i-LATEX: Manipulating Transitional Representations between LATEX Code and Generated Documents

Camille Gobert
gobert@lri.fr
Université Paris-Saclay, CNRS, Inria
Laboratoire Interdisciplinaire des Sciences du Numérique
91400 Orsay, France

Michel Beaudouin-Lafon
mbl@lri.fr
Université Paris-Saclay, CNRS, Inria
Laboratoire Interdisciplinaire des Sciences du Numérique
91400 Orsay, France

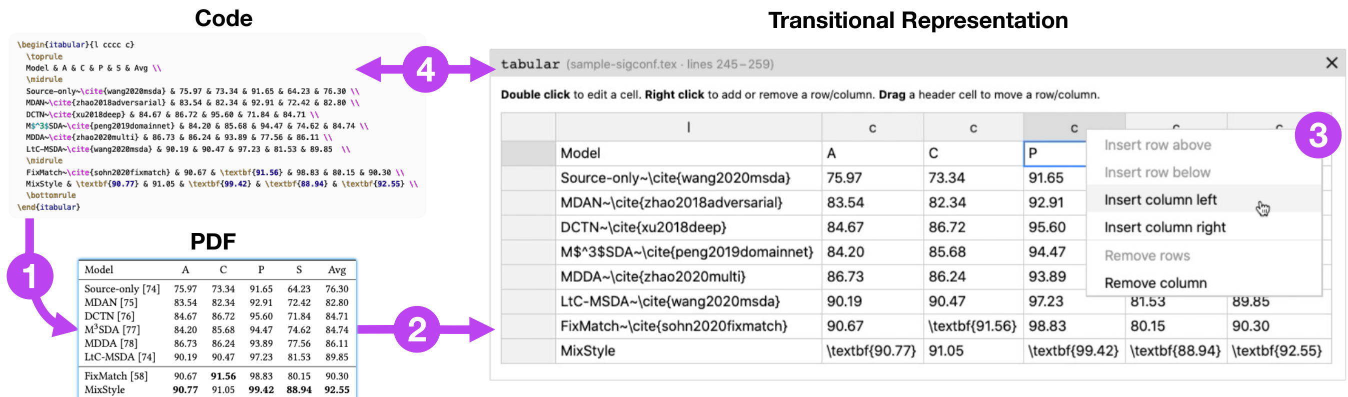


Figure 1: Three representations of the same table in *i*-LATEX (adapted from Zhou et al. [51]). (1) The code of the table is compiled into a static PDF element. (2) The table can be clicked in the PDF (as suggested by the blue halo) to display its transitional—here, the code organised in a grid. (3) The user can interact with the transitional to modify the structure and the content of the table. (4) The transitional and the code are synchronised, so that every change in either one of them instantly updates the other representation (e.g., inserting a new column in the grid adds cell separators in the code).

ABSTRACT

Document description languages such as LATEX are used extensively to author scientific and technical documents, but editing them is cumbersome: code-based editors only provide generic features, while WYSIWYG interfaces only support a subset of the language. Our interviews with 11 LATEX users highlighted their difficulties dealing with textually-encoded abstractions and with the mappings between source code and document output. To address some of these issues, we introduce *Transitional Representations* for document description languages, which enable the visualisation and manipulation of fragments of code in relation to their generated output. We present *i*-LATEX, a LATEX editor equipped with Transitional Representations of formulae, tables, images, and grid layouts. A 16-participant experiment shows that Transitional Representations let them complete common editing tasks significantly faster, with fewer compilations, and with a lower workload. We discuss how Transitional Representations affect editing strategies and conclude with directions for future work.

CCS CONCEPTS

• **Human-centered computing** → *User studies*; **Graphical user interfaces**; • **Applied computing** → *Markup languages.*

KEYWORDS

LATEX document, Code editor, Transitional Representation

ACM Reference Format:

Camille Gobert and Michel Beaudouin-Lafon. 2022. *i*-LATEX: Manipulating Transitional Representations between LATEX Code and Generated Documents. In *CHI Conference on Human Factors in Computing Systems (CHI '22), April 29-May 5, 2022, New Orleans, LA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3491102.3517494

1 INTRODUCTION

Document preparation systems are digital systems for authoring documents. They are often divided into two categories: document description languages, such as LATEX, Markdown, Asciidoc and HTML; and WYSIWYG software, such as Microsoft Word, Apple Pages and LibreOffice Writer. Document description languages require to code the document using a special markup language that must be transformed to generate an output document. This two-step process enables some of these languages to offer powerful reuse and customisation mechanisms, e.g., through scripting, but their textual format and the dualism between the code of a document

CHI '22, April 29-May 5, 2022, New Orleans, LA, USA

© 2022 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *CHI Conference on Human Factors in Computing Systems (CHI '22), April 29-May 5, 2022, New Orleans, LA, USA*, https://doi.org/10.1145/3491102.3517494.

and its output make them less user-friendly and more error-prone. By contrast, WYSIWYG software present documents in their final form at all times, and enable users to directly edit their content and style. While this makes them more accessible, it also hinders the usage of abstractions and limits the control they offer over the structure of the document. As a result, both approaches are widely used, and neither seems to supplant the other.

In this work, we focus on LaTeX, an advanced document description language. Our goal is to understand the needs and difficulties encountered by LaTeX users and create novel tools to address them. LaTeX was created by Leslie Lamport in 1984 [29] and is based on TeX, a document composition system developed by Donald Knuth in the late 1970s [26]. LaTeX offers powerful abstraction mechanisms and a high level of control over the generated document, at the cost of long compilation times and high complexity. Generating a PDF document from LaTeX code often takes seconds to minutes, and according to Knauff & Nejasmic, even expert users may "*experience a loss of productivity when LaTeX is used, compared to other document preparation systems*" [24]. Yet, nearly four decades after its inception and despite its flaws, LaTeX is still widely used to write scientific and technical documents.

This paper introduces the concept of *Transitional Representation* (*transitional* for short) and presents *i*-LaTeX, a LaTeX editor featuring four types of transitionals. Transitionals are interactive visualisations of fragments of source code that can be displayed and edited by clicking on the output they generate (Figure 1).

Our contribution is empirical, theoretical, and technical. After reviewing related work, we present the results of an interview study of 11 LaTeX users and formulate recommendations to improve the design of LaTeX editors. We introduce the concept of transitional and explain how LaTeX editors can benefit from it. We then present *i*-LaTeX and describe its user interface, features, implementation, limitations, and extension capabilities. Finally, we report on an evaluation study of transitionals in *i*-LaTeX. We conclude with a discussion of the effects of transitionals on the editing of LaTeX documents and present directions for future work.

2 RELATED WORK

Very few studies have addressed the usability of LaTeX. Therefore, we first examine existing tools for authoring LaTeX documents and their limits. We then present relevant previous work on augmented documents and hybrid programming environments and discuss how they could be adapted to improve LaTeX editors.

2.1 LaTeX editing tools

LaTeX has been reported to be one of the few authoring systems that is extensively used in the world of academic research, along with Microsoft Word [24]. Overleaf[1], a popular online LaTeX editor, recently reported a user base of 6 million users, as well as partnerships with scientific venues—such as ACM CHI—to provide their users with appropriate templates for writing papers [43].

Since LaTeX is a command language, most LaTeX editors look and work like code editors. Some of them try to provide a user experience closer to that of WYSIWYG systems, but to the best of our knowledge, they either provide basic source code formatting, e.g., AUCTeX[2] or Overleaf's *rich text* mode, or a fully WYSIWYG interface that hides the code and only supports a limited set of features, e.g., Compositor[3]. LyX[4] represents documents in an intermediate format that focuses on the content and the structure rather than the final layout and style—a paradigm referred to as *What You See Is What You Mean* (WYSIWYM) by its authors. However, it uses its own document format, and the use of the LaTeX language is restricted to importing/exporting the document as LaTeX and inserting short pieces of code, e.g., for mathematical formulae.

LaTeX code can also also be synthesised, either using a programming language or an interactive code generator. As an example, LaTeX code for tables can be generated both programatically, e.g., using the `pandas` data-science library[5] in Python, and interactively, e.g., using the tablesgenerator.com web application. The caveat of both approaches is their unidirectionality: once the code has been generated, changes made to the code will not be processed by these tools, and re-generating the code will overwrite these changes.

Finally, certain tools address the lack of mapping between the code and its output. Some LaTeX editors or plugins such as LaTeX-Tools[6] for Sublime Text enable to preview mathematical formulae and images by hovering on the code. Gliimpse [9] animates the transition between the LaTeX code and generated PDF. SyncTeX [31] helps find which region of the code corresponds to a part of the PDF—and vice-versa. However, none of them facilitate the edition of the code.

In summary, current tools for editing LaTeX documents focus on either the code or the output, with limited support for mapping one to the other or synthesising code from more appropriate user interfaces. We believe instead that both the code and the generated document can be useful to author LaTeX documents. We therefore explore how these two representations of the same document can be improved and better connected.

2.2 Augmented documents

In his 2011 essay *Explorable Explanations*[7], Bret Victor argues in favor of making traditionally static documents more interactive. Since then, several systems have been presented that augment documents with interactive features. Dragicevic et al.'s multiverses [10] let readers explore multiple visualisations and analyses of experimental data in a single document. SpaceInk [45] lets users interact with documents by making space for hand-written annotations. ScolarPhi [19] augments scientific papers with overlays that provide definitions and context for technical terms and symbols. The technical challenge of creating more interactive documents has also been addressed by libraries such as Tangle[8], specialised tag languages such as Idyll [8], and computer vision-based systems such as Chameleon [34].

In addition, the recent surge in popularity of *literate computing* [15] such as computational notebooks (e.g., Jupyter notebooks[9])—a modern take on Knuth's *literate programming* [25]—has fostered

[1] https://www.overleaf.com

[2] https://www.gnu.org/software/auctex/
[3] https://compositorapp.com/
[4] https://www.lyx.org/
[5] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_latex.html
[6] https://latextools.readthedocs.io/en/latest/
[7] http://worrydream.com/ExplorableExplanations/
[8] http://worrydream.com/Tangle/
[9] https://jupyter.org/

the development of more interactive systems for editing documents that interweave text and code. mage [23] and B2 [49] display objects encoded as text in code cells (such as dataframes) as interactive widgets. Users can use them to visualise the data they manipulate in their documents, and the systems synthesise code that reify the interactions of the users into code so they can be reused. Wrex [11] is a similar system that synthesises data table transformations inferred from user-provided examples. Sketch-n-Sketch [20] blurs the line between code and graphics even more, as it generates vector graphics images using a functional language and lets the user modify the code by interacting with the image. It illustrates Chugh et al.'s idea of *output-directed programming* [7], which consists in editing the source code of a program by directly manipulating its output.

With the exception of Sketch-n-Sketch, all these systems are either (1) designed for readers rather than authors or (2) focused on facilitating the manipulation of the code that is part of the content of the document—not the code that generated it. While the techniques they use may be adapted to LaTeX editors, none of them currently facilitates the authoring of code-based textual documents.

2.3 Hybrid programming environments

Programming has become an increasingly visual task since the 1970s. Smalltalk-80 [16], one of the first object-oriented languages, reified abstract objects into GUI elements. More recently, block languages such as Scratch [44] have become popular tools for teaching programming to children. Yet, visual programming has not replaced textual languages and code editors, which remain the primary means for creating software today. Instead, following early work such as Avrahami et al.'s FormsVBT system [3] and Erwig & Meyer's heterogeneous languages [14], programming environments have become more and more *hybrid* over the last two decades. We present some examples following Myers' taxonomy [36] that distinguishes between visual languages, which enable to create programs by combining graphical elements, and code visualisations, which represent programs or data primarily encoded as text.

On the one hand, visual programming languages have been increasingly connected to textual languages. Recent work suggests that both students and professionals could benefit from hybrid block languages that combine blocks and text [4, 47]. Droplet[10] and Pencil Code[11] already support this dual view of programs. Frame-based editors [28] unify both views by supporting direct manipulation of blocks of code—a modern take on structured editors [46]. Data-flow oriented languages follow a similar trend, as illustrated by programming environments such as Nodes[12] and Enso[13], which combine data-flow graphs with textual programming.

On the other hand, code editors have been augmented with various types of code visualisations. They have been used to present data to programmers, including raw values [32], small inline visualisations [21, 33], or large schematic data structures [17, 22, 41]. They have also been used to manipulate the underlying code. Codelets [39] and Graphite's *palettes* [38] support the insertion of code snippets through specialised interfaces. Livelits [37] push this approach further by enabling to manipulate textual expressions as GUI elements even after they have been inserted. Similarly, Clint [53] displays nested loops as diagrams that can be manipulated to transform and optimise the code of parallelised programs. Projectional editors, such as Envision [2] and those generated by Barista [27] and JetBrains MPS[14], can display any node of an abstract syntax tree as raw text, structured text, e.g., nested conditions as two-ways tables, or graphics, e.g., state machines as graphs.

Recent work suggests that the convergence of textual and visual programming towards hybrid environments facilitates code comprehension [1], code manipulation [52], and error finding [40]. Yet, in spite of LaTeX's similarity with "regular" programming languages, and with the exception of pre-rendering formulae and images, we are not aware of any hybrid LaTeX editor that eases document authoring in LaTeX.

3 INTERVIEWS OF LaTeX USERS

Given the scarcity of previous studies of LaTeX users, we conducted a series of interviews and performed a thematic analysis of the difficulties they encounter. We present the methodology, the generated themes, and our recommendations for designing better LaTeX editors, which informed our design of *i*-LaTeX.

3.1 Methodology

Participants. We interviewed 11 participants (5 women and 6 men, age 21 to 40), recruited via an internal lab mailing list and a post on a Facebook group of university students. They did not receive any compensation for their participation. Eight were MSc students, the others were a PhD student, a high-school teacher, and an associate professor. Most of them were neither beginners nor experts with LaTeX. Each of them had used LaTeX in the weeks or months prior to the interview, usually with Overleaf (5/11) or Texmaker (5/11). Additional details are available in Appendices A and B.

Setup. All the interviews were conducted remotely via screen sharing. They lasted about one hour in average.

Procedure. The interviews were semi-structured: we used a list of predefined questions to guide the participants, but they were invited to speak freely about each problem they encountered. We adjusted the duration and the questions with three pilot interviews with colleagues. We started each interview by asking the participant to show us the last LaTeX document they worked on (both the code and the generated PDF). We first asked them questions related to the document itself (the type of document, the editor they used, etc). We then asked them to describe the different problems they faced, whether they eventually solved them or not and how, and to show us the related parts of the code when it was relevant. We also invited them to tell us about problems they encountered in other LaTeX documents. We concluded each interview with a few more general questions about the participants' experience with LaTeX.

Data collection. We recorded the screen and the audio of the participants. We also took notes of the problems they faced and the solutions they used.

[10] https://droplet-editor.github.io/
[11] https://pencilcode.net/
[12] https://nodes.io/
[13] https://enso.org/
[14] https://www.jetbrains.com/mps/

Data analysis. We conducted a thematic analysis [6] of the collected data. The first author manually transcribed the first eight interviews, including descriptions of the screen content and participants' actions when it was relevant. He generated more than 650 codes from the transcripts. Codes that conveyed the same idea with different phrasings were merged, resulting in about 80 different codes, that we then used to generate three sets of themes: a first set that grouped sub-themes by topic; a second set that grouped sub-themes by problem (independently of the context in which they appeared); and a third set that mixed the first two based on discussions with colleagues, with the goal of identifying the most relevant themes within the first two sets. This analysis eventually resulted in five themes. In addition, we also listened to the three interviews that were not transcribed to (1) ensure that they fit with the final set of themes and (2) extract quotes from them to both reinforce and contrast what had already been highlighted in the other interviews.

3.2 Results

We present the five themes, summarised as follows: the code must be editable by the users (T_1), users struggle with the language (T_2), plain text is sometimes inappropriate (T_3), abstractions are difficult to visualise (T_4), and the code-PDF duality slows users down (T_5).

3.2.1 T_1 — The code must be editable by the users.

While most participants were comfortable using graphical text processors, P2 complained that the documents are not properly structured and both P4 and P11 highlighted the quality of documents produced with LaTeX.

No participant was aware of or interested in WYSIWYG LaTeX editors, and no participant who used Overleaf actively used its *rich text* mode. When she had a look at it, P8 criticised that hiding the code made it harder to make certain changes, such as transforming a section into a subsection, therefore requiring "*more manipulation, more movement on the screen*" (P8) to regularly go back to the code. P7, the only participant who used it once to write a document, explained that it was because she only had to write plain text to take notes during a history class, without any kind of structured content.

Some participants (5/11) needed the ability to write code to program features that did not exist out of the box. Most notably, P10 created a template for a collection of hundreds of high-school philosophy exams. It enabled him to generate multiple indexes (by author, by topic, by type of philosophical question, etc.) that are automatically updated when he adds new exams later on—a level of automation he could not achieve with his usual text processor: "*with LibreOffice, you can only have an alphabetical index for one category*" (P10). The separation between content and style also enabled him to generate two PDFs with very different layouts by switching a single parameter in the code: one for teachers with extra metadata, and one for students that mimicked the official exam's layout.

Several participants (7/11) explained that they appreciate reusing code. The most expert ones make extensive use of custom commands to reuse mathematical expressions or duplicate a parameterised drawing. P5 explained that having access to the code makes it possible to easily copy-paste snippets found on the internet (by contrast with having to follow the steps listed in a tutorial) and to reuse the code of his scientific papers to create presentations for conferences with Beamer. It also enables him to generate documents from scratch. When he was responsible for creating several hundred similar badges for a Go contest, he wrote a Python program to generate the appropriate LaTeX code: "*If you ask me to make you two hundred badges in Word, you're on your own; I'm not doing it*" (P5).

Most participants (9/11) reported using comments within the code. They rely on them for different purposes: (1) remembering the role of a package or command; (2) discussing with co-authors; (3) keeping unused pieces of code that might be useful later; (4) commenting off lines to find the source of an error; (5) reusing parts of an old piece of code to write a new version; or (6) planning what to write in each part of the document.

3.2.2 T_2 — LaTeX users struggle or avoid learning the language.

Five participants explained that they only learn LaTeX when they need to solve a problem they face. Yet, several of them also pointed out that this often happens in moments of rush, such as before a deadline, therefore leaving them no opportunity to take the time to understand the code that caused or solved a problem. As a consequence, fixing specific problems was often described as time-consuming when there was no ready-made solution. For this reason, P2 and P4 regretted not taking a course on LaTeX. P6, on the other hand, said that he would not make such an effort: "*You find a document that works, you copy and paste it, and you iteratively change it. […] I will not learn the structure of the code.*" (P6).

Instead of trying to learn the language or reading the documentation of the packages they use, all the non-expert participants (8/11) reported that they would rather search for solutions to specific problems: creating a particular style or layout, adjusting a certain margin, fixing an error, etc. In addition, most participants (9/11) seem to forget or ignore the exact names of some commands or the order and the meaning of their parameters, even when they use them on a regular basis: "*I know I had to use a minipage, but I still had to look for how to use it*" (P3). However, all but one participant reported that is was not always easy to find a solution to their problems. When she had to insert ancient Greek into her document, P8 "*looked into stuff for linguists and historians*", but could not find an answer.

3.2.3 T_3 — Plain text is inappropriate for structured content.

Many participants (8/11) complained about the difficulty of describing structured elements such as tables, sub-figures, and chemical formulae. In particular, creating or editing tables was often described as "*really annoying*" (P1). According to P7, because of the syntax of the code of a table, "*forgetting a column is hell*" because "*for every row I must count to put [the cursor] in the right place*" (P7).

In response to these difficulties, more than half the participants (6/11) use third-party tools. Both P2 and P4 started to create sub-figures by combining several images in LaTeX before switching to Inkscape or Adobe Illustrator because it gave them "*more control on the layout*" (P2). P8 reported that one of her co-authors used Paint to annotate a photography they used in their LaTeX document, something that would have been directly "*feasible with Word*" (P8). To create tables from data stored in Microsoft Excel spreadsheets, P7 exported them as CSV files that she loaded into tablesgenerator.com, an online table editor that generates LaTeX code. However, because of the complexity of the generated code, she had to repeat the whole process every time she modified the data in Excel. P1, P7 and P9 also

reported using this website to create tables from scratch. In addition, P7 also complained about not being able to export the LaTeX code of the molecules she created in ChemDraw, a professional tool for chemists, because the textual syntax of the `chemfig` package was "*the hardest thing I've ever used in LaTeX*" (P7).

3.2.4 T_4 — Abstractions are difficult to visualise and formalise. Many commands in LaTeX require specifying dimensions, such as the size of an image or the margin around an element. However, according to several participants (7/11), it is difficult to express a length that they picture in their head as a value with a unit, and it is difficult to picture what a numeric length represents. P8 faced this issue when she inserted images into her document: "*I don't necessarily know the exact size I want the image to be, but [I know] I want it to be* that *size in my head*" (P8). P1 complained about having to try several dimensions to find the right one—a time-consuming strategy, "*especially when you have a lot of figures, as it takes a long time to compile.*" (P1). To overcome these difficulties, some participants (4/11) mentioned using commands they are more familiar with in an alternative way. For example, P3 inserted white text in his document to skip several lines of text: "*I suppose there was a simpler way to do it, but since I was in quite a hurry, that's how it is*" (P3).

Unlike dimensions, LaTeX often determines the positions of various elements for the user. This can result in a lack of control that is not always desirable. P1 explained that her way of positioning figures was "*not very rational*", and P2 complained about the difficulty of displaying an image next to a paragraph that refers to it when LaTeX places it elsewhere. These difficulties were sometimes caused by a lack of understanding of positioning parameters, such as those of `figure` environments, which were often copied with the rest of a piece of code. Even the more expert participants struggled with positioning. In spite of reading about "*how the compiler positions images*" to better control the process, P4 admitted that she still had to lower her expectations concerning the positions of her figures. P5 explained that even though he felt comfortable with the drawing commands of the TikZ package, he would prefer to be able to directly manipulate some of the elements that compose his drawings instead of "*trying to guess*" the correct coordinates or doing "*some kind of trigonometry*" to calculate them.

3.2.5 T_5 — The code-PDF duality causes errors and slowdowns. Participants who wrote a lot of mathematical formulae (3/11) complained about the difficulty of relating regions or symbols in the PDF to the code that generated them. When writing mathematical papers, both P5 and P9 have trouble (1) locating the code of the mathematical formula displayed in the PDF and (2) finding the symbol they want to edit within the code of the formula. To solve the first problem, they often search for a few words from the text located just above or below the formula in the code editor, although P5 admitted that this technique regularly fails. This approach is not unrelated to the lack of support for SyncTeX in the LaTeX editor that P5 uses, and he even explained that he considers switching to another editor for that reason. Neither P5 nor P9 have found a solution to the second problem: every time they want to edit a formula, they have to read the code to find the part to change.

Two participants complained about the time required to compile the code into a PDF, which not only increases the cost of trying alternative layouts, but also makes errors very time-consuming: "*if I get it wrong it costs me a minute*" (P5). Both of them developed strategies to minimise this cost. In order to compile less often, P1 distinguishes between writing and formatting phases: "*when I'm in a writing phase [...] I just write [...] but when I'm formatting, I always open [the code editor and the PDF] on the side, to compile regularly.*" (P1). P5, on the other hand, gave the example of a package that caches images created with TikZ: as long as the code is not modified, it "*re-injects the image instead of the TikZ code*" (P5). In addition, P4 mentioned that compilation time was an important factor for choosing a LaTeX editor.

3.3 Recommendations for design

This analysis reveals a variety of problems faced by LaTeX users and suggests several opportunities to improve the design of LaTeX editors. We summarise them in four recommendations, R_1 to R_4. First, **always give access to the code to the users** (R_1) to enable them to modify, reuse, or generate parts of their documents in ways that may be unsupported by the rest of the user interface. Second, **facilitate specific and common actions** (R_2), especially for less experienced and intermittent users of LaTeX, such as organising subfigures and editing tables. Third, **provide visual representations of abstractions used in LaTeX** (R_3) to facilitate the constitution of mental representations of the document, and therefore reduce the number of compilation cycles. Fourth, **make the links between the code and the PDF more visible and specific** (R_4) to aid the identification and modification of the piece of code responsible for a particular PDF element, and conversely.

4 TRANSITIONAL REPRESENTATIONS

Most document authoring systems rely on a single editable representation of the document—whether it is plain text for document description languages[15], structured content for WYSIWYM, or the final output for WYSIWYG. While this reduces the cost of switching among multiple representations, it also constrains users to visualise and interact with every element of every document through a single type of representation.

We argue that a single editable representation of the code of a LaTeX document is not always sufficient. In order to address this limitation, we introduce *Transitional Representation* (*transitional* for short) as the reification [5] of abstract concepts that are not readily accessible in the code into interactive visualisations. We frame the concept, define its key properties, compare it with related concepts and systems, and show how it applies to LaTeX.

4.1 Definition and properties

We define a transitional as an alternative representation of textual code that mediates the link between a fragment of the source code and the part of the output document that it generates. We situate the concept among existing document authoring paradigms in Figure 2. A transitional can be used to visualise the code of an element with a representation that is complementary to the raw code and to the output, e.g. by showing the grid structure of a table with no visible cells, as well as to manipulate the code more conveniently than by editing it directly, e.g. by reordering rows and columns in that grid using drag and drop (Figure 1). Transitionals are *local*, *bidirectional*,

[15] We do not consider the generated documents here, as they usually are *not* interactive.

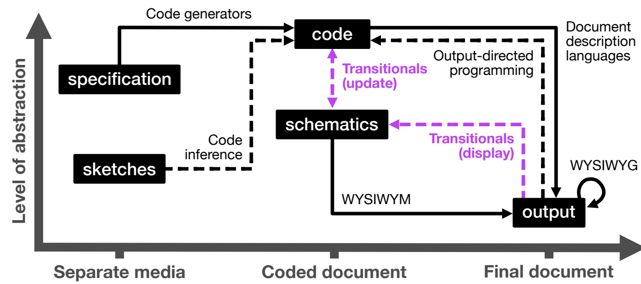


Figure 2: Space of document authoring paradigms. Nodes represent document representations, and arrows represent authoring paradigms that connect different representations. Transitionals can be displayed by interacting with the output, and modifying them updates the code—and conversely. Some other missing connections for coded documents (dotted) have recently been addressed by synthesising code from sketches [12] or from changes in the output [35].

persistent, and *embeddable* representations of textual code that are primarily designed for document description languages.

Locality. Transitionals are **local representations of the code** that are specialised for interacting with a single part of the code, which represents a single element of the document. In contrast with WYSIWYG systems such as Adobe Dreamweaver and output-direct programming systems such as Sketch-n-Sketch [20], which require the whole output to be interactive, transitionals can be used with a static output, as they can be displayed *in addition* to the code and the output—and not *instead* of them.

Bidirectionallity. Transitionals are **bidirectionally linked to the code** they represent, so that every change in the transitional is reflected in the code, and vice-versa. Unlike code generated once from a specification or a sketch [12], this approach enables users to dynamically switch between the code of an element and its transitional, depending on which representation is the most adapted to their current needs. As an example, a user may use a transitional to interactively adjust the size of an image, and switch to the code to fine-tune the width and height parameters.

Persistence. Transitionals are **persistent representations of the code**: similar to Livelits [37], they can be hidden and displayed again at any time. Unlike code generators and Graphite's palettes [38], they are updated when the code is modified.

Embeddability. Transitionals can be **displayed from and within the output**, where they can be visually embedded, without requiring to write a special command in the code as in mage [23] or to locate the piece of code responsible for generating a certain element—with all the challenges it implies, as discussed in theme T_5.

4.2 Related concepts

The concept of Transitional Representation bridges the gap between *projectional editing* [27], where fragments of code can be displayed as alternative representations when needed, and *output-directed programming* [7], where manipulating the output of a program transforms its source code. Transitionals address the following two limitations of these concepts by enabling a form of "output-directed projectional editing".

The first limitation is conceptual: the content of the code or the output alone may not be sufficient or adapted for performing certain edits. As an example, Mozilla Firefox's developer tools include an alternative representation of the value of a CSS timing function property that may be hard to interpret and edit as code, even though the webpage is updated after each modification of the CSS (Figure 3). The authors of Sketch-n-Sketch make a similar observation regarding the usage of output-directed programming in non-trivial documents: "*manipulation of the final output alone will be insufficient*", and therefore, "*some of the intermediate process should be exposed on the canvas for manipulation*" [20]. Transitionals address this limitation by design, as a transitional can include more, less or different information, to make up for what is missing in the output or help the user focus on what is important. In addition, by providing multiple transitionals for a single element, users can choose the representation that fits their current need.

The second limitation is technical: depending on the document description language, evaluating the code may be too slow for real-time code synthesis and document rendering. As an example, non-trivial LaTeX documents are usually too slow to compile to enable real time updates of the output when the code is modified—let alone turning the static output generated by the LaTeX compiler into a fully interactive document [30]. By displaying a projection of the code of a single element of interest, the compiler can be traded for an ad-hoc static analysis that extracts all the information required by the transitional in real-time. In addition, by choosing a projection that is visually similar to the generated element, transitionals enable a form of local live programming—not unlike what Projection Boxes [32] offer in the code editor—where users can see the effects of the changes made in a specific region of the code on the projection in real time, and vice-versa.

Transitionals are further inspired by recent work on GUIs for code such as Graphite [38], mage [23], and Livelits [37], which enable users to use GUIs to write or edit specific pieces of code more conveniently while letting them edit the code directly for all the other edits. However, unlike these systems, transitionals are the only alternative representations of code to satisfy the four properties presented earlier: there is no bidirectionality in Graphite, no persistence in mage, and no embeddability in Livelits. We are not aware of other concept of GUI for code tailored for real-world document description languages as conceptually and technically challenging as LaTeX.

4.3 Application to LaTeX

The properties of transitionals and the concepts they build upon make them conceptually and technically adapted to support the edition of LaTeX documents. Our thematic analysis shows that while editing the code of LaTeX documents is sometimes preferred or required (T_1), plain text is ill-adapted for working with structured content (T_3) and abstract values (T_4). It also highlights two areas where current LaTeX editors fall short: supporting specific actions on common types of elements (T_2), and connecting the code with its output (T_5). We argue that transitionals are a good option for

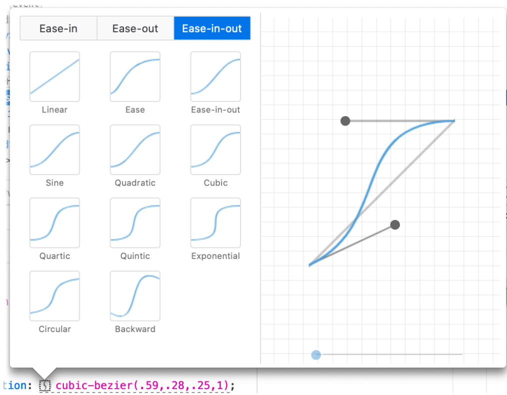


Figure 3: Alternative representation for timing functions of CSS animations in Mozilla Firefox. The curve on the right represents the function that controls an animation that runs in the webpage. Manipulating the two handles transforms the curve and updates both the numeric parameters of the CSS property's value (shown below) and the animation running in the webpage (not shown).

improving LaTeX editors, as they adhere to the four recommendations presented earlier: they complement code editing (R_1), they are designed to facilitate specific tasks in specific elements (R_2), they can visualise structures and abstractions (R_3), and they help linking a piece of code and its output (R_4).

5 THE *i*-LaTeX EDITOR

Informed by the results of the thematic analysis, we selected four candidates for Transitional Representations in LaTeX and created *i*-LaTeX, a LaTeX editor featuring interactive code visualisations for mathematical formulae, tables, images, and grid layouts. We present the design of *i*-LaTeX's user interface, the four transitionals, the key points of the implementation, and its current limitations and possible extensions.

5.1 User interface

The interface of *i*-LaTeX resembles that of most traditional *i*-LaTeX editors, with the source code on the left and the generated PDF on the right. However, some elements in the PDF have a blue outline, indicating that they are interactive. Clicking on one of them displays an interactive visualisation of the piece of code that generated the element (Figure 4). These transitionals let users (1) visualise invisible structures and abstractions and (2) modify the source code of the document in a more interactive way.

Depending on the position of the clicked element on the screen, the transitional is displayed in a panel either above or below it so as to leave the rendered element as visible as possible. The rest of the document output is darkened until the transitional is closed by clicking on the cross at the top-right of the panel or anywhere on the darkened document. Closing the transitional also recompiles the LaTeX document and updates the PDF.

The title bar of each transitional displays the name of the file and the range of the code that is visualised. Clicking on it displays the code in the code editor by opening the appropriate file if needed and scrolling to the relevant section of the code. The user can edit the code directly in the code editor. The code is re-parsed after every keystroke, to update the visualisation. If an error is introduced, the piece of code is highlighted in red in the code editor, and the visualisation is replaced by an error message that invites the user to fix the problem. The visualisation is restored as soon as the error is fixed.

5.2 Transitionals

We have implemented four kinds of transitional in *i*-LaTeX (Figure 5): three for standard LaTeX structures (mathematics, images, tables), and one for a custom grid layout.

5.2.1 Mathematics. Interactive mathematical formulae can be added to the document with the `imaths` environment, a wrapper around the `align*` environment[16]. The interactive visualisation of a formula (Figure 5a) displays an editable copy of the code of the formula along with the typeset formula. Hovering over a symbol or a group of symbols in the typeset formula—such as a fraction—highlights the piece of code that produced it, and clicking on it selects that piece of code. Editing the copy of the code instantly updates the typeset formula in the transitional. If an error is detected, an error message is instantly displayed in the visualisation, so that the user can fix the code of the formula without having to recompile the whole document. This transitional provides a strong link between the code of a formula and its output (R_4), while acknowledging that many users prefer to edit the code rather than directly manipulate formulae (R_1).

5.2.2 Tables. Interactive tables can be added with the `itabular` environment, with the same syntax as the standard `tabular` environment. The interactive visualisation of a table (Figure 5b) displays the code of the table in a grid, as well as the type of each column in the header row. The raw content of each cell can be selected in the code by clicking and edited by double-clicking. Columns and rows can be inserted and deleted via a contextual menu as well as reordered by dragging their respective headers. This transitional lets users see and manipulate the table structure that is usually only visible in the PDF (R_3), making common transformations such as inserting and rearranging rows and columns much easier than with a code editor (R_2). By displaying the raw content of the table, the user is free to use arbitrary LaTeX code within each cell (R_1).

5.2.3 Images. Interactive images can be added with the `\iincludegraphics` command, with the same syntax as the standard `\includegraphics` command[17]. The interactive visualisation of an image (Figure 5c) displays it at the same size as in the PDF. The image can be resized by dragging one of the handles. Clicking a button displays a cropper that lets the user select the region of the image to display. These manipulations automatically update the parameters of the command by inserting, modifying, or deleting the `width`, `height`, `trim` and `clip` options. This transitional facilitates visualising and formalising dimensions (R_3) and helps discover and use lesser-known command options such as cropping (R_2).

[16] As provided by the `amsmath` package (https://ctan.org/pkg/amsmath).
[17] As provided by the `graphicx` package (https://ctan.org/pkg/graphicx).

Figure 4: User interface of *i*-LaTeX when a transitional has been displayed. (1) Code editor. (2) Generated PDF. (3) Transitional representation of the code of a table displayed on top of the PDF, just below the table that has been clicked. (4) Raw version of the code displayed in the transitional. The document was adapted from Xiong et al. [50] by replacing all the `tabular` environments by `itabular`.

(a) Mathematical formula

(b) Table

(c) Image

(d) Grid layout

Figure 5: User interface of the four transitionals available in *i*-LaTeX, showing examples of how they can be used. (a) Hovering over the ∇ symbol highlights the corresponding command in the code. (b) Right-clicking a cell displays a contextual menu that enables to insert and delete rows and columns. (c) Dragging a handle resizes the image while preserving the same aspect ratio. (d) Hovering over a cell displays buttons for inserting adjacent cells or deleting the cell.

5.2.4 Grid layouts. Interactive grid layouts can be added using the `gridlayout` environment, a custom environment we developed specifically for *i*-LaTeX. It consists of `minipage` environments arranged in a fixed-size area made up of rows of cells parameterised with relative dimensions to support local positioning of various types of content (text, images, tables, etc). The interactive visualisation of a grid layout (Figure 5d) displays the otherwise invisible grid-like structure. Cells can be resized by dragging a separator between two cells, reordered by dragging a cell, as well as inserted and deleted by clicking the appropriate button while hovering over a cell. Rows can be resized in a similar fashion, and a new row can be appended at the end of the grid by clicking a button. As with interactive tables, each cell displays its raw LaTeX content. It cannot be edited directly from the visualisation in the current version, but clicking on a cell selects its content in the code editor. This transitional supports the editing of structured content (R_3) and the concrete representation of abstractions such as relative dimensions (R_4). It also illustrates that transitionals may foster the development of new LaTeX environments that would otherwise be too difficult to use with raw code only.

5.3 Implementation

i-LaTeX is implemented as an extension for Visual Studio Code[18] that consists of approximately 11,000 lines of TypeScript code, along with HTML and CSS. We open-sourced the code of the extension at https://github.com/exsitu-projects/ilatex. We present the key aspects of the implementation below.

5.3.1 Providing custom commands and environments. In order to use the special commands and environments presented above to create elements that can be visualised and manipulated through transitionals, a custom `ilatex` package must be included in the preamble of the document. Each use of one of these commands/environments is associated with a unique identifier that is written to an external file of *code mappings*, along with other metadata such as the location in the code (file path and line number) and the current values of several length macros such as `\textwidth`, so that lengths using them can be evaluated by *i*-LaTeX. In addition, a PDF annotation containing the same unique identifier is added to the generated PDF, with the same bounding box as the element produced by the command/environment. While we decided to create custom commands and environments for the sake of simplicity, the existing ones we rely upon—such as `\includegraphics`—could be patched to behave in the way we just described, therefore enabling LaTeX users to benefit from transitionals without having to learn new commands/environments.

5.3.2 Extracting the code to visualise. Unlike most programming languages, LaTeX has no predefined grammar [13]. Instead, it uses some unique features, such as TeX's category codes [26, ch. 7], which enable to modify the lexical meaning of every character (such as \ denoting the start of a macro) anywhere in the document—therefore making LaTeX theoretically impossible to parse using conventional parser generators[19]. Nevertheless, certain conventions are very commonly used, such as the structure of environments. In order to extract the pieces of code to visualise, we developed a LaTeX parser that accepts a reasonable proportion of documents that follow these conventions. Every time the document is compiled, *i*-LaTeX reads and parses every file whose path exists in the file of code mappings into an abstract syntax tree (AST). For each code mapping, it then attempts to find the corresponding piece of code in the given file, at the given line and of the given type, and creates a model of the transitional with the matching AST node. The parser is designed to be simple enough to minimise the number of parsing errors and the execution time. Each transitional model can perform a more thorough analysis of its own AST node if necessary.

5.3.3 Displaying the augmented PDF. Once the document has been compiled into an annotated PDF, it is displayed using a custom PDF renderer[20]. The renderer extracts all the annotations inserted by the custom commands/environments along with their unique identifiers and uses them to add a blue halo to every element of the PDF whose code can be visualised. When one of these elements is clicked, *i*-LaTeX matches the unique identifier of the element with the correct model. If a matching model is found, it is used to populate the view of the transitional with the appropriate data that was extracted by the model, such as the content of each cell of a table. When the user interacts with a transitional, the view notifies the controller of every action of interest. The latter forwards them to the model, which is responsible for modifying the code of the LaTeX document. Every time the code is modified by a visualisation model or by the user, the AST of the file is updated, and every model whose AST is modified updates its internal representation of the code and provides new data to the view.

5.4 Limitations

5.4.1 Features. A first limitation of *i*-LaTeX is the fact that transitionals cannot interpret certain pieces of code even though their syntax may be valid and they may produce the expected result in the PDF. Such limitations could be addressed by (1) improving the static analysis of the code performed by the transitionals to extract more information and (2) developing new features in these transitionals to exploit that information. As an example, while merged cells are currently not supported by the transitional for tables, its model could be modified to process the `\multirow` and `\multicolumn` commands and the view could be modified to enable users to merge/unmerge cells interactively. However, because of LaTeX's extensible nature, there is no way to ensure that all the features available as code will be available in a particular transitional.

5.4.2 Abstraction. A second limitation of *i*-LaTeX is the absence of support for transitionals that represent PDF elements generated by custom commands. Supporting this type of abstraction in *i*-LaTeX is challenging, because it requires to (1) identify the provenance [48] of all the pieces of code that, put together, generate a certain PDF element and (2) deal with situations where a custom macro is used in multiple places, such as resizing an image inserted by a custom macro that is also used to insert the same image in other places. While some research prototypes make use of custom language interpreters designed to track the provenance of every value they compute, we could not readily use this approach in *i*-LaTeX since

[18] https://code.visualstudio.com/

[19] See https://tex.stackexchange.com/a/4205 for more details on this limitation.

[20] The renderer is based on PDF.js (https://github.com/mozilla/pdf.js/).

no LaTeX compiler currently tracks such information[21]. There is no consensus on how to solve the second challenge, which remains an open question for future work.

5.4.3 Performance. A third limitation of *i*-LaTeX is the lower performance on large LaTeX files. Since *i*-LaTeX updates the AST of a file that contains at least one transitional every time it is modified, transformations to perform in the code of a large file can accumulate. When too many edits are performed in a short amount of time, e.g., when an image is being rapidly resized, this accumulation can make *i*-LaTeX look jerky. In practice, performance is excellent for small to medium-size files, and larger documents can be split into multiple LaTeX files. For example, we could fluidly edit the source code of several long papers using *i*-LaTeX on a 2GHz MacBook Pro, such as the 750-lines long LaTeX file of the paper shown in Figure 4 [50]. In addition, the propagation of changes could be optimised to better support large ASTs.

5.5 Extensibility

i-LaTeX has a modular design that facilitates extension. Creating a new transitional requires creating (1) a model that can extract the information to be visualised from the AST node and make the necessary changes when it is modified, and (2) a view that represents the data provided by the model in the desired format. We provide controllers with an API for exchanging messages between the model and the view and registering callbacks for various events, as well as a number of utilities, such as a class for parsing, converting and manipulating LaTeX lengths and an API for operating on the AST. In addition, the `ilatex` package for LaTeX must also be modified to create—or patch—the LaTeX command or environment that will benefit from the new transitional so that every time they are used, they behave as described in section 5.3.1. Creating transitional for pieces of code that are neither a command nor an environment is also possible but not as straightforward, as it may require adapting *i*-LaTeX's parser to create new types of AST nodes.

6 EVALUATION

We conducted a controlled experiment with *i*-LaTeX. Our main goal was to evaluate whether transitionals would improve the performance of LaTeX users on several specific editing tasks that *i*-LaTeX's transitionals were designed to facilitate. In addition, we were interested in the effects of transitionals on the workload and the strategies used by the participants. We also collected feedback to improve *i*-LaTeX. We present the methodology and the results of this study and discuss its outcomes.

6.1 Methodology

6.1.1 Participants. We recruited 16 participants (2 women and 14 men, age 20–65) by posting a message on the mailing lists of several computer science labs and a group of HCI practitioners, and on a Facebook group of university students and alumni. They did not receive any compensation for their participation. All participants had used LaTeX before. 5 participants had used it for less than 5 years, 8 participants for 5 to 10 years and the other 3 for more than 10 years. 3 participants had never used mathematical formulae and 1 had never used tables in LaTeX before. Slightly less than half the participants (7/16) self-ranked their overall expertise with LaTeX as 4 or 5 on a 5-point Likert scale.

[21]The difficulty of tracking the provenance of PDF elements generated by LaTeX is further discussed by Laurens [30], who faced the same limitations regarding custom commands when developing SyncTeX [31, sec. 5].

Table 1: Description of the tasks. The tasks are grouped by type of content to edit: mathematics, tables, images.

Task	Type of instruction
T1	Insert a term in a multi-line formula
T2	Remove several parentheses in a multi-line formula
T3	Modify a term in one formula among six
T4	Sort the rows of a table by a certain column
T5	Modify the values of several cells of a table
T6	Remove a column from a table
T7	Resize an image to make it as wide as another element
T8	Remove the whitespace that surrounds an image
T9	Hide a part of an image

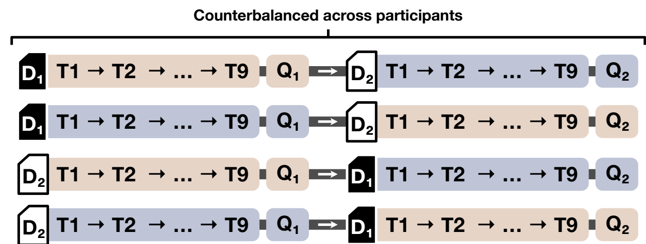


Figure 6: The four configurations of the experiment. The colour of the blocks represent whether transitionals are ■ enabled or ■ disabled. D_1 and D_2 represent the two document sets. Q_1 and Q_2 represent the two workload questionnaires.

6.1.2 Setup. The study was carried out remotely. Participants used *i*-LaTeX on their own computers and shared their screen. The study lasted between one and two hours per participant, including setup and debriefing.

6.1.3 Procedure. We used a 2×2 within-participant design with two independent variables, Transitionals (*Enabled*, *Disabled*) and Document (D_1, D_2). We adjusted the design of the study to ensure it was understandable and not too long to complete with two pilot participants. Before starting each experiment, we explained the steps of the study to the participant, asked them to read and sign the consent form, and helped them install the *i*-LaTeX extension in Visual Studio Code.

We started the experiment by asking participants to open, read, and edit an introductory LaTeX document with *i*-LaTeX. The document presented the features of *i*-LaTeX and the three transitionals used in the study (mathematics, tables, and images), with one interactive example per transitional. We also invited participants to ask questions about *i*-LaTeX or the study.

Once they felt ready, we asked participants to perform a series of 9 tasks (T1–T9), as fast as possible, on one of two similar LaTeX document sets, D_1 and D_2. Participants had to perform the 9 tasks with the first document set in one of the TRANSITIONALS conditions, and again with the other document set in the other TRANSITIONALS condition. Each document set consisted of three LaTeX documents (one for each type of tasks). The first document contained tasks T1–T3 (maths); the second document tasks T4–T6 (tables); and the third document tasks T7–T9 (images). In the condition where transitionals were enabled, we explicitly told the participants that they were not required to use them, and could always edit the code directly if they believed it was faster. After completing all the tasks in a document set, participants had to fill in a workload questionnaire based on the NASA-TLX[22]. We counterbalanced the order of the two document sets and the two conditions across participants. The four configurations of the experiment are presented in Figure 6.

Once all the tasks were completed, we asked the participants to fill in a post-study questionnaire. We debriefed them about their experience with *i*-LaTeX, answered their questions, and asked them to give us feedback on the transitional for grid layouts using a LaTeX document that we provided.

6.1.4 Tasks. The nine tasks were similar in each of the two document sets (Table 1). Each task fits on a single page of the generated PDF that contains (1) the instructions and (2) the current output of the code to modify. The tasks were designed so that they could be completed in at most a few minutes. In order to move to the next task, participants had to compile the document with no error, and the generated PDF had to contain the expected result. Participants were allowed to use external resources to complete the tasks, including online searches and other programs, as long as they did not reuse LaTeX code from other files of the study.

Most of the tasks were inspired by issues mentioned during the interviews, such as finding symbols in complex formulae and editing large tables, that we adapted to ensure that all tasks could be solved with transitionals. We decided not to include tasks with grid layouts after testing them in a pilot study, as using the `gridlayout` environment without transitionals confused participants and made the study last more than two hours.

6.1.5 Data collection. We recorded the screen and the audio of the participants, and we took notes of the strategies used and difficulties faced by the participants. At the end of the study, we collected log files generated by *i*-LaTeX on the participants' computers.

6.1.6 Data analysis. For each group of tasks, we measured the task completion times (*TIME*) and the number of compilation (*COMPILATIONS*) by processing the collected files. We also reviewed their answers to the three questionnaires and watched parts of the recordings to compare participants' behaviours and collect examples of the strategies they used to solve the tasks.

6.2 Results

6.2.1 Performance. We used Python, R, and SAS JMP to analyse the logged data. We eliminated data from one participant for tasks with images, for both conditions, because that participant could not finish tasks T7–T9 after spending more than 30 minutes trying to solve them without transitionals as they had to leave (with transitionals, this participant completed tasks T7–T9 in about 14 minutes).

A mixed ANOVA showed no significant effect of the order of the two TRANSITIONALS conditions ($F_{1,12} = 2.83$, $p = 0.12$) nor of the order of the two document sets ($F_{1,12} = 0.05$, $p = 0.82$) on *TIME*. Thereafter, we ignore the two order factors. To compare both conditions, we report the results of paired t-tests on log-transformed data for *TIME* and Wilcoxon signed-rank tests for *COMPILATIONS*. We also report effect sizes using Cohen's d for t-tests and Rank-biserial correlation (RBC) for Wilcoxon signed-rank tests. The rationale for these tests and the full results are available in Appendix C.

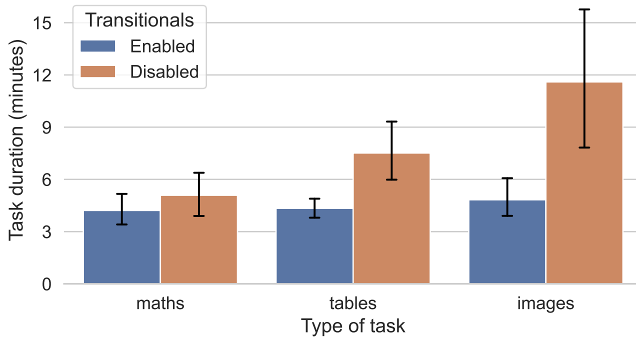
Regarding task completion time (Figure 7a), we found a significant effect of TRANSITIONALS on *TIME* for tasks with tables ($t_{15} = -4.95$, $p < 0.001$, $d = 1.39$) and images ($t_{14} = -3.75$, $p = 0.002$, $d = 1.17$), but not for mathematics ($t_{15} = -1.45$, $p = 0.17$, $d = 0.34$). According to mean task completion times, tasks were performed 44% faster when transitionals were enabled (42% faster for tasks with tables, 58% faster for tasks with images). We further analysed the results for tasks with mathematics by comparing the 6 most efficient participants with the others (Appendix D explains how we identified these groups). We found a significant effect of TRANSITIONALS on *TIME* for the least efficient participants ($t_9 = -2.37$, $p = 0.04$, $d = 0.64$), but not for the most efficient participants ($t_5 = 0.61$, $p = 0.57$, $d = 0.28$).

Regarding the number of compilations (Figure 7b), we found a significant effect of TRANSITIONALS on *COMPILATIONS* for tasks with tables ($W = 8$, $p = 0.008$, RBC $= -0.82$) and images ($W = 6$, $p = 0.001$, RBC $= -0.90$), but not for mathematics ($W = 38$, $p = 0.97$, RBC $= -0.03$). According to mean numbers of compilation, participants compiled 41% less often when transitionals were enabled (26% less often for tasks with tables, 58% less often for tasks with images).

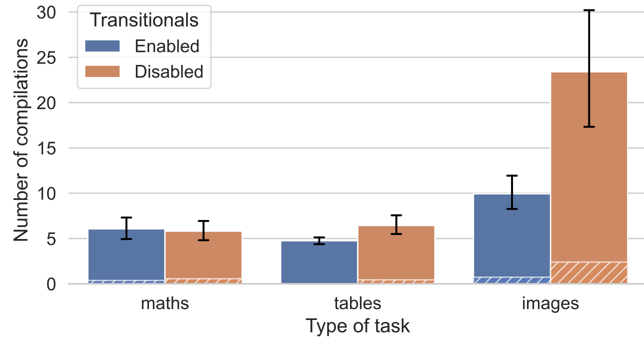
6.2.2 Workload and feedback. The answers to the questionnaires are in line with the quantitative analysis. According to the questionnaires (Figure 8), participants experienced a lower workload and performed better when they had access to transitionals. A majority of participants reported that having access to transitionals to perform the tasks was less mentally demanding (16/16), less temporally demanding (15/16), less frustrating (13/16), and made them perform better (13/16). A few participants (3/16) also reported a slightly higher physical demand when they had access to transitionals, which some participants related to the increased use of the mouse required to use the code visualisations. Several participants qualified the tool as "*very impressive*", and all the participants reported that they would *probably* (5/16) or *certainly* (11/16) use transitionals if they were available in their LaTeX editor.

Most participants made positive comments and suggestions to improve *i*-LaTeX. The suggestions include new features, mainly for tables, such as merging cells, resizing columns, manipulating row separators, and enabling multi-row or multi-column selections, as well as transitionals for other types of elements such as TikZ drawings and citations. Several participants were frustrated that the transitional would hide a part of the document they were interested in, and suggested to let users move transitionals up and down. Some

[22] We adapted five out of the six measures, with no weighting process [18].



(a) Task completion time (*TIME*).



(b) Number of compilations (*COMPILATIONS*).

Figure 7: Effect of transitionals on (a) task completion time and (b) number of compilations. Hatched areas in (b) represent failed compilations. Error bars represent 95% CIs.

participants also suggested to add a way to close a transitional without recompiling the document.

At the end of the study, all but two participants agreed to try the transitional for grid layouts. Their reaction was mostly positive. While some commented that the transitional currently lacks some features they would like to use, such as grouping cells by column, previewing the output of the cells' content, and equally distributing the available width/height, they also noted that it was already better than the solutions they use to locally position elements in LaTeX.

6.2.3 Strategies. During the study, we noticed changes in the strategies employed for solving tasks when transitionals were available. When they were only allowed to edit the code, several participants had to search either online (10/16) or in a document/book (2/16) to solve some of the tasks. Two participants copy-pasted code into Emacs to sort table rows in task T4, and four participants used an image editor to crop the images in tasks T8 and T9. Some participants also tried to come up with elaborate solutions, including computing the size of an element in LaTeX, measuring an image displayed on their screen with a ruler, and playing with negative whitespace. In such situations, most participants eventually admitted that they could not achieve what they had in mind after a few minutes of trying, and often resolved to simpler solutions and approximation by trial-and-error.

We did not observe such behaviours when participants were allowed to use transitionals. In this condition, most participants (13/16) edited the code directly at some point during the tasks, but the majority of the edits were performed through a transitional. Two participants used the code editor to search for values to replace in T5; four participants approached the expected image width using the transitional and fine-tuned the value by editing the code in T7; and seven participants used the code editor to fix errors introduced by *i*-LaTeX when resizing or cropping images[23] in T7 and T9.

No participants attempted to find "clever" solutions when transitionals were available, with one exception: a participant who completed task T4 very efficiently by using both the transitional and the code editor. He opened the transitional to move the column with the values to sort by to the left of the table, switched to the editor to select the code of all the rows in the editor, triggered a command to sort the selected lines, and switched back to the transitional to move the column back to its original position.

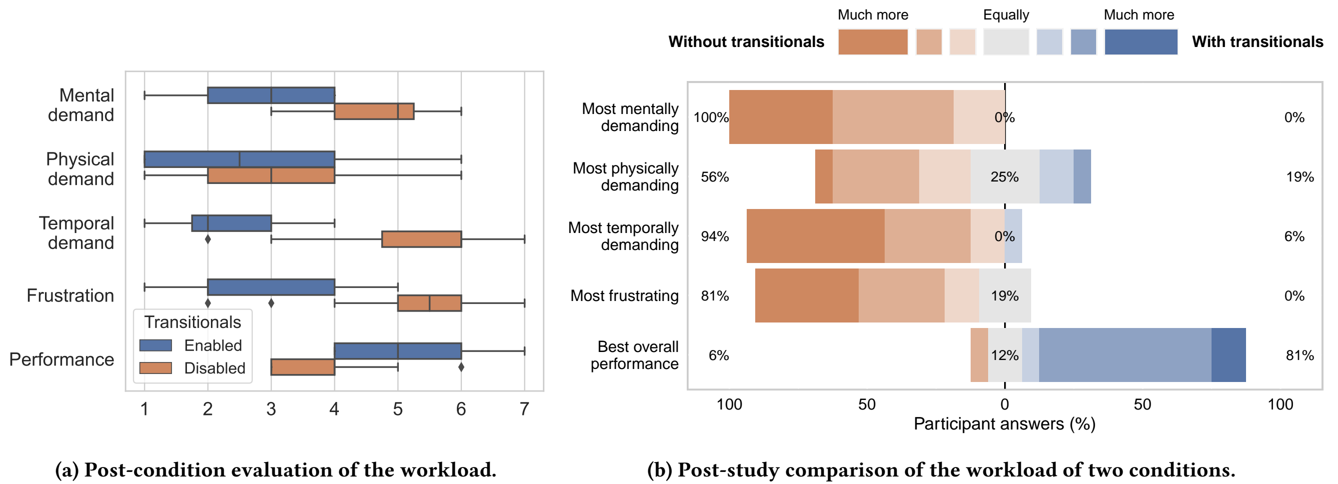
[23]Due to a bug in the current implementation of *i*-LaTeX, fast successive changes sometimes cause transitionals to become out-of-sync with the code and result in erroneous code generation.

6.3 Discussion

The results of the study show that participants solved common tasks with tables and images 44% faster and recompiled the document 41% less often when they had access to transitionals, with large effect sizes ($d > 0.8$ for t-tests, $|\mathrm{RBC}| > 0.8$ for Wilcoxon tests). While the difference is not significant for tasks with mathematical formulae overall, it is significant for the least efficient participants. This might be explained by the higher proficiency and experience with the syntax of mathematics in LaTeX of the most efficient participants, who might be more used to finding the location of a certain symbol in the code for tasks T1 and T3, or to remembering to delete both `\left` and `\right` commands along with parentheses for task T2.

In addition to improving performance, transitionals helped participants solve tasks with a lower workload, using more straightforward strategies. We hypothesise that this difference mainly stems from two characteristics of *i*-LaTeX's transitionals. The first characteristic is that transitionals enable to modify the code of the document by interacting with a possible mental model of the code, without requiring participants to (1) build their own mental model of the code and (2) map changes in their mental models to changes in the code. The second characteristic is that transitionals can help discover and use features that participants were not always familiar with, such as cropping an image directly via the `\includegraphics` command, which reduced the need for searching for tutorials and documentation. Interestingly, these characteristics may also encourage participants to solve tasks in more direct ways, without resorting to tools designed to automate sub-tasks such as sorting lines and searching and replacing text.

In summary, this study shows that transitionals can be useful to beginner and expert LaTeX users alike. Transitionals can be used by beginners to learn about common LaTeX commands and environments and try alternatives for, e.g., mathematical symbols, column



(a) Post-condition evaluation of the workload.

(b) Post-study comparison of the workload of two conditions.

Figure 8: Effect of transitionals on participant workload. (a) Post-condition rating of the workload on 7-point Likert scales (1 = very little, 4 = normal, 7 = very much). (b) Post-study comparison of the workload under each condition. The five measures are the same as in (a), but we asked participants how much one condition applies more than the other, e.g., *"Which condition was the most mentally demanding for you?"*, using symmetrical 7-point Likert scales (1 = much more the 1st condition, 4 = equally, 7 = much more the 2nd condition).

orders, or image sizes, without the time-consuming burden of re-compiling after every change. Since transitionals are optional by design, expert users can freely decide if they prefer to use them or to edit the code directly. As an example, they could use a transitional to make changes in a table only when it is large enough, or to find the command that produced a symbol in a formulae when they cannot readily find it in the code.

7 LIMITATIONS AND FUTURE WORK

While the study shows that *i*-LaTeX's transitionals helped participants perform a number of editing operations, the choice of tasks and the limits of the current implementation of *i*-LaTeX must be taken into account when interpreting the results. For instance, pointing or clicking on a parenthesis in the transitional for mathematics only selects its content, excluding `\left(` and `\right)` commands, which still had to be deleted in task T2. In addition, erroneous code generation slowed down seven participants who were using transitionals by requiring them to fix the code manually in tasks T7 and T9. Therefore we do not claim that the results of this study generalise to arbitrary editing operations—which may not always benefit from transitionals—nor to the authoring of new LaTeX documents from scratch. Furthermore, the controlled setting of the experiment does not have the same ecological validity as longer-term field studies. In light of these observations, we suggest three directions for future work.

First, *i*-LaTeX could be enhanced by creating transitionals for other types of content, e.g., TikZ drawings or chemical formulae. Existing transitionals should support more features, e.g., merging cells in tables. The overall user interface should also be improved to enable users to move transitionals and close them without recompiling. The robustness of *i*-LaTeX could also be improved by supporting a richer set of documents, both in terms of parsing—recognising more diverse syntax—and in terms of performance—better handling large LaTeX files and changes within the AST.

Adding transitionals to existing LaTeX editors such as Overleaf would make them accessible to a much wider audience. Doing so would facilitate the long-term evaluation of transitionals in more ecological settings, without requiring participants to switch to another LaTeX editor. Increasing the availability of transitionals may also foster the development of new ones by authors and users of LaTeX packages deal with concepts that are hard to visualise and manipulate as text only. Conversely, it could also enable the development of new packages designed to be used with transitionals, similar to what we did with grid layouts.

Finally, since transitionals are not specific to LaTeX, we argue that other document description languages, such as HTML/CSS, could also benefit from them. While many other languages than LaTeX can be evaluated fast enough to be used in live programming [42] or WYSIWYG environments, we argue that they could still benefit from transitionals. As an example, the sheer number of HTML/CSS code generators available online provides hints for good candidates for transitionals, such as CSS gradients, grids and transformations.

8 CONCLUSION

We presented the concept of *Transitional Representation*, an alternative and interactive representation of a fragment of code designed for document description languages. Transitionals facilitate understanding and manipulating the code of the document, and can can be displayed by interacting with the output they generate. We applied the concept to LaTeX by developing *i*-LaTeX, a LaTeX editor with transitionals for mathematical formulae, tables, images, and grid layouts. We grounded the design in the results of an interview study of LaTeX users, and we evaluated the effects of transitionals with a controlled experiment. Transitionals enabled the participants to

complete various editing tasks in LaTeX documents up to 58% faster, with fewer compilations and a lower workload. We also observed that transitionals enabled participants to achieve their goals with simpler strategies, less LaTeX knowledge and fewer trial-and-errors. Overall, *i*-LaTeX was beneficial to beginner and expert users alike, who all reported that they would like to use transitionals in their LaTeX editors. We proposed several directions to extend this work, including improving *i*-LaTeX, facilitating its adoption and evaluation, and applying transitionals to other document description languages. We believe transitionals are a powerful concept that improves both the power and simplicity of document authoring systems.

ACKNOWLEDGMENTS

We thank Wendy Mackay for her helpful inputs on the design of the evaluation and the data analysis. We also thank the interviewees and the participants of the evaluation study for their time and valuable feedback. This work was partially supported by European Research Council (ERC) grant n°695464 ONE: Unified Principles of Interaction.

REFERENCES

[1] Dimitar Asenov, Otmar Hilliges, and Peter Müller. 2016. The Effect of Richer Visualizations on Code Comprehension. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 5040–5045. https://doi.org/10.1145/2858036.2858372

[2] Dimitar Asenov and Peter Muller. 2014. Envision: A Fast and Flexible Visual Code Editor with Fluid Interactions (Overview). In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 9–12. https://doi.org/10.1109/VLHCC.2014.6883014

[3] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. 1989. A Two-View Approach to Constructing User Interfaces. *ACM SIGGRAPH Computer Graphics* 23, 3 (1989), 137–146. https://doi.org/10.1145/74334.74347

[4] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (2017), 72–80. https://doi.org/10.1145/3015455

[5] Michel Beaudouin-Lafon and Wendy E. Mackay. 2000. Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '00)*. Association for Computing Machinery, 102–109. https://doi.org/10.1145/345513.345267

[6] Virginia Braun and Victoria Clarke. 2019. Reflecting on Reflexive Thematic Analysis. *Qualitative Research in Sport, Exercise and Health* 11, 4 (2019), 589–597. https://doi.org/10.1080/2159676X.2019.1628806

[7] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. *Proceedings of the 37th Conference on Programming Language Design and Implementation - PLDI 2016* (2016), 341–354. https://doi.org/10.1145/2908080.2908103 arXiv:1507.02988

[8] Matthew Conlen and Jeffrey Heer. 2018. Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web. In *Proceedings of the 31st Symposium on User Interface Software and Technology - UIST '18*. ACM, 977–989. https://doi.org/10.1145/3242587.3242600

[9] Pierre Dragicevic, Stéphane Huot, and Fanny Chevalier. 2011. Gliimpse: Animating from Markup Code to Rendered Documents and Vice Versa. In *Proceedings of the 24th Symposium on User Interface Software and Technology - UIST '11*. ACM, 257–262. https://doi.org/10.1145/2047196.2047229

[10] Pierre Dragicevic, Yvonne Jansen, Abhraneel Sarma, Matthew Kay, and Fanny Chevalier. 2019. Increasing the Transparency of Research Papers with Explorable Multiverse Analyses. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*. ACM, 1–15. https://doi.org/10.1145/3290605.3300295

[11] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems - CHI'20*. ACM, 1–12. https://doi.org/10.1145/3313831.3376442

[12] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. *arXiv:1707.09627 [cs]* (2018). arXiv:1707.09627 [cs]

[13] Sebastian Thore Erdweg and Klaus Ostermann. 2011. Featherweight TeX and Parser Correctness. In *Software Language Engineering (Lecture Notes in Computer Science)*, Brian Malloy, Steffen Staab, and Mark van den Brand (Eds.). Springer, 397–416. https://doi.org/10.1007/978-3-642-19440-5_26

[14] M. Erwig and B. Meyer. 1995. Heterogeneous Visual Languages-Integrating Visual and Textual Programming. In *1995 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Comput. Soc. Press, 318–325. https://doi.org/10.1109/VL.1995.520825

[15] Bjarke Vognstrup Fog and Clemens Nylandsted Klokmose. 2019. Mapping the Landscape of Literate Computing. (2019), 10.

[16] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

[17] Keith Hanna. 2002. Interactive Visual Functional Programming. In *Proceedings of the 7th International Conference on Functional Programming - ICFP '02*. ACM, 145–156. https://doi.org/10.1145/581478.581493

[18] Sandra G. Hart. 2006. Nasa-Task Load Index (NASA-TLX); 20 Years Later. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 50, 9 (2006), 904–908. https://doi.org/10.1177/154193120605000909

[19] 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*. ACM, 1–18. https://doi.org/10.1145/3411764.3445648

[20] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Symposium on User Interface Software and Technology - UIST'19*. ACM, 281–292. https://doi.org/10.1145/3332165.3347925

[21] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*. ACM, 1–12. https://doi.org/10.1145/3173574.3174106

[22] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Symposium on User Interface Software and Technology - UIST'17*. ACM, 737–745. https://doi.org/10.1145/3126594.3126632

[23] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. Mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Symposium on User Interface Software and Technology - UIST'20*. ACM, 140–151. https://doi.org/10.1145/3379337.3415842

[24] Markus Knauff and Jelica Nejasmic. 2014. An Efficiency Comparison of Document Preparation Systems Used in Academic Research and Development. *PLOS ONE* 9, 12 (2014), 1–12. https://doi.org/10.1371/journal.pone.0115069

[25] Donald Ervin Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[26] Donald Ervin Knuth. 1984. *The TeXbook*. Addison-Wesley.

[27] Amy J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Proceedings of the 2006 CHI Conference on Human Factors in Computing Systems - CHI '06*. ACM, 387–396. https://doi.org/10.1145/1124772.1124831

[28] Michael Kölling, Neil Brown, and Amjad Altadmri. 2017. Frame-Based Editing. *Journal of Visual Languages and Sentient Systems* 3, 1 (2017), 40–67. https://doi.org/10.18293/VLSS2017-009

[29] Leslie Lamport. 1994. *LaTeX: A Document Preparation System: User's Guide and Reference Manual*. Addison-Wesley.

[30] Jérôme Laurens. 2007. Will TeX Ever Be WYSIWYG or the PDF Synchronization Story. *The PracTeX Journal* 3, 3 (2007), 8.

[31] Jérôme Laurens. 2008. Direct and Reverse Synchronization with SyncTeX. *TUGBoat* 29, 3 (2008), 365–371.

[32] Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems - CHI'20*. ACM, 1–7. https://doi.org/10.1145/3313831.3376494

[33] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions about Code with Always-on Programming Visualizations. In *Proceedings of the 2014 ACM Conference on Human Factors in Computing Systems - CHI '14*. ACM, 2481–2490. https://doi.org/10.1145/2556288.2557409

[34] Damien Masson, Sylvain Malacria, Edward Lank, and Géry Casiez. 2020. Chameleon: Bringing Interactivity to Static Digital Documents. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems - CHI'20*. ACM, 1–13. https://doi.org/10.1145/3313831.3376559

[35] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28. https://doi.org/10.1145/3276497

[36] Brad A. Myers. 1990. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing* 1, 1 (1990), 97–123. https://doi.org/10.1016/S1045-926X(05)80036-9

[37] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 511–525. https://doi.org/10.1145/3453483.3454059

[38] Cyrus Omar, Young Seok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active Code Completion. In *Proceedings of the 34th International Conference on Software Engineering - ICSE'12*. IEEE, 859–869. https://doi.org/10.1109/ICSE.2012.6227133

[39] Stephen Oney and Joel Brandt. 2012. Codelets: Linking Interactive Documentation and Example Code in the Editor. In *Proceedings of the 2012 CHI Conference on Human Factors in Computing Systems - CHI '12*. ACM, 2697. https://doi.org/10.1145/2207676.2208664

[40] Jibin Ou, Martin Vechev, and Otmar Hilliges. 2015. An Interactive System for Data Structure Development. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 3053–3062. https://doi.org/10.1145/2702123.2702319

[41] Clément Pit-Claudel. 2020. Untangling Mechanized Proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 155–174. https://doi.org/10/ghs5sn

[42] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *The Art, Science, and Engineering of Programming* 3, 1 (2018), 1. https://doi.org/10.22152/programming-journal.org/2019/3/1

[43] Paulo Reis, John D Lees-Miller, and Sven Laqua. 2021. Merging SaaS Products In A User-Centered Way — A Case Study of Overleaf and ShareLaTeX. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, 1–8. https://doi.org/10.1145/3411763.3443455

[44] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (2009), 60–67. https://doi.org/10.1145/1592761.1592779

[45] Hugo Romat, Emmanuel Pietriga, Nathalie Henry-Riche, Ken Hinckley, and Caroline Appert. 2019. SpaceInk: Making Space for In-Context Annotations. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, 871–882. https://doi.org/10.1145/3332165.3347934

[46] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24, 9 (1981), 563–573. https://doi.org/10.1145/358746.358755

[47] David Weintrop and Nathan Holbert. 2017. From Blocks to Text and Back: Programming Patterns in a Dual-Modality Environment. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 633–638. https://doi.org/10.1145/3017680.3017707

[48] Jack Williams and Andrew D. Gordon. 2021. Where-Provenance for Bidirectional Editing in Spreadsheets. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–10. https://doi.org/10.1109/VL/HCC51201.2021.9576272

[49] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *Proceedings of the 33rd Symposium on User Interface Software and Technology - UIST'20*. ACM, 152–165. https://doi.org/10.1145/3379337.3415851

[50] Bo Xiong, Haoqi Fan, Kristen Grauman, and Christoph Feichtenhofer. 2021. Multiview Pseudo-Labeling for Semi-Supervised Learning from Video. *arXiv:2104.00682 [cs]* (2021). arXiv:2104.00682 [cs]

[51] Kaiyang Zhou, Yongxin Yang, Yu Qiao, and Tao Xiang. 2021. MixStyle Neural Networks for Domain Generalization and Adaptation. *arXiv:2107.02053 [cs]* (2021). arXiv:2107.02053 [cs]

[52] Oleksandr Zinenko, Cédric Bastoul, and Stéphane Huot. 2015. Manipulating Visualization, Not Codes. In *International Workshop on Polyhedral Compilation Techniques (IMPACT)*. 1–8.

[53] Oleksandr Zinenko, Stephane Huot, and Cedric Bastoul. 2014. Clint: A Direct Manipulation Tool for Parallelizing Compute-Intensive Program Parts. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 109–112. https://doi.org/10.1109/VLHCC.2014.6883031

APPENDICES

A LEVELS OF EXPERTISE

In both the interview and the evaluation studies, we refer to three levels of expertise with LaTeX, defined as follows:

Beginner: I do not feel comfortable when I use LaTeX and I need help to use basic features.

Intermediate: I feel comfortable writing basic documents and I can use the most common features without help.

Expert: I feel comfortable with various kinds of document and I am able to create or customise commands and environments when needed.

B INTERVIEWEES

Table 2 reports the details on each participant we interviewed.

C EVALUATION DATA ANALYSIS

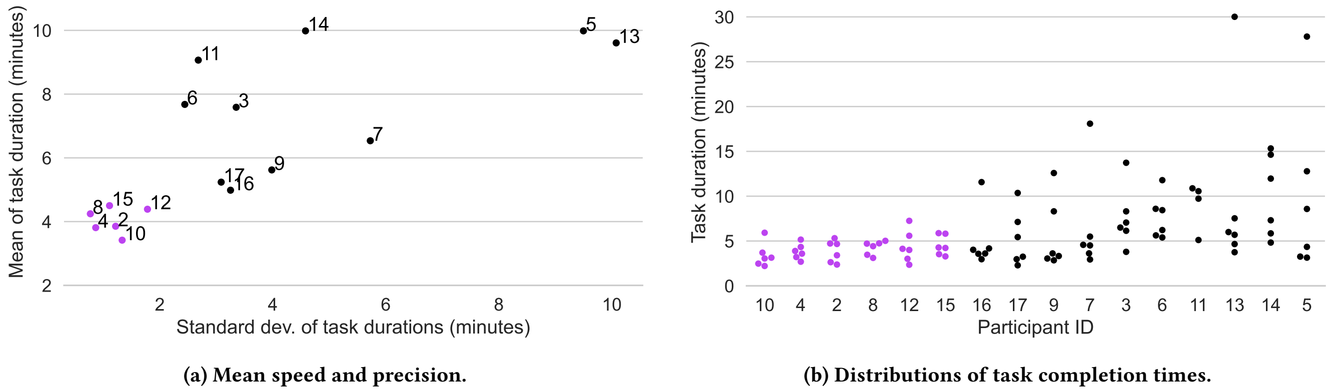
Since both task duration times (*TIME*) and numbers of compilation (*COMPILATIONS*) are strictly positive measures, we tested the log-normality of the distribution of each measure with Kolmogorov's D tests. Task completion times fit log-normal distributions for tasks with tables ($D = 0.127$, $p > 0.15$) and images ($D = 0.151$, $p = 0.06$), but not for tasks with mathematics ($D = 0.173$, $p = 0.02$). Numbers of compilation fit a log-normal distribution for tasks with images ($D = 0.15$, $p = 0.07$), but not for tasks with mathematics ($D = 0.266$, $p < 0.01$) or tables ($D = 0.219$, $p < 0.01$). Given these results, we performed paired t-tests on log-transformed data for task duration time and Wilcoxon signed-rank tests for the numbers of compilations. We report the full results of all tests in Table 3.

D EFFICIENT PARTICIPANTS

In order to look further into the effect of the transitional for mathematics in the controlled experiment, we split participants into two groups based on their efficiency. We define efficiency as a combination of high speed (low task completion times) and high precision (low variance between task completion times), and used those two criteria to discriminate participants as follows.

We plot the mean task completion time of each participant across all tasks in both TRANSITIONALS conditions against the standard deviation of the task completion times. We observe that 6 participants form a distinct cluster in the bottom-left hand corner of that space (Figure 9a). We group the 6 participants from that cluster (mean task completion time shorter than 5 minutes, standard deviation of task completion times lower than 2 minutes) into the *efficient* group (P2, P4, P8, P10, P12, P15, shown in purple), and the 10 others into the *non-efficient* group.

As shown in Figure 9b, overall, all the efficient participants solved the tasks faster than all the non-efficient participant. These groups are also consistent with the self-assessed levels of expertise with LaTeX collected in the post-study questionnaires (Table 4), where mean and median values are higher for efficient participants.



(a) Mean speed and precision.

(b) Distributions of task completion times.

Figure 9: Details on the performance of each participant. Data points in purple correspond to efficient participants. (a) Plot of mean task completion time (Y axis) by standard deviation of task completion times (X axis). Each mark is labelled with the ID of the participant. Efficient participants are faster (low Y value) and more consistent (low X value). (b) Task completion times of each participant (one data point per group of tasks and per condition). Participants are sorted by their mean task completion time over both conditions.

Table 2: Details on the participants of the interviews.

Participant	Occupation	Domain	Expertise	Main presented document
P1	MSc student	Biology	Intermediate	Internship report
P2	PhD student	Data visualisation	Intermediate	PhD thesis
P3	MSc student	Ecology	Intermediate	Biophysics assignment
P4	MSc student	Geology	Intermediate	Hackathon project
P5	Associate prof.	Computer science	Expert	Mathematics paper
P6	MSc student	Complex systems	Intermediate	Thesis proposal
P7	MSc student	Geochemistry	Intermediate	Internship report
P8	MSc student	Archaeology	Beginner	Internship report
P9	MSc student	Computer science	Expert	Computer science paper
P10	High school teacher	Philosophy	Intermediate	Archive of philosophy exams
P11	MSc student	Computer science	Expert	Mathematics class notes

Table 3: Comparisons of task duration times and number of compilations between the two conditions, for each type of task and subset of participants. Effect sizes are reported using Cohen's d and Rank-biserial correlation (RBC). Tests where $p < 0.05$ are marked with asterisks.

(a) Paired t-tests on log-transformed *TIME*.

Type of tasks	Subset	#DoF	t	p	d
	All	15	−1.45	0.17	0.34
Mathematics	Efficient	5	0.61	0.57	0.28
	Non-efficient	9	−2.37	0.04 *	0.64
Tables	All	15	−4.95	< 0.001 *	1.39
Images	All	14	−3.75	0.002 *	1.17

(b) Wilcoxon signed-rank tests on *COMPILATIONS*.

Type of tasks	Subset	W	p	RBC
	All	38	0.97	−0.03
Mathematics	Efficient	5	1.00	0.00
	Non-efficient	16	0.83	−0.11
Tables	All	8	0.008 *	−0.82
Images	All	6	0.001 *	−0.90

Table 4: Statistics on the participants' self-assessed expertise with LaTeX on a 5-point Likert scale (1 = Beginner, 3 = Intermediate, 5 = Expert).

	Mean	Median	Min.	Max.
Efficient participants	3.8	4	3	5
Non-efficient participants	2.8	3	1	5