Représentations intermédiaires interactives pour la manipulation de code LaTeX

Interactive Intermediate Representations for LaTeX Code Manipulation

Camille Gobert
gobert@lri.fr
Université Paris-Saclay, CNRS, Inria, LISN
91400 Orsay, France

Michel Beaudouin-Lafon
mbl@lri.fr
Université Paris-Saclay, CNRS, Inria, LISN
91400 Orsay, France

ABSTRACT

Editing documents written in a description language such as LaTeX is difficult. Few editors improve the experience, and WYSIWYG interfaces for LaTeX are often limited. Yet, LaTeX is still used extensively, especially in the scientific community. We interviewed 11 LaTeX users and performed a thematic analysis of the issues they face. We define the concept of interactive intermediate representations (IIR) and propose to use it to answer some of these problems. IIR constitute a new kind of user interface for document description languages that enable to visualise and manipulate certain pieces of code through suitable representations. We present *i*-LaTeX, a prototype of a LaTeX editor equipped with IIR, and discuss its design, its implementation, its preliminary evaluation and its limits. We conclude on the benefits of IIR for document description language editing.

CCS CONCEPTS

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

KEYWORDS

LaTeX, code editor, intermediate representation

ACM Reference Format:
Camille Gobert and Michel Beaudouin-Lafon. 2021. Représentations intermédiaires interactives pour la manipulation de code LaTeX: Interactive Intermediate Representations for LaTeX Code Manipulation. In *32e Conférence Francophone sur l'Interaction Homme-Machine (IHM '20.21), April 13–16, 2021, Virtual Event, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3450522.3451325

RÉSUMÉ

L'édition de documents rédigés dans un langage de description tel que LaTeX est un processus difficile. Peu d'éditeurs améliorent cette expérience, et les interfaces WYSIWYG existantes sont souvent limitées. Pourtant, LaTeX demeure largement utilisé, notamment dans le milieu scientifique. Nous avons interviewé 11 utilisateurs de LaTeX et réalisé une analyse thématique des problèmes qu'ils rencontrent. Nous définissons le concept de représentations intermédiaires interactives (RII) et proposons de l'utiliser pour répondre à certains de ces problèmes. Les RII constituent un nouveau genre d'interface utilisateur pour les langages de description de document qui permet de visualiser et de manipuler certains morceaux de code à travers des représentations adaptées. Nous présentons *i*-LaTeX, un prototype d'éditeur LaTeX doté de RII, et discutons son design, son implémentation, son évaluation préliminaire et ses limites. Nous concluons sur les avantages des RII pour l'édition de langages de description de documents.

MOTS-CLÉS

LaTeX, éditeur de code, représentation intermédiaire

1 INTRODUCTION

Les systèmes de préparation de documents sont des outils numériques qui peuvent être utilisés pour créer et modifier des documents textuels. Ils sont généralement divisés en deux catégories [16] : les langages de description de document (*e.g.* Markdown, Asciidoc, LaTeX, HTML), et les systèmes WYSIWYG (*e.g.* Microsoft Word, Apple Pages, LibreOffice Writer). Les langages de description requièrent de coder le document d'une certaine manière, en utilisant un langage particulier (typiquement à base de commandes ou de balises), afin de décrire comment le système doit interpréter le document source afin de générer le document final. À l'inverse, les systèmes WYSIWYG permettent aux utilisateurs de voir à tout moment à quoi ressemble le document final et de modifier directement son contenu ou son apparence, sans passer par un langage intermédiaire. Aucune de ces deux approches ne semble en mesure de supplanter l'autre : les éditeurs de code ne semblent pas adaptés pour visualiser et manipuler le contenu structuré et les abstractions couramment utilisées, et aucune interface graphique ne semble pouvoir capturer l'expressivité d'un langage aussi puissant que LaTeX.

Nous proposons de les combiner à l'aide de *représentations intermédiaires interactives* (RII) de certains morceaux de code, qui permettent de les visualiser et de les manipuler à travers des représentations plus adaptées. Nous définissons une RII comme une représentation alternative d'un code textuel possédant deux caractéristiques : elle doit être *intermédiaire*, c'est-à-dire conceptuellement située *entre* le code et le document final ; et elle doit être *interactive*, c'est-à-dire permettre de modifier le code auquel elle est associée. Bien que cette idée ne soit pas nouvelle, elle n'a à notre connaissance jamais été clairement nommée, définie, ni appliquée au domaine des langages de description de documents.

Ce nouveau genre d'interface utilisateur pour éditeur de langages de description de document se distingue des interfaces WYSIWYG, car nous défendons que manipuler directement le document final

IHM '20.21, April 13–16, 2021, Virtual Event, France
© 2021 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 *32e Conférence Francophone sur l'Interaction Homme-Machine (IHM '20.21), April 13–16, 2021, Virtual Event, France*, https://doi.org/10.1145/3450522.3451325.

n'est pas systématiquement mieux que manipuler son code source. Plutôt que de restreindre les utilisateurs à utiliser l'un ou l'autre de ces substrats de document [3], nous proposons plutôt d'en introduire de nouveaux et de laisser le choix aux utilisateurs en fonction de leurs besoins. En utilisant un éditeur doté de RII, un utilisateur pourrait par exemple choisir de visualiser un tableau sous forme de grille lors de sa conception ou pour réordonner certaines colonnes par manipulation directe, tout en conservant la possibilité d'éditer directement le code des commandes utilisées dans les cellules.

Afin de mieux comprendre les besoins auxquels pourraient répondre les RII, nous avons interviewé 11 utilisateurs de LaTeX et réalisé une analyse thématique des problèmes que ceux-ci rencontrent. Nous avons ensuite développé *i*-LaTeX, un prototype d'éditeur LaTeX doté de RII. Celui-ci permet d'interagir avec certains éléments du PDF produit par LaTeX afin d'afficher des représentations intermédiaires interactives du code qui les a générées. Notre contribution est donc à la fois empirique, théorique et technique. Nous commençons par présenter d'autres travaux en lien avec le concept de RII ; nous poursuivons en détaillant la méthodologie et les résultats de notre analyse thématique ; nous introduisons *i*-LaTeX et discutons son design, son implémentation, son évaluation préliminaire et ses limites ; et nous concluons sur la pertinence des RII pour les éditeurs de langages de description de document.

2 TRAVAUX CONNEXES

Nous présentons les divers travaux ayant inspiré le concept de RII et le prototype *i*-LaTeX : les documents composés d'éléments interactifs ou programmables ; les langages, paradigmes et environnements de programmation non-textuels ; et les représentations intermédiaires utilisées dans d'autres domaines.

2.1 Éditeurs de documents LaTeX

LaTeX est un langage de description de document créé par Leslie Lamport en 1984 [22] qui repose sur TeX, un système de composition de document développé par Donald Knuth à la fin des années 1970 [20]. Aujourd'hui, il demeure un choix courant pour la rédaction de publications académiques et techniques, et ce en dépit de la difficulté à l'utiliser — y compris pour des utilisateurs experts, qui peuvent « subir une perte de productivité lorsque LaTeX est utilisé, par rapport à d'autres systèmes de préparation de documents » [18]. Étant donné que LaTeX est un langage de commandes, la plupart des éditeurs de documents LaTeX ressemblent plus à des éditeurs de code qu'à des éditeurs WYSIWYG — incluant des fonctionnalités telles que la coloration syntaxique et l'auto-complétion. Certains éditeurs LaTeX essaient d'offrir une expérience se rapprochant de celle des éditeurs WYSIWYG ; mais à notre connaissance, soit ils ne fournissent qu'un formatage de base du code source (*e.g.* AUCTeX[1], le mode *rich text* d'Overleaf[2]), soit ils fournissent une interface entièrement WYSIWYG qui masque le code et ne prend en charge qu'un ensemble limité de fonctionnalités (*e.g.* Compositor[3]). Les éditeurs LyX[4] et TeXmacs[5] proposent de représenter un document entier sous une forme graphique intermédiaire mettant en avant sa structure et son contenu plutôt que son rendu final — un paradigme souvent appelé *What You See Is What You Mean*. Néanmoins, à notre connaissance, aucun éditeur de ce type ne permet de manipuler directement le code LaTeX (au delà de l'import/export au format LaTeX et de l'insertion de courts morceaux de code, *e.g.* une formule mathématique écrite en LaTeX). Plusieurs solutions permettent de pré-visualiser le rendu de certains éléments isolés (*e.g.* une formule mathématique, une figure), d'animer la génération du document (*e.g.* Gliimpse [8]), ou encore d'associer à chaque élément du PDF généré une position dans le code, et inversement (*e.g.* SyncTeX [23]) ; mais aucune d'entre elles ne permet de modifier le code plus facilement.

i-LaTeX se distingue des éditeurs de documents LaTeX existants en offrant un accès immédiat et complet au code LaTeX et au PDF généré (à la façon des éditeurs de code comme TeXstudio) tout en permettant de visualiser et de manipuler le code de certains éléments à travers des représentations plus adaptées plus ou moins similaires à celles imposées par des éditeurs comme LyX et Compositor.

2.2 Documents interactifs

Dans son essai de 2011, *Explorable Explanations*[6], Bret Victor plaide en faveur de documents plus interactifs, dans lesquels les utilisateurs pourraient dynamiquement modifier des valeurs et observer comment d'autres informations évoluent. Ce principe est repris par les multivers de Dragicevic *et al.* [9], qui démontrent l'intérêt de permettre aux lecteurs d'explorer différentes façons d'analyser et de visualiser des données expérimentales. Afin de rendre la conception de documents interactifs plus accessible aux non-programmeurs, Conlen *et al.* présentent Idyll [7], un langage de balises conçu pour les journalistes. Afin d'outrepasser le manque d'interactivité lié aux limites de certains formats de fichiers, Chameleon [25] propose d'identifier certains contenus sur l'écran à l'aide de techniques de vision par ordinateur afin d'y surimprimer des versions animées ou interactives du même contenu.

En permettant de modifier le code source des documents en interagissant avec le PDF, *i*-LaTeX exploite la notion de document interactif. Cependant, contrairement aux travaux présentés ci-dessus, il ne s'agit pas de produire un document PDF interactif pour les *lecteurs*, mais de rendre son édition plus interactive pour les *auteurs*.

2.3 Programmation de documents

L'idée de mélanger rédaction de documents et programmation est apparue avec le développement de nouveaux langages de description de document dans les années 1970. Elle s'est poursuivie par la promotion de concepts tels que le *literate programming* [19] de Knuth et la conception de langages et de paradigmes de programmation centrés sur l'humain (tels que Smalltalk [11]) dans les années 1980. Cette idée a depuis été remise au goût du jour par la popularité croissante des *computational notebooks* comme Jupyter[7], largement adoptés par des communautés de sciences des données. Les *notebooks* permettent de faire cohabiter des cellules dont le contenu est rédigé par les auteurs (texte, code) avec des cellules dont le contenu est généré par l'exécution d'un programme. Bien que ces dernières soient traditionnellement statiques, Kery *et al.* [17] présentent plusieurs pistes pour rendre celles-ci plus interactives — par exemple

[1] https://www.gnu.org/software/auctex/
[2] https://www.overleaf.com
[3] https://compositorapp.com/
[4] https://www.lyx.org/
[5] https://www.texmacs.org

[6] http://worrydream.com/ExplorableExplanations/
[7] https://jupyter.org/

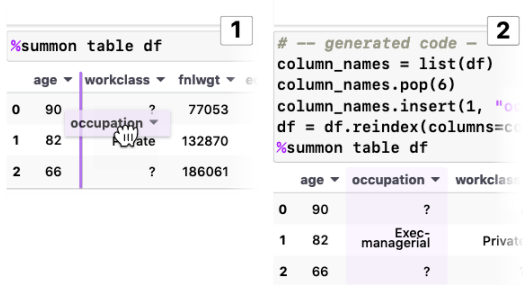


Figure 1: Déplacer une colonne du tableau génère le code qui implémente la modification (adapté de Kery *et al.* [17]).

pour les tableaux de données (Figure 1). Wrex [10] permet de synthétiser le code permettant d'insérer et de remplir une nouvelle colonne à partir d'un ou plusieurs exemples fournis interactivement par les utilisateurs via la représentation d'un tableau de données.

i-LaTeX inclut un éditeur de code et permet donc naturellement d'utiliser toutes les fonctionnalités de programmation de document offertes par LaTeX. Les représentations intermédiaires qu'il inclut constituent un moyen supplémentaire de visualiser et de manipuler le code, au même titre que celles développées pour les *notebooks* sus-mentionnés.

2.4 Programmation non-textuelle

Depuis les années 1980, des langages de programmation visuels particulièrement adaptés au traitement de signaux multimédia tels que Max[8] et Pure Data[9] ont été développés. La syntaxe graphique simple et la manipulation directe des blocs qui composent les programmes rendent ces langages particulièrement accessibles à des utilisateurs sans formation en programmation. Cela pourrait expliquer pourquoi ce type de langage continue d'être utilisé aujourd'hui. Il a par exemple été choisi pour concevoir Dynamic Brushes [15], un système de programmation du comportement d'un pinceau numérique destiné à des illustrateurs. D'autres langages visuels ont également été largement adoptés à des fins éducatives (*e.g.* Scratch [27]). Néanmoins, certains chercheurs affirment qu'ils pourraient également être utiles pour des professionnels [2]. Dans cette optique, Droplet[10] et les *frame-based editors* [5] illustrent deux façons de faire fusionner la richesse des langages textuels avec la manipulation directe des langages à blocs.

La programmation non-textuelle peut également prendre la forme de nouveaux genres d'éditeurs de code. Plusieurs travaux proposent d'aider les programmeurs à visualiser l'état courant du programme en augmentant des éditeurs de code traditionnels avec des données brutes [24] ou visualisées [14]. Python Tutor [12] affiche par exemple les structures de données utilisées sous une forme schématique à des fins éducatives. D'autres travaux portent plutôt sur de nouvelles façons d'interagir avec le code. Sketch-n-Sketch [13] est un éditeur expérimental permettant de programmer une image vectorielle en utilisant un langage fonctionnel et de modifier son code en interagissant directement avec le SVG généré. Il illustre l'idée du *output-directed programming* de Chugh [6], qui consiste à éditer le code source d'un programme en manipulant directement sa sortie graphique.

Les représentations intermédiaires de *i*-LaTeX sont inspirées de ces différentes façons de penser le code autrement que comme du texte brut — et plus particulièrement de l'approche de Sketch-n-Sketch. Ces représentations permettent de s'affranchir de la compilation du document pour visualiser le moindre changement tout en permettant de donner forme à des abstractions invisibles dans le code et effacées dans le document généré.

2.5 Représentations intermédiaires

La représentation la plus appropriée du code d'un programme ou d'une description de document peut se situer entre le code brut et la sortie du programme : on parle alors de représentation *intermédiaire*. Si celle-ci permet de modifier le code qu'elle représente, elle est de surcroît *interactive* (Figure 2). Barista [21], Envision [1] et JetBrains MPS[11] sont différents éditeurs de code capables d'afficher certains éléments de langages textuels sous une forme structurée (*e.g.* organiser le code sous forme d'un tableau) ou graphique (*e.g.* dessiner les symboles d'une expression mathématique). Graphite [26] permet de générer le code d'une couleur à partir d'une palette interactive, tandis que Clint [29] permet de modifier le code de boucles imbriquées afin d'améliorer les performances d'un programme à travers la manipulation de diagramme interactifs.

i-LaTeX a pour but de fournir des représentations intermédiaires de morceaux de code LaTeX qui gagnent à pouvoir être visualisés et manipulés autrement que sous forme textuelle, de la même manière que des systèmes tels que Barista et Clint le font pour des langages de programmation et des paradigmes plus traditionnels. En outre, *i*-LaTeX permet d'afficher ces visualisations à proximité des éléments correspondants du PDF généré, renforçant ainsi les liens entre les deux ; tandis que les exemples donnés ci-dessus les affichent généralement uniquement à proximité du code concerné.

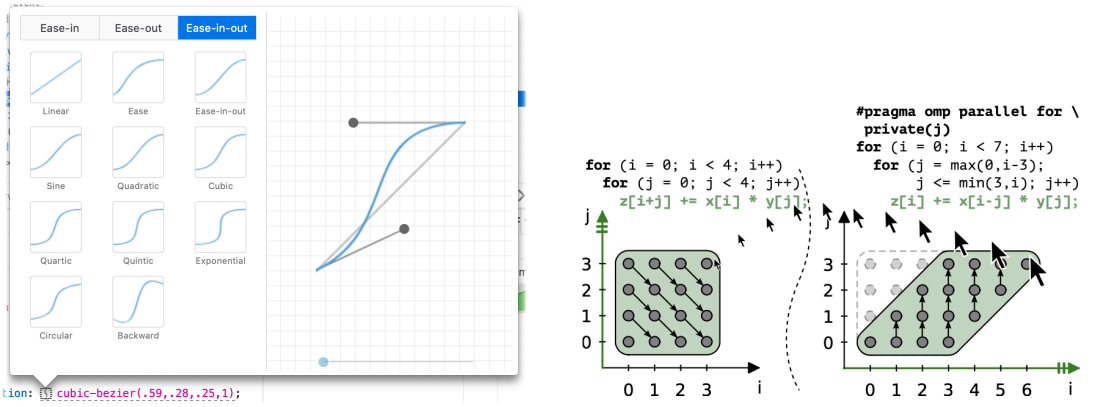
3 INTERVIEWS D'UTILISATEURS DE LaTeX

Afin de mieux comprendre de quelles façons les RII pourraient améliorer l'édition de langages de description de document, nous avons interviewé 11 utilisateurs de LaTeX et réalisé une analyse thématique des difficultés que ceux-ci rencontrent. Dans cette section, nous présentons la méthodologie utilisée et les résultats obtenus. À notre connaissance, aucune étude similaire n'a déjà été publiée.

3.1 Méthodologie

3.1.1 Participants. Nous avons interviewé 11 participants (5 femmes et 6 hommes, âgés de 21 à 40 ans). La plupart sont des étudiants de Master (8/11) ou des personnes issues du milieu universitaire, spécialisés dans des domaines très divers. Les participants ont été recrutés via un mailing interne au laboratoire et un message sur un groupe Facebook d'une grande école. Ils n'ont reçu aucune compensation pour leur participation. Chacun d'entre eux avait utilisé LaTeX au cours des semaines ou des mois précédant leur interview. Les participants ont l'habitude d'utiliser Overleaf (5/11) ou Texmaker (5/11) pour rédiger des documents LaTeX, mais plusieurs d'entre eux ont également utilisé ou essayé au moins un

[8] https://cycling74.com/products/max
[9] https://puredata.info/
[10] http://droplet-editor.github.io/
[11] https://www.jetbrains.com/mps/



(a) Éditeur d'animation de Mozilla Firefox

(b) Éditeur de boucles imbriquées de Clint (d'après Zinenko *et al.* [28])

Figure 2: Exemples de représentations intermédiaires interactives. (a) La manipulation des poignées pour modifier la courbe met à jour les paramètres numériques de la propriété CSS (affichée en dessous). (b) L'inclinaison du polygone transforme le code C++ (affiché au dessus) pour refléter la façon dont les opérations doivent être parallélisées selon le nouveau diagramme.

éditeur alternatif parmi TeXstudio, TeXworks, Kyle et Gedit — sur Windows, MacOS ou Linux. En outre, la majorité d'entre eux utilise également des logiciels de traitement de texte. Leur expertise avec LaTeX est très variable[12], mais la plupart des participants (8/11) sont des utilisateurs intermédiaires. Des informations supplémentaires sur chaque participant peuvent être trouvés dans la Table 1.

3.1.2 Procédure. Toutes les interviews étaient semi-structurées, menées en français, et d'une durée moyenne d'une heure environ (entre 29 et 99 minutes). La durée des interviews et le choix des questions ont été ajustés à l'aide de trois interviews pilotes de collègues doctorants et post-doctorants.

Nous avons commencé chaque interview en demandant à chaque participant de nous montrer le dernier document LaTeX sur lequel il avait travaillé via un partage d'écran (comprenant à la fois le code LaTeX et le PDF généré). Nous leur avons tout d'abord posé plusieurs questions relatives au document présenté (*e.g.* type de document, éditeur utilisé, etc). Nous leur avons ensuite demandé de nous décrire les différents problèmes auxquels ils avaient fait face (qu'ils les aient résolus ou non) en se référant le plus possible au code du document (dans le double but de les aider à s'en souvenir et de rendre leurs explications plus claires pour nous). Les interviews étant semi-structurées, nous avons utilisé une liste de questions prédéfinies pour guider les participants, mais ceux-ci étaient invités à parler librement de chaque problème rencontré. Les participants étaient également invités à nous parler de problèmes rencontrés sur d'autres documents si cela leur semblait pertinent. Nous avons conclu chaque interview avec quelques questions plus générales sur l'expérience de chaque participant avec LaTeX.

[12]Les niveaux d'expertise sont définis de la façon suivante : les *débutants* ne se sentent pas à l'aise avec LaTeX et ont besoin d'aide pour utiliser la plupart des fonctionnalités ; les utilisateurs *intermédiaires* se sentent à l'aise pour écrire des documents simples avec quelques structures et mettre le texte en forme ; et les utilisateurs *avancés* se sentent à l'aise pour inclure du contenu varié et créer leurs propres commandes. Chaque participant a été invité à auto-évaluer son niveau d'expertise, que nous avons ensuite adapté en fonction des fonctionnalités avec lesquelles il se sentait à l'aise ou non.

3.1.3 Données récoltées. Toutes les interviews ont été enregistrées (audio et écran partagé) et retranscrites manuellement, incluant de courtes descriptions du contenu affiché à l'écran ou des interactions des participants lorsque cela était pertinent. Nous avons également pris des notes manuscrites synthétisant les problèmes principaux et les solutions utilisées.

3.1.4 Analyse des données. Nous avons réalisé une analyse thématique [4] à partir des données récoltées. Le premier auteur a tout d'abord généré plus de 650 codes à partir des huit premières transcriptions, qu'il a ensuite classés dans près de 80 sous-thèmes. Nous nous sommes alors servis de ces sous-thèmes afin de générer trois jeux de thèmes : un premier jeu regroupant les sous-thèmes similaires (*i.e.* des *domain summaries*, dans le vocabulaire de Braun et Clarke) ; un second jeu regroupant les problèmes similaires (indépendamment du contexte dans lequel ils sont apparus) ; et un troisième jeu issu d'un mélange des deux premiers.

Ce dernier jeu a été élaboré à l'aide de discussions avec plusieurs chercheurs de l'équipe dans le but d'identifier les groupes de sous-thèmes les plus pertinents dans les deux premiers thèmes. Nous avons également réécouté les trois interviews non-transcrites afin de nous assurer de leur inscription dans les cinq thèmes finaux, et nous en avons extrait plusieurs citations afin de renforcer et de contraster ce qui avait déjà été mis en évidence par les huit premières interviews. Nous présentons le contenu des cinq thèmes finaux ci-dessous.

3.2 Résultats

3.2.1 T1 — Le code source doit être accessible et modifiable.

Les interfaces WYSIWYG sont limitées. En dépit des difficultés que cela induit, la plupart des participants semblent à l'aise avec le paradigme consistant à décrire leur document dans un langage particulier ; et peu ont évoqué le désir d'un éditeur WYSIWYG pour LaTeX. P2 craint par exemple que déplacer des morceaux de texte dans Microsoft Word « *[fasse] du code dégueu***** derrière* »,

Participant	Profession	Domaine	Expertise	Principal document présenté
P1	Étudiante en Master	Biologie	Intermédiaire	Rapport de stage
P2	Doctorante	Visualisation de données	Intermédiaire	Thèse de doctorat
P3	Étudiant en Master	Écologie	Intermédiaire	Devoir à rendre
P4	Étudiante en Master	Géologie	Intermédiaire	Projet de hackathon
P5	Enseignant-chercheur	Informatique	Avancé	Article de mathématiques
P6	Étudiant en Master	Systèmes complexes	Intermédiaire	Sujet de thèse
P7	Étudiante en Master	Géochimie	Intermédiaire	Rapport de stage
P8	Étudiante en Master	Archéologie	Débutant	Rapport de stage
P9	Étudiant en Master	Informatique	Intermédiaire	Article d'informatique
P10	Professeur de lycée	Philosophie	Intermédiaire	Liste d'examens de philo.
P11	Étudiant en Master	Informatique	Avancé	Notes de cours de math.

Table 1: Détails sur les participants interviewés.

tandis que P4 a expliqué se sentir « *impuissante* » lorsqu'elle ne peut pas obtenir ce qu'elle veut dans un éditeur WYSIWYG. À l'exception de P7, aucun de ceux qui connaissaient le mode *rich text* d'Overleaf ne l'utilisent. Plusieurs participants lui reprochent de manquer de fonctionnalités en cachant le code, impliquant ainsi « *plus de manipulation, plus de mouvement sur l'écran* » (P8) afin de régulièrement passer d'un mode à l'autre. P7 a d'ailleurs reconnu qu'elle l'avait uniquement utilisé pour prendre des notes pendant un cours d'histoire : « *il fallait juste taper ; j'avais pas de maths à faire ; j'avais pas de photo à rajouter* » (P7).

Le document doit pouvoir être programmé. Plusieurs participants parmi les plus experts ont expliqué qu'ils aimaient pouvoir créer des commandes personnalisées pour générer ou réutiliser du contenu (*e.g.* développer des abréviations, réutiliser un dessin paramétré). Au fil des années, P11 s'est par exemple constitué un long préambule contenant plusieurs dizaines de commandes personnalisées pour générer des expressions mathématiques (telles que les noms de théorèmes et des constructions courantes), qu'il continue d'utiliser. P10 explique avoir choisi LaTeX pour créer une archive unique de centaines d'examens de philosophie de lycée, car il souhaitait pouvoir générer plusieurs index (par auteur, par thème, par type de question philosophique, etc) qui seraient automatiquement mis à jour s'il ajoutait de nouveaux examens par la suite. Selon lui, un tel niveau d'automation est impossible à atteindre avec un système de préparation de document de type WYSIWYG : « *sous LibreOffice, on ne peut avoir un index alphabétique que pour une seule catégorie* » (P10). La nature textuelle de LaTeX permet également de générer des documents de toute pièce facilement. P5 a ainsi écrit un programme Python pour générer un code LaTeX représentant plusieurs centaines de badges similaires pour un concours de Go : « *Si vous me demandez de vous faire deux cent badges en Word, vous vous démer*** ; je le fais pas* » (P5).

Les commentaires ont de multiples utilités. Seule une fraction des participants a déclaré utiliser des commentaires dans le code ; mais ceux qui s'en servent expliquent que leur utilité est multiple. Les raisons de s'en servir incluent (1) se rappeler de l'utilité d'un paquet ou d'une commande ; (2) discuter avec d'autres co-auteurs ; (3) conserver des morceaux de code inutilisés qui pourraient servir plus tard ; (4) commenter le code afin de trouver la source d'une erreur ; (5) réutiliser des extraits d'un ancien morceau de code pour en écrire une nouvelle version ; ou encore (6) planifier le contenu à écrire dans chaque partie du document.

L'accès au code facilite la réutilisation. P5 a expliqué que le fait d'avoir accès au code permettait de copier-coller immédiatement des solutions trouvées sur internet, sans avoir à reproduire les étapes énumérées dans un tutoriel destiné aux éditeurs WYSIWYG. Cet accès lui permet également de réutiliser le code LaTeX de ses articles scientifiques afin de créer des présentations pour des colloques.

3.2.2 *T2 — Apprendre LaTeX est compliqué et chronophage.*

LaTeX est difficile à apprendre. La plupart des participants ont expliqué qu'ils attendaient généralement de rencontrer des problèmes pour se renseigner sur LaTeX et ses fonctionnalités. Plusieurs ont souligné que cela arrivait souvent dans des moments de hâte, peu avant un rendu — ne leur laissant donc pas la possibilité de prendre le temps de comprendre le code qu'ils devaient utiliser (après l'avoir copié-collé) : « *Pour l'instant je suis plus dans l'optique que ça doit marcher* » (P2). Le nom des commandes, l'ordre de leurs paramètres, et les valeurs autorisées pour ces derniers semblent souvent être oubliés ou ignorés par les participants — y compris lorsqu'ils ont déjà l'habitude de s'en servir : « *Je saurais qu'il faut une minipage, mais je serais quand même obligé d'aller chercher quelque part comment l'écrire* » (P3).

Ne pas comprendre LaTeX est coûteux. Certains participants ont reconnu que leur incompréhension de LaTeX était parfois coûteuse : « *[Personnaliser] la biblio, c'est le truc qui était le plus long, parce que j'ai pas trouvé de solution toute faite quoi. Il a fallu que j'invente un petit peu [...]* » (P7). P2 et P4 ont exprimé des regrets de n'avoir jamais eu de cours sur LaTeX avant de devoir l'utiliser : « *J'aurais voulu prendre une journée [pour apprendre LaTeX] ; il y a des formations qui présentent les bases* » (P2). P6 a au contraire expliqué qu'il ne voudrait pas faire un tel effort : « *Tu trouves un document qui marche, tu copies-colle, et tu changes par itération [...] je vais pas apprendre la structure du code.* » (P6)

Les recherches portent sur des problèmes spécifiques. Les participants ont indiqué qu'ils cherchaient souvent des solutions à des problèmes spécifiques : produire un symbole particulier, comprendre

une erreur, trouver la documentation d'un paquet, etc. Lorsqu'il a eu besoin de poursuivre la numérotation d'une liste, P3 a par exemple rapidement trouvé la solution sur internet : « *j'ai trouvé que c'était après le* `\begin{enumerate}` *qu'il fallait mettre le start, égal, ce qu'on veut* » (P3). Néanmoins, trouver une réponse ne semble pas toujours être évident pour tous les participants. Afin de trouver une solution pour insérer du grec ancien dans son document LaTeX, P8 a par exemple expliqué avoir « *plutôt cherché sur des trucs de linguistique et d'historien* » (P8) — où elle n'a pas pu trouver de réponse.

3.2.3 T3 — Les éditeurs textuels sont inadaptés au contenu structuré.

Les éditeurs de code ignorent les structures. De nombreux participants se sont plaints de la difficulté de décrire des éléments structurés tels que des tableaux, des sous-figures, ou même des formules chimiques. La création de tableaux a souvent été décrite comme « *vachement chi**** » (P1), et la difficulté de leur édition leur a été reprochée : « *ça m'est arrivé de devoir déplacer des colonnes […] ça se fait, mais c'est juste que ça prend du temps* » (P3). Selon P7, à cause de la syntaxe des tableaux, « *oublier une colonne c'est un enfer* » car « *il faut que je rajoute des trucs et qu'à chaque ligne je compte, je me mette au bon endroit* » (P7). Ces observations s'appliquent également à d'autre types de contenu structuré, en particulier lorsque les participants ne sont pas familiers avec un paquet offrant une solution clé en main : « *L'autre truc qui est vraiment difficile aussi, c'est de mettre les images côte à côte* » (P4).

Le recours aux outils externes. En raison des difficultés à créer et modifier du contenu structuré depuis leurs éditeurs LaTeX, plusieurs participants se tournent vers des outils externes. P2 et P4 ont par exemple toutes deux commencé par utiliser des sous-figures constituées d'images séparées dans LaTeX avant de finalement les fusionner en utilisant Inkscape ou Adobe Illustrator car « *comme ça je maîtrisais la mise en page* » (P2). Afin de créer des tableaux à partir de classeurs Excel, P7 les a exportés au format CSV afin de générer le code LaTeX correspondant en utilisant le convertisseur en ligne tablesgenerator.com. Elle a cependant expliqué avoir dû répéter le processus à chaque fois qu'elle souhaitait modifier les tableaux, dont le code généré était trop difficile à interpréter sous forme de texte brut. D'autres participants ont par ailleurs indiqué qu'ils utilisaient ce même site web pour créer des tableaux LaTeX à partir de zéro. P7 s'est également plainte de ne pas pouvoir exporter les structures créées dans des logiciels spécialisés tels que Microsoft Excel (pour les tableaux) ou ChemDraw (pour les molécules) sous forme de code LaTeX — soulignant là encore à quel point les éditeurs de code sont inadaptés pour décrire et manipuler certaines structures : « *un truc qui serait utile c'est de dessiner ma molécule de A à Z et que le code s'affiche à côté quoi* », car « *le truc le plus difficile que j'ai jamais utilisé sous LaTeX c'est le package pour faire les molécules chimiques* » (P7).

3.2.4 T4 — Les abstractions sont difficiles à visualiser et à formaliser.

Les dimensions. De nombreuses commandes LaTeX requièrent de spécifier des dimensions, comme par exemple la taille d'une image ou les marges de certains éléments. Or, selon plusieurs participants, les dimensions de LaTeX souffrent de deux problèmes : il est difficile de quantifier une longueur que l'on imagine, et il est difficile d'imaginer une longueur quantifiée. « *Je ne sais pas forcément quelle taille précise je veux que l'image ait, mais [je sais] que je veux qu'elle ait cette taille-là dans ma tête* » (P8). Comme de nombreux autres participants, P1 s'est aussi plainte de devoir essayer plusieurs dimensions jusqu'à trouver celle qui convient — une stratégie parfois coûteuse : « *quand tu veux que [tes figures] soient un peu plus grosses, un peu plus petites, tu compiles quinze mille fois et c'est chi*** […] surtout quand tu as beaucoup de figures, et que du coup il met beaucoup de temps à compiler* » (P1). La plupart des participants semblent donc favorables à une forme d'interaction plus directe avec les dimensions. Afin d'outrepasser ces difficultés, certains participants ont fait le choix de détourner des commandes avec lesquelles ils se sentent plus familiers. P3 a ainsi inséré du texte de couleur blanche dans son document afin de sauter « *artificiellement* » plusieurs lignes successives : « *je pense qu'il y avait une façon plus simple de le faire, mais comme j'étais assez pressé c'est comme ça* » (P3).

Les positions. Au contraire des dimensions, LaTeX se charge traditionnellement de calculer les positions des éléments à la place de l'utilisateur. Cependant, selon certains participants, ce manque de contrôle n'est pas toujours souhaitable. P1 a par exemple expliqué que sa façon de positionner les figures n'était « *pas très rationnelle* » (P1), et P2 s'est plainte de la difficulté de placer une image suffisamment près d'un paragraphe y faisant référence lorsque LaTeX en décidait autrement. Ces difficultés sont probablement renforcées par le fait que plusieurs participants ne sont pas conscients du sens des paramètres de positionnement qu'ils utilisent (*e.g.* avec l'environnement `figure`) — souvent copiés avec le reste d'un morceau de code. Par conséquent, P4 — qui a pourtant lu « *comment le compilateur faisait pour placer les images* » afin de gagner en contrôle sur leur positionnement — a ainsi admis avoir dû revoir ses attentes concernant les positions des figures à la baisse. Ces difficultés concernent également les positions plus locales. Bien qu'il se sente à l'aise avec les commandes de dessin du paquet TikZ, P5 a expliqué qu'il aimerait bien pouvoir déplacer certains éléments de ses dessins au lieu « *d'essayer de deviner* » les bonnes coordonnées ou de faire « *vaguement de la trigo* » pour les calculer.

3.2.5 T5 — La dualité entre le code et le PDF est coûteuse.

Identifier le code d'un élément du PDF est difficile. Certains participants se sont plaints de la difficulté de faire le lien entre un élément du PDF et le code l'ayant généré. P5 et P9 ont tous deux du mal à (1) localiser le code d'une formule mathématique affichée dans le PDF et (2) trouver le symbole qu'ils veulent corriger à l'intérieur du code de celle-ci. Afin de résoudre le premier problème, ils recherchent souvent un extrait du texte voisin de la formule dans leurs éditeurs de code, bien que P5 ait admis que cette technique échoue régulièrement. Cette approche n'est pas sans lien avec l'absence de support de SyncTeX dans l'éditeur LaTeX que P5 utilise. Il a d'ailleurs expliqué qu'il aimerait en changer pour cette raison — soulignant l'importance de ce genre d'outil. En revanche, ni P5 ni P9 n'ont trouvé de solution au second problème : pour eux, modifier une formule mathématique implique systématiquement « *de relire le code LaTeX sans relire le PDF* » (P9).

Compiler est un processus lent. De nombreux participants se sont plaints du temps requis pour compiler leurs documents LaTeX. Essayer plusieurs alternatives (*e.g.* différentes tailles pour une image) est ainsi coûteux, et les erreurs font perdre du temps : « *si on se trompe ça nous coûte quand même une minute* » (P5). Certains participants ont élaboré des stratégies en conséquence. Afin de compiler moins souvent, P1 a expliqué qu'elle distinguait rédaction et mise en forme : « *quand je suis dans une phase de rédaction [...] je tape juste et j'écris [...] mais quand je fais la forme j'ouvre toujours [l'éditeur de code et le PDF] à côté, pour compiler régulièrement* » (P1). P5 a également donné l'exemple de la mise en cache d'images crées avec TikZ : « *tant qu'il n'y a pas de modification faite sur l'image, [l'outil] réinjecte l'image à la place du code TikZ* » (P5).

3.3 Recommandations pour le design

Cette analyse thématique révèle la variété des problèmes auxquels les utilisateurs de LaTeX font face et suggère plusieurs opportunités pour améliorer le design des éditeurs de documents LaTeX. Nous résumons celles-ci dans les quatre recommandations suivantes.

Conserver l'accès au code. Nous considérons qu'aucune interface graphique ne peut complètement dissimuler le code d'un document LaTeX sans brider ses utilisateurs (*e.g.* en ne supportant pas certaines fonctionnalités). Nous recommandons donc de permettre aux utilisateurs de LaTeX de lire et de modifier librement et facilement le code de leurs documents. De plus, certains utilisateurs veulent pouvoir réutiliser du code facilement ou automatiser sa génération.

Cibler les problèmes spécifiques et courants. Même si la majorité des participants n'était pas fermée à l'idée d'apprendre à utiliser LaTeX, aucun d'entre eux ne peut ou ne veut y consacrer beaucoup de temps. Nous recommandons donc de ne pas restreindre les éditeurs LaTeX à des fonctionnalités génériques plutôt destinées aux experts (*e.g.* l'auto-complétion de commandes, qu'il faut donc connaître), mais également de prendre en compte les problèmes et les besoins plus spécifiques mais courants (*e.g.* tableaux, sous-figures, références) et d'aider les utilisateurs moins expérimentés et intermittents à trouver ou concevoir des solutions à leurs problèmes.

Donner forme aux abstractions invisibles. Étant donné la nature entièrement textuelle de LaTeX, nous recommandons d'aider les utilisateurs à (1) visualiser les structures et les abstractions codées sous forme de texte lorsqu'elles s'y prêtent et (2) manipuler ces dernières à l'aide d'interactions adaptées tout en maintenant la synchronisation du texte source. Ces visualisations doivent viser à réduire le nombre de cycles de compilation en facilitant la constitution de représentations mentales du document.

Renforcer les liens entre code source et document généré. Face à la dualité entre code et rendu d'un même document, nous recommandons d'améliorer la visibilité et la spécificité des liens qui unissent ces deux substrats (*e.g.* faciliter l'identification et la modification de la commande responsable d'un symbole mathématique particulier dans le PDF).

Les représentations intermédiaires interactives nous semblent pouvoir répondre à certains des problèmes soulevés par cette analyse thématique en respectant les recommandations que nous proposons ci-dessus. Nous avons ainsi développé *i*-LaTeX, un prototype d'éditeur LaTeX doté de RII, que nous présentons dans la section suivante.

4 PROTOTYPE

4.1 Description

i-LaTeX est un prototype d'éditeur de documents LaTeX doté de représentations intermédiaires interactives. Son interface ressemble à celle de la plupart des éditeurs LaTeX traditionnels (code source à gauche, PDF généré à droite), mais certains éléments du PDF sont interactifs, de telle sorte qu'en cliquer un affiche la représentation intermédiaire associée au morceau de code qui l'a généré (Figure 3). Cette visualisation peut alors être utilisée par les auteurs du document pour (1) visualiser des structures et des abstractions invisibles (*e.g.* la grille d'un tableau sans bordure visible) et (2) modifier le code source du document (*e.g.* réordonner les colonnes d'un tableau en manipulant sa grille). Selon la position de l'élément cliqué sur l'écran, sa représentation intermédiaire est affichée au-dessus ou en-dessous de celui-ci, de façon à le laisser le plus visible possible. Le reste du document est assombri jusqu'à ce que la représentation soit fermée en cliquant sur la croix ou sur le document assombri — déclenchant alors une recompilation du document LaTeX afin de mettre le PDF à jour. En outre, chaque visualisation affiche le nom et la position du morceau de code visualisé dans son en-tête. Ceux-ci peuvent être cliqués afin d'afficher le code correspondant dans l'éditeur de code. Nous avons implémenté trois types de RII : deux pour des éléments LaTeX courants (images et tableaux), et une pour un élément personnalisé (une grille de mise en page). Elles sont décrites plus en détail ci-après et illustrées par la Figure 4.

4.1.1 *Détail des visualisations.*

Images. Les images sont détectées par l'utilisation de la commande `includegraphics` (telle que fournie par le paquet `graphicx`[13]). La représentation intermédiaire d'une image (Figure 4(a)) affiche l'image à la même taille que l'image affichée dans le PDF. Elle peut être redimensionnée (à l'aide de la molette de la souris) et rognée (en redimensionnant le masque représenté par le rectangle avec une bordure sombre et une poignée). Ces manipulations mettent automatiquement à jour les paramètres de la commande en modifiant les valeurs des clés `width`, `height`, `trim` et `clip` (ou en les supprimant si elles ne sont plus nécessaires). Cette visualisation répond principalement à la difficulté de visualiser et de formaliser des dimensions (T4), mais elle facilite également la découverte de certains paramètres utiles (T2).

Tableaux. Les tableaux sont détectés par l'utilisation de l'environnement `tabular`. Leur représentation intermédiaire (Figure 4(b)) affiche le contenu brut de chaque « cellule »de l'environnement, qui peut être édité en double-cliquant sur la cellule. Le type de chaque colonne est affiché dans une ligne d'en-tête, et les colonnes et les lignes peuvent être réordonnées en cliquant longuement sur leurs en-têtes respectives et en les déplaçant. Cette visualisation permet

[13] https://ctan.org/pkg/graphicx?lang=en

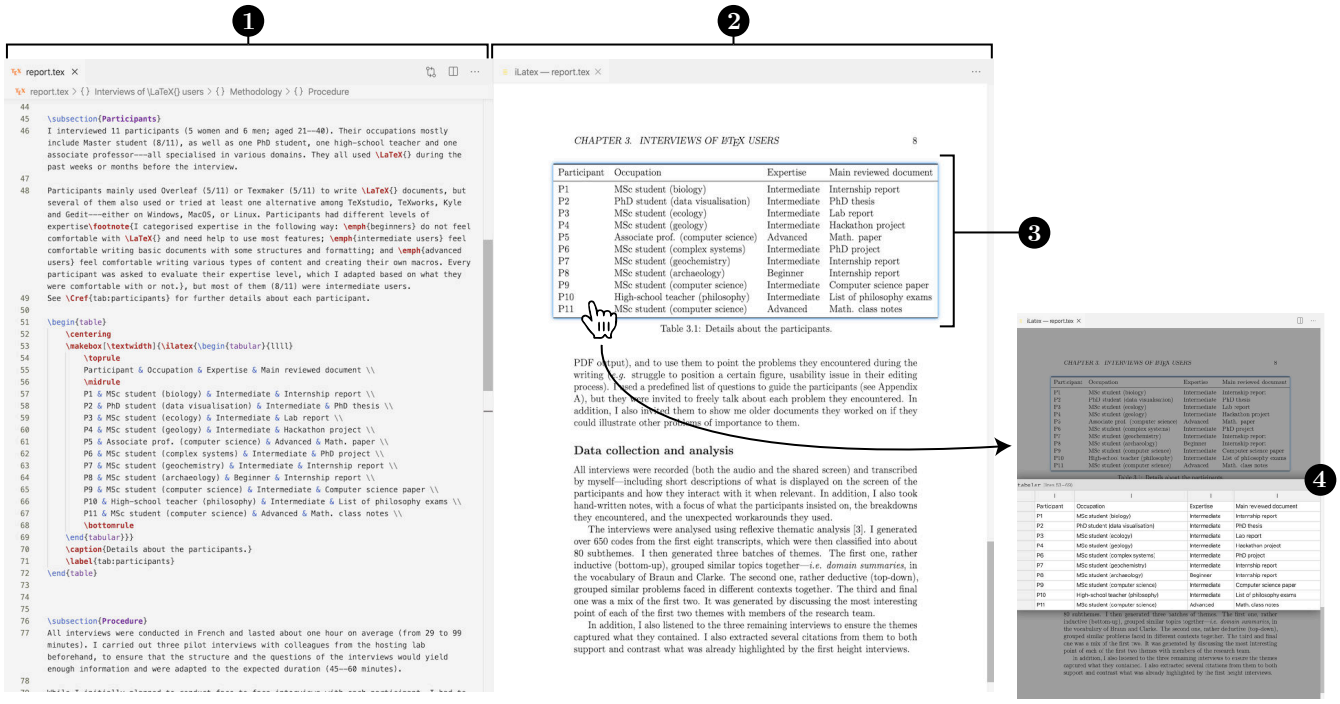


Figure 3: Interface du prototype. (1) Éditeur de code ; (2) Pré-visualisation du PDF généré ; (3) Tableau disposant d'une RII (cliquable) ; (4) Visualisation du code du tableau (affichée par dessus le PDF). La Figure 4 illustre les différentes visualisations.

de mettre en exergue une structure abstraite pouvant être invisible dans le PDF (T4). Elle rend également des transformations telles que la réorganisation de colonnes beaucoup plus faciles qu'elles ne le seraient dans un éditeur de code (T3).

Grilles de mise en page. Les grilles de mise en page sont détectées par l'utilisation de l'environnement `gridlayout`, un environnement personnalisé que nous avons développé afin d'illustrer l'utilité de *i*-LaTeX pour ce type de structure. Il permet d'utiliser une grille de mise en page (constituée de lignes et de cellules), qui permet de disposer localement du contenu de type varié (*e.g.* texte, images, tableaux) suivant une grille invisible. La représentation intermédiaire (Figure 4(c)) rend la grille visible. Les cellules peuvent être redimensionnées horizontalement en faisant glisser leur bordure droite, et les lignes peuvent être redimensionnées verticalement en faisant glisser leur bordure inférieure. En outre, cliquer une cellule affiche et sélectionne son contenu dans l'éditeur de code. Cette visualisation supporte l'édition de contenu structuré (T3) et la représentation concrète d'abstractions telles que des dimensions relatives (T4). Elle illustre également le potentiel de développement de nouvelles fonctionnalités pour LaTeX dont l'utilisation serait plus difficile ou désagréable sans RII.

4.2 Implémentation

Le prototype est implémenté en tant qu'extension de l'éditeur Visual Studio Code[14] (VSC), un éditeur de code open-source. L'extension est écrite en TypeScript, HTML et CSS et organisée selon le patron de conception MVC. Les particularités de son implémentation sont discutées ci-dessous.

4.2.1 Extraction du code à visualiser. Contrairement à la plupart des langages de programmation, LaTeX n'est pas un langage hors-contexte, mais un langage récursivement énumérable (avec des caractéristiques peu communes telles que le concept de *catcodes*, qui permet de modifier dynamiquement la signification de toute unité lexicale n'importe où dans le document). En théorie, il est donc impossible d'écrire un analyseur syntaxique destiné à structurer le code que l'on souhaite visualiser en utilisant des outils traditionnellement dédiés à ce type de tâche. Néanmoins, certaines conventions propres à LaTeX demeurent très couramment utilisées (*e.g.* la notion d'environnement). En pratique, il est donc possible d'écrire un analyseur syntaxique qui accepte une proportion raisonnable de documents LaTeX qui respectent lesdites conventions.

Pour les besoins du prototype, nous avons donc écrit un analyseur syntaxique de documents LaTeX le plus générique possible dont le but est de fournir juste assez d'informations et de structure pour identifier les morceaux de code à visualiser, tels que certaines commandes spécifiques et leurs paramètres. L'arbre de syntaxe abstraite ainsi généré est ensuite utilisé pour identifier les sous-arbres représentant des morceaux de code pertinents à visualiser (*e.g.* un environnement nommé `tabular`). Cette conception minimale a été choisie pour (1) maximiser le succès de l'analyse et (2) minimiser le temps d'exécution. En contrepartie, chaque visualisation est responsable d'effectuer une analyse statique du code plus approfondie si nécessaire.

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

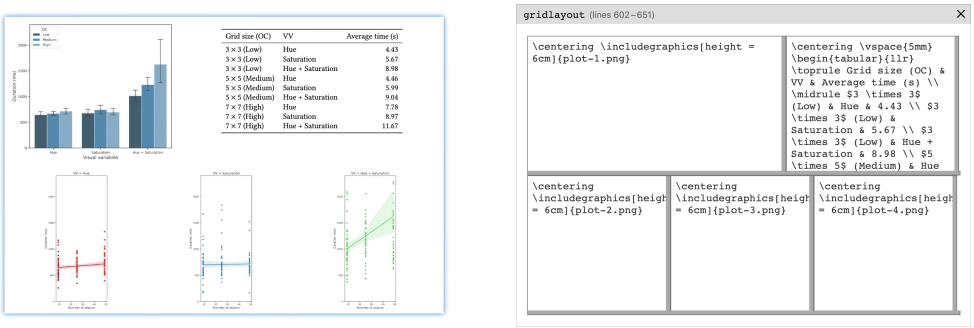
(a) Une image et sa visualisation

Non-English or Math	Frequency	Comments
Ø	1 in 1,000	For Swedish names
π	1 in 5	Common in math
\$	4 in 5	Used in business
Ψ_1^2	1 in 40,000	Unexplained usage

tabular (lines 462–471)

	c	c	l
	Non-English or Math	Comments	Frequency
	\O	For Swedish names	1 in 1,000
	π	Common in math	1 in 5
	\$	Used in business	4 in 5
	Ψ^2_1	Unexplained usage	1 in 40,000

(b) Un tableau et sa visualisation



(c) Une grille de mise en page et sa visualisation

Figure 4: Exemples des visualisations disponibles dans *i*-LaTeX. Les éléments interactifs du PDF tels qu'ils apparaissent dans *i*-LaTeX sont affichés à gauche, et les visualisations du code les ayant générés sont affichées à droite (les images ne sont pas à l'échelle). Chaque visualisation ayant été modifiée, le code a déjà été mis à jour, et le document sera recompilé dès que la visualisation sera fermée. (a) L'image a été rognée de façon à isoler le tableau central dans la visualisation — d'où la région éclaircie en dehors du rectangle. (b) Les deux dernières colonnes ont été inversées dans la visualisation. (c) La séparation entre les deux cellules de la première ligne a été déplacée vers la droite dans la visualisation.

4.2.2 Annotation du document PDF. Le format de fichier PDF — typiquement utilisé en sortie par les moteurs LaTeX — n'est pas conçu pour contenir des informations sur les régions du code LaTeX à l'origine des différents éléments qui le composent. Afin d'outrepasser cette limitation, nous avons choisi d'utiliser des annotations PDF contenant un identifiant unique et dont la boîte se superpose à celle de l'élément à associer à un morceau de code. Celles-ci sont générées en englobant le morceau de code à visualiser dans une commande `ilatex` personnalisée[15], comme dans `\ilatex{\includegraphics{fig.pdf}}`. Chaque utilisation de cette commande écrit également une nouvelle entrée dans un fichier regénéré à chaque compilation du document, qui contient des informations sur le morceau de code (*e.g.* sa position dans le code) et sur la valeur courante de certaines macros (*e.g.* `\linewidth`).

4.2.3 Affichage du PDF augmenté. Une fois le document compilé en un PDF annoté, celui-ci est affiché à l'aide d'un moteur de rendu personnalisé (basé sur PDF.js[16]). Celui-ci est responsable d'extraire toutes les annotations insérées par la commande `ilatex` et de les afficher sous forme de rectangles colorés autour des éléments disposant d'une visualisation. Lorsque l'un d'eux est cliqué, *i*-LaTeX identifie le morceau de code auquel correspond l'annotation afin d'afficher la bonne visualisation. Celui-ci est identifié en comparant l'identifiant attribué à l'annotation du PDF à ceux présents dans le fichier généré lors de la compilation du document.

4.3 Évaluation préliminaire

Étant donné la variabilité de l'expertise et des besoins de chaque utilisateur de LaTeX (comme en attestent les différents profils des participants interviewés — voir la Table 1), une étude quantitative de l'efficacité de *i*-LaTeX (*e.g.* vitesse d'édition, nombre d'essais-et-erreurs) ne nous semble pas être très pertinente. À l'inverse, une étude plus qualitative — telle qu'une étude de terrain — permettrait de mieux comprendre les contextes dans lesquels les RII sont préférées à l'édition du code et de découvrir des utilisations inattendues et des fonctionnalités manquantes. Cependant, le contexte sanitaire en place depuis l'émergence de la pandémie de Covid-19 ne nous a pour l'instant pas permis d'organiser une telle évaluation. En dépit de cela, une fois les interviews terminées, nous avons toutefois décidé de recueillir les avis des participants sur une vidéo d'une version précoce de *i*-LaTeX afin d'obtenir une première estimation de son potentiel. Conscients du risque de biais dus à cette

[15] Bien que cela nécessite l'utilisation d'une commande spéciale pour le moment, il convient de noter que puisque tout peut être redéfini dans LaTeX, les commandes et environnements existants pourraient être silencieusement modifiés afin que ceux-ci utilisent automatiquement la commande `ilatex`.

[16] https://github.com/mozilla/pdf.js/

approche, nous avons fait particulièrement attention à ne pas parler du prototype ni de la notion de RII avant la fin de chaque interview.

4.3.1 Méthodologie. Nous avons présenté à chaque participant une vidéo d'un prototype de *i*-LaTeX d'une durée d'environ deux minutes. Celle-ci présentait l'interface utilisateur du logiciel et l'utilisation de versions préliminaires des visualisations du code des images et des tableaux (semblables à celles présentées dans cette section). La vidéo étant muette, nous expliquions chaque action en voix off à l'aide d'un script. Une fois la vidéo terminée, nous avons demandé aux participants ce qu'ils avaient compris du système, s'ils aimeraient l'essayer, et s'il y avait d'autres types d'éléments du PDF ou de morceaux de code LaTeX qu'ils souhaiteraient manipuler d'une façon similaire. Ces évaluations ont été enregistrées dans les mêmes conditions que celles des interviews.

4.3.2 Résultats. La majorité des participants ont exprimé des réactions positives concernant la vidéo du prototype : « *Mais comment t'as codé ça ? C'est trop cool !* » (P1) ; « *vend ton truc à Overleaf […] qu'ils le rajoutent !* » (P7) Tous ont indiqué avoir compris le fonctionnement de *i*-LaTeX. Plusieurs ont également fait la remarque que cela semblait facile à utiliser : « *c'est vachement intuitif, […] ça fait référence à des compétences que les gens ont déjà* » (P4) ; « *je pense que c'est plus user-friendly que juste modifier le code* » (P8). Les utilisateurs les plus experts semblent eux aussi enthousiastes à l'idée d'utiliser le système. P5 a d'ailleurs insisté sur le fait que *i*-LaTeX ne cherche pas à supprimer l'accès au code (« *si ce n'était pas comme ça je ne me verrais pas interagir avec* »), et P11 a reconnu que *i*-LaTeX pourrait « *m'encourager à utiliser plus d'illustrations et de tableaux* » (P11). En outre, P4 a souligné le potentiel pédagogique des visualisations : « *si je devais donner un cours sur LaTeX, c'est un outil que j'utiliserais vraiment beaucoup je pense* », car il permet selon elle d'« *en apprendre un peu plus et un peu plus vite sur le code* » (P4).

Certains participants ont suggéré de nouvelles fonctionnalités pour les visualisations existantes (*e.g.* sélectionner l'alignement horizontal des cellules d'une colonne de tableau), et plusieurs d'entre eux ont également mentionné qu'ils aimeraient avoir un contrôle plus direct sur les positions et les marges des différents éléments de leurs documents — motivant ainsi le développement ultérieur de l'environnement `gridlayout`. Quelques participants ont estimé qu'ils n'interagiraient que peu avec les visualisations de code peu compliqué (*e.g.* les paramètres de la commande `includegraphics`), mais qu'ils seraient beaucoup plus intéressés par le système si celui-ci permettait de visualiser le code d'autres types de contenu. Les exemples recueillis comprennent les figures composées de plusieurs sous-figures (P4), les molécules créées avec le paquet `chemfig` (P7), la configuration du style des éléments bibliographiques en éditant directement l'un d'entre eux (P6), et l'édition localisée des longues formules mathématiques (P9). En outre, il est intéressant de noter que très peu de participants ont exprimé le souhait de pouvoir interagir avec le PDF dans le but de mettre en forme le document — allant ainsi dans le sens de la distinction entre RII et WYSIWYG.

4.4 Limitations

Bien que le prototype fonctionne dans sa forme actuelle, celui-ci souffre de plusieurs limitations qui en font plutôt une preuve de concept. D'une part, bien que la grammaire reconnue par l'analyseur syntaxique puisse être facilement étendue pour détecter l'utilisation d'autres commandes ou environnements (*e.g.* pour faire une RII de listes implémentées avec les environnement `itemize` et `enumerate`), celle-ci ne permet pas de reconnaître certaines constructions (*e.g.* les commandes utilisant des chevrons dans les présentations Beamer). D'autre part, les trois visualisations que nous avons implémentées sont limitées en terme de fonctionnalités (*e.g.* pas de fusion de cellules de tableau) et de robustesse (*e.g.* tous les types de colonne ne sont pas supportés).

En outre, une évaluation plus systématique et longitudinale de *i*-LaTeX est nécessaire pour mieux comprendre les apports et les limites des RII pour l'édition de langages de description de document. Néanmoins, bien que les avis que nous avons récoltés ne permettent pas de conclure de manière définitive sur l'apport en utilité ou en efficacité de *i*-LaTeX, il mettent en évidence l'intérêt que des utilisateurs de LaTeX semblent porter aux RII — soulignant l'importance de mener une évaluation plus approfondie d'un tel système.

5 CONCLUSION

Nous avons présenté le concept de représentations intermédiaires interactives, un nouveau genre d'interface utilisateur pour les éditeurs de langages de description de document. Nous avons motivé son utilité en réalisant une analyse thématique des difficultés rencontrées par 11 utilisateurs de LaTeX que nous avons interviewés, et nous l'avons également mis en œuvre à travers *i*-LaTeX, un éditeur LaTeX expérimental doté de RII. *i*-LaTeX a reçu des critiques majoritairement positives lors d'une évaluation préliminaire auprès des participants que nous avions interviewés, qui nous encouragent donc à poursuivre ce travail.

Parmi les directions possibles, quatre d'entre elles nous semblent particulièrement intéressantes à explorer. Premièrement, une évaluation plus complète de *i*-LaTeX nous paraît primordiale. Nous pensons qu'une étude de terrain auprès d'utilisateurs de *i*-LaTeX serait la plus adaptée pour évaluer le prototype de manière plus approfondie. Deuxièmement, une analyse des besoins spécifiques de certains utilisateurs (*e.g.* formules mathématiques, molécules) permettrait de doter les visualisations existantes de fonctionnalités supplémentaires adaptées et de développer de nouvelles visualisations répondant à d'autres besoins. Troisièmement, développer des métriques spécialisées permettrait de mieux estimer la robustesse de *i*-LaTeX (*e.g.* le taux de succès de l'analyse syntaxique sur un corpus de fichiers LaTeX[17]) et l'utilité potentielle de ses visualisations (*e.g.* la proportion de documents LaTeX contenant du code susceptible d'être visualisé). Quatrièmement, nous pensons qu'il est important d'appliquer le concept de RII à d'autres langages de description de documents que LaTeX (*e.g.* adapter la visualisation de tableau afin de l'utiliser avec des tableaux HTML ou Markdown). Cela permettrait non seulement de mieux comprendre le degré de généricité de ce type d'interface utilisateur, mais également le potentiel de réutilisation de ses visualisations.

[17] L'évaluation pourrait par exemple porter sur les archives des sources LaTeX des articles d'un laboratoire, ou encore sur celles publiées sur une plate-forme en ligne comme arXiv (https://arxiv.org/).

REMERCIEMENTS

Nous remercions Han (Evan) Han et Wendy Mackay pour leurs commentaires sur les premières versions de l'analyse thématique, ainsi que Jean-Philippe Rivière, Elizabeth Walton, Téo Sanchez et Alexandre Battut pour leurs critiques de la première version de cet article. Nous remercions également tous les participants interviewés pour leur temps. Ce travail à été en partie financé par : European Research Council (ERC) grant n° 695464 ONE : Unified Principles of Interaction.

RÉFÉRENCES

[1] 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

[2] 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

[3] Michel Beaudouin-Lafon. 2017. Towards Unified Principles of Interaction. In *Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter*. ACM, 1–2. https://doi.org/10.1145/3125571.3125602

[4] 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

[5] Neil C. C. Brown, Amjad Altadmri, and Michael Kolling. 2016. Frame-Based Editing : Combining the Best of Blocks and Text Programming. In *Proceedings of the 2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. IEEE, 47–53. https://doi.org/10.1109/LaTiCE.2016.16

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

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

[12] Philip J. Guo. 2013. Online Python Tutor : Embeddable Web-Based Program Visualization for CS Education. In *Proceeding of the 44th Technical Symposium on Computer Science Education - SIGCSE '13*. ACM, 579–584. https://doi.org/10.1145/2445196.2445368

[13] 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

[14] 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

[15] Jennifer Jacobs, Joel Brandt, Radomír Mech, and Mitchel Resnick. 2018. Extending Manual Drawing Practices with Artist-Centric Programming Tools. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*. ACM, 1–13. https://doi.org/10.1145/3173574.3174164

[16] Jeff Johnson and Richard J. Beach. 1988. Styles in Document Editing Systems. *Computer* 21, 1 (1988), 32–43. https://doi.org/10.1109/2.222115

[17] Mary Beth Kery, Donghao Ren, Kanit Wongsuphasawat, Fred Hohman, and Kayur Patel. 2020. The Future of Notebook Programming Is Fluid. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems (CHI EA '20)*. ACM, 1–8. https://doi.org/10.1145/3334480.3383085

[18] 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

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

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

[21] Andrew 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

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

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

[24] 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

[25] 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

[26] 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

[27] 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

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

[29] 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