needed to add an additional parameter to an equation due to I❤LA lacking support for computing partial derivatives. Overall, we see that H❤rtDown enables paper authors to leverage our modified version of I❤LA to produce code libraries from the same source that generates the paper, vastly improving replicability by building it into the paper authoring and reading process.

6.2 Expert Study

We conducted an asynchronous expert study to understand how active researchers can make use of H❤rtDown and the executable code it generates. We recruited 3 computer science PhD students to author an original document related to their computer graphics research. They spent a total of 24, 7, and 6 hours, respectively, using H❤rtDown over a period of two weeks. Participants were given initial and follow-up questionnaires to understand their current practices and share their thoughts about H❤rtDown. A longer description of our study's results can be found in Appendix B. Our study protocol was approved by our university's ethics board. Informed consent was obtained from all participants.

Our pre-study questionnaire asked about current research processes. The experts wrote that initial discussions involve handwriting on a real or virtual surface (like a whiteboard or drawing software) followed by formalizing ideas mathematically (possibly in LaTeX or Markdown). Finally, they implement the ideas in code. The experts observed that converting mathematical formulas to executable code is much more difficult than writing the math itself; one noted that the process is also error-prone.

The H❤rtDown documents the participants wrote can be seen in our supplemental materials. In our protocol, participants conceptualized their documents outside of H❤rtDown. If I❤LA didn't support the notation in the document, we either added support or discussed changes to the formulas with participants. Then, participants used H❤rtDown to write the documents. Two participants used Python and one used C++ to test and verify the correctness of the generated library.

At the conclusion of the study, we sent participants a follow-up questionnaire. Two participants appreciated that writing in H❤rtDown is similar to writing Markdown. Two commented that writing math in I❤LA is harder than with Markdown/LaTeX, since they were unfamiliar with the language, while the third stated that it was easier. One commented that the generated code compensates for the additional time spent writing the equations. This participant wanted a way to convert existing files to H❤rtDown documents. All participants liked the dynamic reader features. One participant commented, "H❤rtDown is an excellent tool to share tutorial[s] online—it highlights the vector dimension and variable meaning…following all the vectors/matrices/their dims is the hardest part of reproducing a paper."

Most of the limitations they encountered were due to I❤LA language limitations, such as limitations around summation ranges. We fixed the cosmetic usability problems raised by the participants, like stale information being shown when an error occurs. We plan to address limitations in I❤LA functionality. Since our goal is for H❤rtDown to be adopted by researchers, user feedback will guide development efforts.

7 CONCLUSION

We have demonstrated that H❤rtDown is a low-overhead, ecologically compatible document processor that supports authors and improves replicability, readability, and experimentation. We re-wrote a variety of papers in computer graphics and obtained implementations of key formula in multiple programming languages virtually for free. In our expert study, participants found uses for H❤rtDown in their research practice. The tutorial written by one expert became a dynamically annotated document generating with canonical executability available in multiple programming environments.

Limitations and Future Work. One limitation of H❤rtDown is that it does not consider pseudocode, literate programming, or algorithmic steps described in prose. Algorithms are often needed to make formulas useful. Without them, significant scaffolding may still be needed beyond the code H❤rtDown generates. This can be seen in our procedural figure examples. We would like to explore mechanisms for corresponding pseudocode and procedural descriptions with scaffold code. We believe that H❤rtDown would already be useful for literate programs, which can focus on scaffolding rather than re-writing the formulas.

Another limitation stems from the kinds of formulas that our extended version of I❤LA can handle. We have focused on formulas appearing in computer graphics venues, but the space of executable math and potential application domains for H❤rtDown is much broader than linear algebra or computer graphics. Authors sometimes adopt unusual notational practices that are nevertheless understandable by readers. For example, functions that make use of their parameters' subscripts as parameters in their own right (Figure 6). Another example is symbol re-definition, which may occur when re-defining a symbol approximately or when building to a more complex definition. Although readers can typically arrive at the correct interpretation without too much difficulty [Ganesalingam 2013], unusual notation poses a problem when attempting to formalize existing practice.

In the future, we would like to explore automatic or semi-automatic conversion from LaTeX to H❤rtDown. Extending I❤LA to support non-compilable, display-only math would allow authors to write all of their math in I❤LA, rather than a mix of I❤LA and LaTeX. Incorporating a proof checker [de Moura et al. 2015; Skřivan 2022] could allow the verification of derivations. We would also

like to explore callbacks and delegates for expanding the abilities of the generated code. We would also like to improve our reading environment to support active reading activities such as annotating and comparing [Tashman and Edwards 2011].

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their suggestions, Seth Walker for helping design the reader environment, and Zoya Bylinskii for a discussion on related research projects. We are grateful to Zhecheng Wang, Jialin Huang, and Xue Yu for additional feedback. Alec Jacobson was supported in part by the Sloan Foundation and the Canada Research Chairs Program. Yotam Gingold was supported in part by a gift from Adobe Inc.

REFERENCES

2021. Python-Markdown. https://python-markdown.github.io/

Ajax.org. 2022. Ajax.org Cloud9 Editor. https://github.com/ajaxorg/ace

Yağiz Aksoy, Tunç Ozan Aydin, Aljoša Smolić, and Marc Pollefeys. 2017. Unmixing-based soft color segmentation for image manipulation. *ACM Transactions on Graphics (TOG)* 36, 2 (2017), 1–19.

Lara Alcock and Nicola Wilkinson. 2011. e-Proofs: Design of a Resource to Support Proof Comprehension in Mathematics. *Educational Designer: Journal of the International Society for Design and Development in Education* (2011). https://www.educationaldesigner.org/ed/volume1/issue4/article14/

Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, and others. 2005. The Fortress language specification. *Sun Microsystems* 139, 140 (2005).

Dennis S. Arnon. 1988. Workshop on environments for computational mathematics. *ACM SIGGRAPH Computer Graphics* 22, 1 (Feb. 1988), 26–28. https://doi.org/10.1145/48155.48158

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98.

Nicolas Bonneel, David Coeurjolly, Julie Digne, and Nicolas Mellado. 2020. Code Replicability in Computer Graphics. *ACM Trans. Graph.* 39, 4, Article 93 (July 2020). https://doi.org/10.1145/3386569.3392413

Mike Bostock. 2017. A Better Way to Code. https://medium.com/@mbostock/a-better-way-to-code-2b1d2876a3a0

Florian Cajori. 1993. *A history of mathematical notations*. Vol. 1. Courier Corporation.

Alexandre Chapiro, Robin Atkins, and Scott Daly. 2019. A luminance-aware model of judder perception. *ACM Transactions on Graphics (TOG)* 38, 5 (2019), 1–10.

Yu Ju Chen, Seung Heon Sheen, Uri M Ascher, and Dinesh K Pai. 2020. SIERE: A Hybrid Semi-Implicit Exponential Integrator for Efficiently Simulating Stiff Deformable Objects. *ACM Transactions on Graphics (TOG)* 40, 1 (2020), 1–12.

Chia-Hsing Chiu, Yuki Koyama, Yu-Chi Lai, Takeo Igarashi, and Yonghao Yue. 2020. Human-in-the-loop differential subspace search in high-dimensional latent space. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 85–1.

Matthew Conlen and Jeffrey Heer. 2018. Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. ACM, Berlin Germany, 977–989. https://doi.org/10.1145/3242587.3242600

Will Crichton. 2021. A New Medium for Communicating Research on Programming Languages. https://willcrichton.net/nota/

Darwin Darakananda. 2015. Bibliography and Citation support for Python-Markdown. https://github.com/darwindarak/mdx_bib original-date: 2015-07-13T16:11:45Z.

Fernando De Goes and Doug L James. 2017. Regularized kelvinlets: sculpting brushes based on fundamental solutions of elasticity. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1–11.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

Gyorgy Denes, Akshay Jindal, Aliaksei Mikhailiuk, and Rafał K Mantiuk. 2020. A perceptual model of motion quality for rendering with adaptive refresh-rate and resolution. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 133–1.

Anton N Dragunov and Jonathan L Herlocker. 2003. Designing Intelligent and Dynamic Interfaces for Communicating Mathematics. 3.

Mohan Ganesalingam. 2013. *The Language of Mathematics*. Lecture Notes in Computer Science, Vol. 7805. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37012-0

Alyssa Goodman, Josh Peek, Alberto Accomazzi, Chris Beaumont, Christine L. Borgman, How-Huan Hope Chen, Merce Crosas, Christopher Erdmann, August Muench, Alberto Pepe, and Curtis Wong. 2017. *The "Paper" of the Future*. Technical Report. Authorea, Inc. https://doi.org/10.22541/au.148769949.92783646 Type: dataset.

John Gruber and Aaron Swartz. 2004. Markdown. https://daringfireball.net/projects/markdown/

Tobias Günther, Markus Gross, and Holger Theisel. 2017. Generic objective vortices for flow visualization. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1–11.

Andrew Head, Kyle Lo, Dongyeop Kang, Raymond Fok, Sam Skjonsberg, Daniel S Weld, and Marti A Hearst. 2021. Augmenting scientific papers with just-in-time, position-sensitive definitions of terms and symbols. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 1–18.

Andrew Head, Amber Xie, and Marti A. Hearst. 2022. Math Augmentation: How Authors Enhance the Readability of Formulas using Novel Visual Design Practices. (2022), 18.

Alec Jacobson, Daniele Panozzo, et al. 2018. libigl: A simple C++ geometry processing library. https://libigl.github.io/.

Caigui Jiang, Cheng Wang, Florian Rist, Johannes Wallner, and Helmut Pottmann. 2020. Quad-mesh based isometric mappings and developable surfaces. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 128–1.

Ken Kennedy. 1978. Use-definition chains with applications. *Computer Languages* 3, 3 (1978), 163–179. https://doi.org/10.1016/0096-0551(78)90009-7

Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3173574.3173748

Byungsoo Kim, Vinicius C Azevedo, Markus Gross, and Barbara Solenthaler. 2020. Lagrangian neural style transfer for fluids. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 52–1.

Theodore Kim, Fernando De Goes, and Hayley Iben. 2019. Anisotropic elasticity for inversion-safety and element rehabilitation. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–15.

D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (Jan. 1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

Brooke Krajancich, Petr Kellnhofer, and Gordon Wetzstein. 2021. A Perceptual Model for Eccentricity-dependent Spatio-temporal Flicker Fusion and its Applications to Foveated Graphics. *arXiv preprint arXiv:2104.13514* (2021).

Lei Lan, Ran Luo, Marco Fratarcangeli, Weiwei Xu, Huamin Wang, Xiaohu Guo, Junfeng Yao, and Yin Yang. 2020. Medial Elastics: Efficient and Collision-Ready Deformation via Medial Axis Transform. *ACM Transactions on Graphics (TOG)* 39, 3 (2020), 1–17.

Christian Lessig. 2020. Local Fourier Slice Photography. *ACM Transactions on Graphics (TOG)* 39, 3 (2020), 1–16.

Yong Li, Shoaib Kamil, Alec Jacobson, and Yotam Gingold. 2021. I Heart LA: Compilable Markdown for Linear Algebra. *ACM Transactions on Graphics (TOG)* 40, 6 (Dec. 2021).

Hsueh-Ti Derek Liu, Jiayi Eris Zhang, Mirela Ben-Chen, and Alec Jacobson. 2021. Surface Multigrid via Intrinsic Prolongation. *ACM Trans. Graph.* 40, 4 (2021).

Wei Liu, Pingping Zhang, Xiaolin Huang, Jie Yang, Chunhua Shen, and Ian Reid. 2020. Real-time image smoothing via iterative least squares. *ACM Transactions on Graphics (TOG)* 39, 3 (2020), 1–24.

Michal Lukáč, Daniel Sỳkora, Kalyan Sunkavalli, Eli Shechtman, Ondřej Jamriška, Nathan Carr, and Tomáš Pajdla. 2017. Nautilus: Recovering regional symmetry transformations for image editing. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1–11.

Sizhuo Ma, Shantanu Gupta, Arin C Ulku, Claudio Bruschini, Edoardo Charbon, and Mohit Gupta. 2020. Quanta burst photography. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 79–1.

Xingyu Ni, Bo Zhu, Bin Wang, and Baoquan Chen. 2020. A level-set method for magnetic substance simulation. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 29–1.

Stefan Pillwein, Kurt Leimer, Michael Birsak, and Przemyslaw Musialski. 2020. On elastic geodesic grids and their planar to spatial deployment. *arXiv preprint arXiv:2007.00201* (2020).

Fons van der Plas. 2020. fonsp/Pluto.jl. https://github.com/fonsp/Pluto.jl original-date: 2020-02-23T01:50:12Z.

Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3173606

Szymon Rusinkiewicz. 2019. A symmetric objective function for ICP. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–7.

Christopher Schroers, Jean-Charles Bazin, and Alexander Sorkine-Hornung. 2018. An omnistereoscopic video pipeline for capture and display of real-world VR. *ACM Transactions on Graphics (TOG)* 37, 3 (2018), 1–13.

Tomáš Skřivan. 2022. SciLean: Scientific Computing Assistant. https://github.com/lecopivo/SciLean original-date: 2021-09-27T21:50:10Z.
Breannan Smith, Fernando De Goes, and Theodore Kim. 2018. Stable neo-hookean flesh simulation. *ACM Transactions on Graphics (TOG)* 37, 2 (2018), 1–15.
Craig S Tashman and W Keith Edwards. 2011. Active reading and its discontents: the situations, problems and ideas of readers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2927–2936.
Editorial Team. 2021. Distill Hiatus. *Distill* (2021). https://doi.org/10.23915/distill.00031 https://distill.pub/2021/distill-hiatus.
V-Sense. 2020. Implementation for Soft Color Segmentation. https://github.com/V-Sense/soft_segmentation
Bret Victor. 2011. Tangle: a JavaScript library for reactive documents. http://worrydream.com/Tangle/
Stephen Wolfram. 1988. *Mathematica: a system for doing mathematics by computer*. Addison-Wesley Pub. Co., Advanced Book Program, Redwood City, Calif.
Stephen Wolfram. 2000. Mathematical notation: Past and future. In *MathML and Math on the Web: MathML International Conference, Urbana Champaign, USA*.
Nan Xiao, Zhe Zhu, Ralph R Martin, Kun Xu, Jia-Ming Lu, and Shi-Min Hu. 2018. Computational Design of Transforming Pop-up Books. *ACM Transactions on Graphics (TOG)* 37, 1 (2018), 1–14.
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, 4 (July 2020). https://doi.org/10.1145/3386569.3392375
Jiayao Zhang. 2021. Implementation for Stable-Neo-Hookean-Flesh-Simulation. https://github.com/jiayaozhang/Stable-Neo-Hookean-Flesh-Simulation

A CASE STUDIES

The papers used in our case studies are listed in Table 1. For each paper, the supplemental materials displays the H♥rtDown source file, H♥rtDown's generated paper reading environment, and H♥rtDown's generated code library for C++, Python, and MATLAB. We also provide a link to the original paper for comparison and side-by-side screenshots. For papers with implementations, we link to the original source code and the modified code that calls our output.

Case Study 1: Iterative Closest Point. This code is written by the author in C++. The paper describes several energy objectives which we re-wrote in H♥rtDown. Equations 10 and 11 contain the solution of the least-squares problem. In the paper, Equation 11 is a free expression. In order to access its value in the code library, we modified Equation 11 by creating a symbol and defining it equal to the expression. We then modified the code to call our generated library and verified the equivalence.

Parentheses for parameters	$L(\alpha) = \coth\alpha - \frac{1}{\alpha}$	[Ni et al. 2020]
Implicit parameters	$E(\mathbf{u}) = \frac{1}{2h^2}\left\|\mathbf{M}^{\frac{1}{2}}(\mathbf{u}-\mathbf{u}^*)\right\|^2 + \sum W(\mathbf{u})$	[Lan et al. 2020]
Function subscripts are parameters	$\varphi_p(x) = \frac{p}{2}(x^2+\epsilon)^{\frac{p}{2}-1}$	[Liu et al. 2020]
Unused parameters	$S_{SR}(x,y) = \frac{\sum_{i\in\mathcal{N}} w_i \cdot S_i}{\sum_{i\in\mathcal{N}} w_i}$	[Ma et al. 2020]
Conditional assignment	$W(r,h)_{\text{cubic}} = \begin{cases} \frac{2}{3} - r^2 + \frac{1}{2}r^3, & 0 \le r \le 1, \\ \frac{1}{6}(2-r)^3, & 1 \le r \le 2, \\ 0, & r > 2. \end{cases}$	[Kim et al. 2020]
Square brackets for parameters	$E[L_{x_p}] = \frac{2}{(n-1)\sqrt{\pi}}\frac{\Gamma\left(\frac{n}{2}\right)}{\Gamma\left(\frac{n-1}{2}\right)} = \frac{2}{n\sqrt{\pi}}\frac{\Gamma\left(\frac{n+2}{2}\right)}{\Gamma\left(\frac{n+1}{2}\right)}$	[Chiu et al. 2020]
Function superscripts are parameters	$\zeta_s^\alpha(x) \equiv \frac{2^j\alpha^{-1}}{(2\pi)^{3/2}}\sum_n \beta_{j,n}^t \zeta_s^{\alpha,n}(x) = \int_{\mathbb{R}_u} \psi_s(S_\alpha(x,u)^T)\,du$	[Lessig 2020]
Parameter superscripts are parameters	$\text{area}(f^\delta) = \text{area}(f)(1 - 2\delta H(f) + \delta^2 K(f))$	[Jiang et al. 2020]

Fig. 6. Examples of local functions from our quantitative study.

Table 1. SIGGRAPH papers we entirely or partially re-wrote in H♥rtDown. Papers with an asterisk had implementations available, which we used for our comparisons. The right-most column displays the total number of I♥LA equation lines and, in parentheses, the number of I♥LA **blocks** and the number of **inline** I♥LA formulas.

Case	Source	Type	#Lines (#B/#I)
1	Schroers et al. [2018]	Paper	18 (13/4)
2	Rusinkiewicz [2019]	Paper(*)	11 (11/5)
3	Krajancich et al. [2021]	Paper	11 (9/3)
4	Chapiro et al. [2019]	Paper(*)	8 (8/0)
5	De Goes and James [2017]	Paper(*)	7 (7/0)
6	Chen et al. [2020]	Section	16 (6/0)
7	Kim et al. [2019]	Section(*)	12 (9/3)
8	Pillwein et al. [2020]	Section	5 (5/0)
9	Denes et al. [2020]	Section	5 (4/0)
10	Smith et al. [2018]	Section(*)	4 (4/2)
11	Xiao et al. [2018]	Section	4 (4/0)
12	Lukáč et al. [2017]	Section	4 (3/0)
13	Günther et al. [2017]	Section(*)	3 (3/2)
14	Aksoy et al. [2017]	Section(*)	1 (1/0)

Case Study 2: Judder Perception. The authors released MATLAB code to calculate their judder model. Because the symbol F_a is used on both the left hand side of Equation 4 and as a parameter on the right hand side of Equation 6, we used different context names when writing them in H♥rtDown. Equations 1, 7, and 8 had corresponding implementations in the source code. The latter two equations defined two local functions which were used in Equation 1. Our modified I♥LA compiler correctly analyzed these dependencies and inferred the correct ordering. We implemented the three equations and verified that our MATLAB output matched the original.

Case Study 3: Regularized Kelvinlets. This code is from the libigl library [Jacobson et al. 2018], written in C++. Libigl implemented the 3D regularized Kelvinlets for Equations 6, 15, 16, and 17 in the paper. Equation 6 defined a regularized Kelvinlet and the other three defined twisting, scaling, and pinching types, respectively. We implemented and verified all four equations. We modified Equation 6, which contained both a derivation and a formula (two equals signs) We also replaced $q \times r$ in Equation 15 by Fr since the prose in the paper established the equivalence and the implementation in libigl actually used Fr directly.

Case Study 4: Flow Visualization. The authors released code written in C++. We rewrote Section 4 in H♥rtDown and implemented Equation 17. The equation defined a 2×6 matrix M in 2D. The matrix M also relied on variables x_p and v_p which were defined as inline I♥LA blocks in the prose following the equation.

Case Study 5: Anisotropic Elasticity. There is a MATLAB implementation from the author to verify the eigenpairs derived for I_5. We rewrote Section 4.1 in H♥rtDown. We modified Equations 5, 7, and 8 to make them assignment expressions. We verified the 3D Hessian matrix in Equation 7. Its definition relies on a matrix A

defined inline in the prose between Equations 5 and 6. The section included equations for both 3D and 2D scenarios. We used different context names to separate them.

Case Study 6: Soft Color Segmentation. We were unable to find code written by the original author, but discovered an implementation in C++ on GitHub [V-Sense 2020]. We rewrote Section 3 of the paper in H❤rtDown and verified Equation 4 by showing the results were equivalent to the existing implementation. The equation defined a sparse color unmixing energy function based on multiple parameters. We wrote parameter D as a sequence of functions in I❤LA. Terms in Equation 4, such as α and u, appear in derivations. H❤rtDown tracks this symbol usage, making the derivations easier to follow.

Case Study 7: Stable Neo-Hookean Flesh Simulation. The code is from a GitHub repository [Zhang 2021] not by the original authors. We rewrote Section 4.2 in H❤rtDown and implemented Equations 18 and 19 in I❤LA. We verified only Equation 19 which defined the cofactor matrix. The implementation of Equation 18 in the original code was different from the paper so we were unable to verify it. While Equation 18 should be a local function with a single parameter F, we defined it instead in terms of F and $\frac{\partial J}{\partial F}$, since I❤LA does not support partial derivatives.

A.1 Authoring Overhead

Without access to existing LaTeX sources, we copied-and-pasted prose and wrote mathematical expressions in either LaTeX (for derivations) or I❤LA (for executable formulas). Writing any given mathematical equation in I❤LA is similar in difficulty to writing it in LaTeX, save for the additional effort needed to explicitly declare the types of all parameters and import symbols or functions from external modules. When writing in H❤rtDown, we try to use the fewest contexts possible. In some cases, we can generate a single module for all symbols if one context can handle the entire paper or section. We use a new context only when symbols are defined multiple times or when the prose clearly indicates a new context is required (e.g., having separate 2D and 3D cases). The aim of rewriting papers in H❤rtDown is to demonstrate that H❤rtDown handles all components of a technical paper: the title, the authors, the abstract, the figures, the equations, the citations and references. Thus, we don't strive for identical typesetting to the original paper, and omit details such as whether symbols are bold or not. Since H❤rtDown has detailed type information, a viewer could style the symbols automatically by, e.g., displaying matrix symbols in bold and marking vectors with arrows on top.

As H❤rtDown automatically finds all uses of an I❤LA symbol in LaTeX, the overhead for labeling *uses* of symbols is non-existent. However, symbol definitions are often described with prose. For example, "The matrix $V \in \mathbb{R}^{n\times 3}$ contains vertex positions". Authors must label this span of relevant prose as a definition.

The two largest sources of overhead are correctly specifying the types of parameters (necessary for executable code) and labeling spans of prose as definitions (extremely beneficial for readers). We claim that this overhead would be smaller for the original authors when writing a paper than for us when re-writing. We had to search the prose carefully, whereas the original authors would have directly known the types and definition locations.

B EXPERT STUDY

We conducted an asynchronous expert study to understand how active researchers can make use of H❤rtDown and the executable code it generates. We recruited 3 researchers whose work involves writing up descriptions of linear algebra formulas. Participants used H❤rtDown to author an original document related to their research. The experts spent a total of 24, 7, and 6 hours, respectively, using H❤rtDown over a period of two weeks. Participants were given initial and follow-up questionnaires to understand their current practices and share their thoughts about H❤rtDown, respectively. Our study protocol was approved by our university's ethics board. Informed consent was obtained from all participants.

Our three experts were computer science PhD students noinline]undergrad? "current or incoming" from two universities, all of whom work in the area of computer graphics. In our pre-study questionnaire, we asked participants eight questions to elicit descriptions of their research processes from initial discussions to mathematical formalization to implementation for experimentation. The experts wrote that initial discussions often involve handwriting on a real or virtual surface (like a whiteboard or drawing software). Experts then formalize their ideas mathematically (possibly in a LaTeX or Markdown document—participants were familiar with both). This formalization may include implementation details and variables. Finally, they implement the ideas in code to test the formulas. The participants reported sometimes writing math in messages software (Slack, Discord, and email), though they lamented the difficulty of writing math. The experts all observed that converting mathematical formulas to executable code is much more difficult than writing the math itself. Two experts estimated 2–4 and 2–5 times as long, respectively. The third noted that the process is also error-prone: "First, it highly depends on how detailed the authors have documented the discretized version of those equations in the paper/supplementary material. Second, even if [they] are well-documented, it still takes a long time to implement them in C++ (Eigen). And it is very easy to make mistakes/typos in this process. I would say it varies from one week to three months to get all equations in one paper correctly implemented in C++, depending on the complexity of the physical model."

One researcher wrote a tutorial about physical simulation (elastic rods), one wrote about hand-object intersection, and one wrote about relationships between line segments for 3D modeling. The H❤rtDown documents they wrote can be seen in our supplemental materials. In our protocol, participants conceptualized their documents outside of H❤rtDown. This is compatible with participants' self-described research process. Participants sent us their proposed written documents in conceptual form (for example, a picture of a whiteboard with the proposed math). If I❤LA didn't support the notation in the document, we either added support or discussed changes to the formulas with participants. For example, one participant wrote sequence indices as superscripts and one wrote a summation over elements in a set. Then, the participants wrote their documents using H❤rtDown. noinline]It would be nice to

say something about the size of these documents in terms of number of lines or number of equations or size of library or etc etc As these documents were related to participants' research projects, they were all able to test and verify the correctness of the generated library. Two participants used Python and one used C++.

At the conclusion of the study, we sent participants a follow-up questionnaire consisting of eight questions about H❤️rtDown. Two participants appreciated that writing in H❤️rtDown is similar to writing Markdown. Two commented that writing math in I❤️LA is harder than with Markdown/LaTeX, since they were unfamiliar with the language, while the third stated that it was easier. LaTeX also has a particular syntax, but participants are more familiar with it. Moreover, LaTeX is more forgiving since it doesn't check the consistency of the math. One commented that the generated code makes compensates for additional time spent writing the equations. This same participant wanted a way to convert existing files (LaTeX/Markdown/handwriting scans) to H❤️rtDown documents.

All participants liked the dynamic reader features. The glossary provided convenient access to descriptions, and the highlighting and arrows and math augmentations) that related symbols in math to the definition in prose. One participant commented, "H❤️rtDown is an excellent tool to share tutorial online—it highlights the vector dimension and variable meaning. And trust me, following all the vectors/matrices/their dims is the hardest part of reproducing a paper."

Most of the limitations they encountered were due to I❤️LA language limitations, such as limitations around summation ranges. We fixed the cosmetic usability problems raised by the participants, like stale information being shown when an error occurs and the location of saved files. We plan to address limitations in I❤️LA functionality. Since our goal is for H❤️rtDown to be adopted by researchers, user feedback will guide development efforts.

C EXAMPLE DOCUMENTS WITH FIGURES

See Figures 7–9.

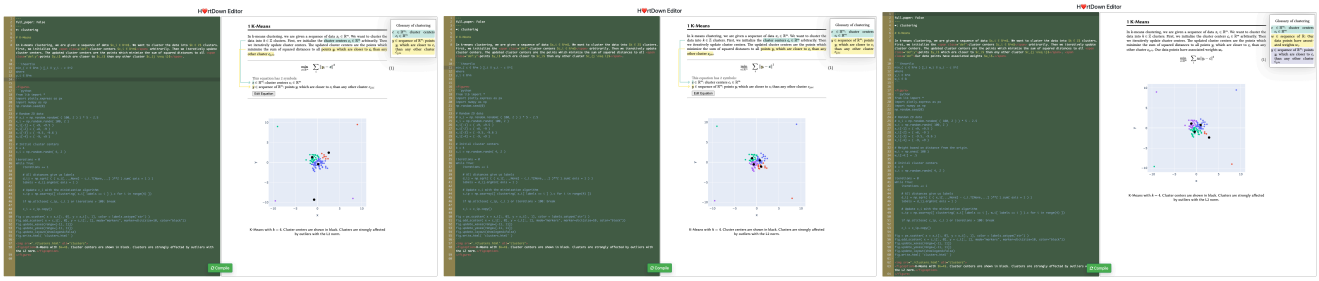

Fig. 7. Left: k-means clustering with outliers in the four corners. Middle: By changing to the $L1$-norm, we obtain k-medians clustering robust to outliers. Right: Modifying the $L2$-norm formula with weights, and assigning $w_i = \frac{1}{2}$ to the outliers mitigates their influence.

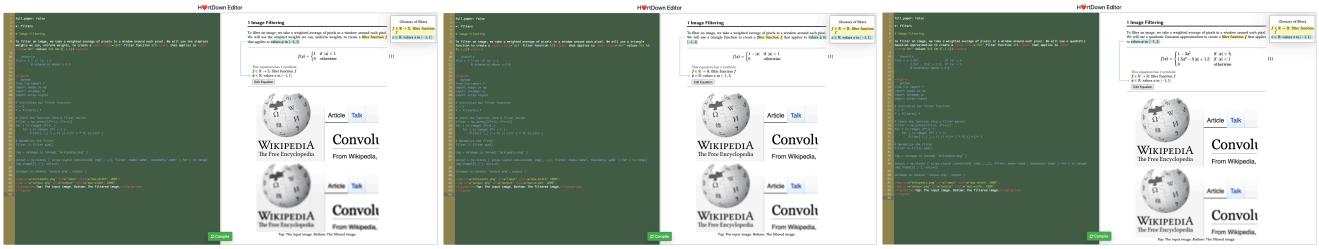

Fig. 8. Left: Convolving with a box function leads to ringing artifacts. Middle: Changing the filter function to a triangle filter leads to a higher-quality result. Right: A quadratic approximation of the Gaussian does not noticeably improve the smoothing quality.

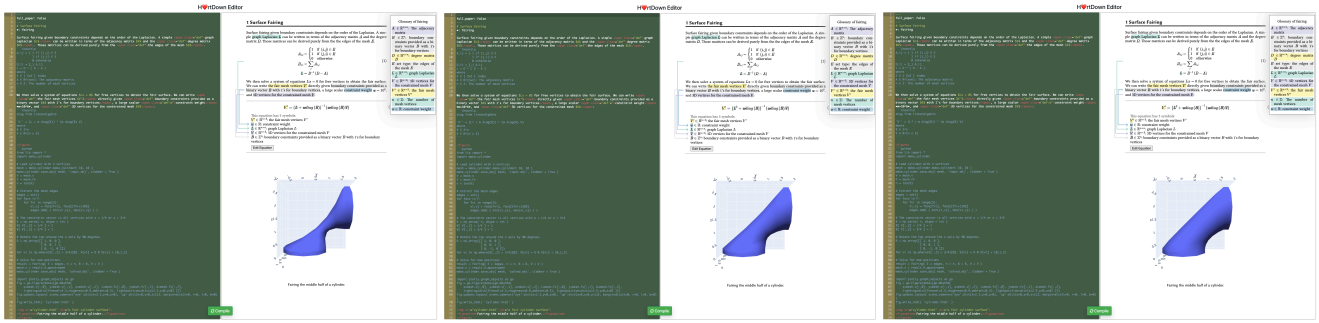

Fig. 9. Left: Fairing the middle of a bent cylinder with by solving a Laplace equation minimizing surface area. Middle: Squaring the Laplacian minimizes the thin-plate bending energy. Right: The cubic Laplacian minimizes the variation of curvature.